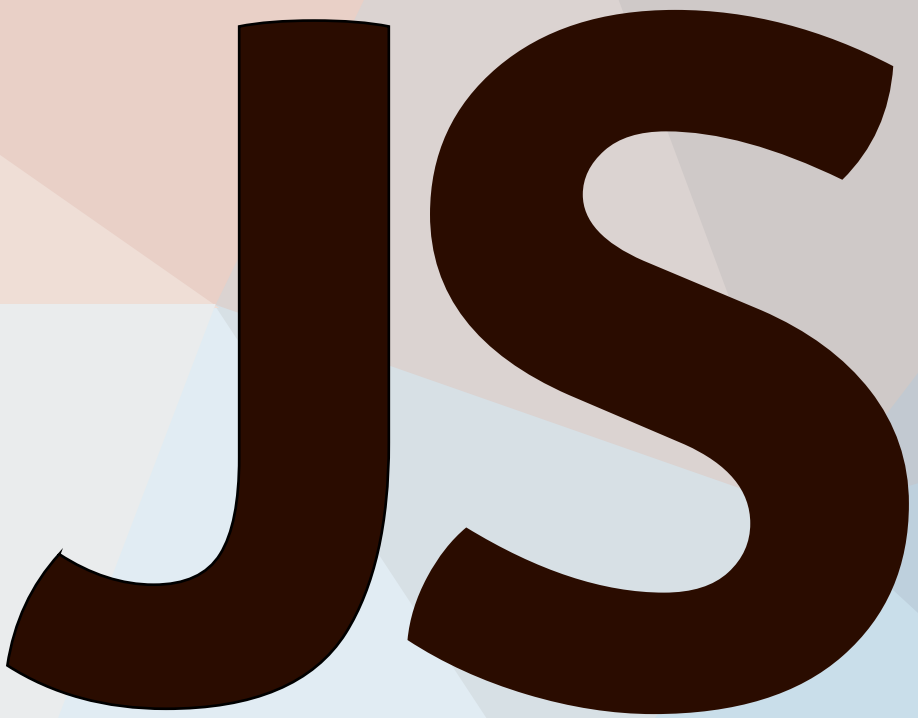
Часть 3

Дополнительно



Илья Кантор

2015

Дополнительно

Сборка от 17 марта 2017 г.

Последняя версия учебника находится на сайте learn.javascript.ru.

Мы постоянно работаем над улучшением учебника. При обнаружении ошибок пишите о них на нашем баг-трекере.

Веб-компоненты: взгляд в будущее

Веб-компоненты – «платформа будущего»: совокупность стандартов, которые позволяют описывать новые типы DOM-элементов, со своими свойствами и методами, инкапсулировать их DOM и стили.

С высоты орбитального полёта

Этот раздел посвящён набору современных браузерных стандартов, описывающих создание «веб-компонент» (web components).

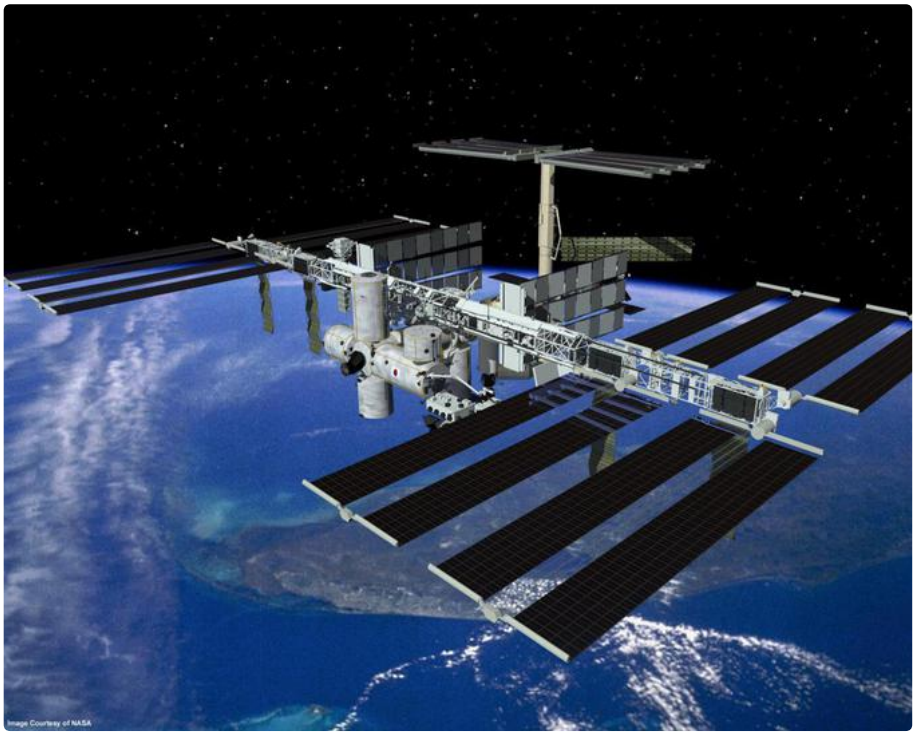
На текущий момент эти стандарты – в разработке, браузеры по-разному поспевают за их развитием. Статьи можно читать в любом браузере, но для тестирования я бы рекомендовал использовать Chrome Canary, поскольку главной «движущей силой» этих стандартов являются сотрудники Google и, как правило, этот браузер реализует последний вариант спецификации.

Понимание этих стандартов важно потому, что это «взгляд в будущее», в то, какой станет интернет-разработка. Сейчас уже существуют некоторые полифиллы, частично реализующие их.

Идеи веб-компонент взяты не с пустого места. Они эксплуатируются в более «приземлённых», текущих подходах к разработке.

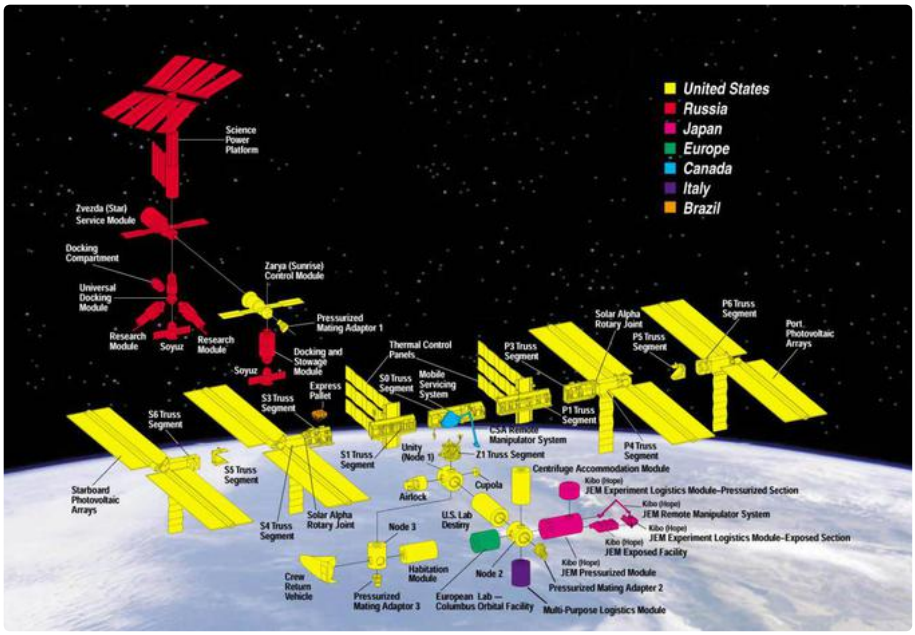
Что общего, между...

Прежде, чем мы перейдём к веб-компонентам, посмотрим на одну очень классную штуку:



Да, это Международная Космическая Станция (МКС).

И вот, как она, приблизительно, устроена:



МКС:

- Состоит из множества компонентов.
- Каждый компонент – в свою очередь состоит из множества деталей внутри.
- Эти компоненты очень сложны, они гораздо сложнее, чем большинство сайтов.
- Их разрабатывают команды из разных стран, разговаривающие на разных языках.

...И эта штука летает!

За счёт чего строятся настолько сложные вещи?

Что мы могли бы позаимствовать, чтобы наша разработка была столь же надёжной и масштабируемой? Ну, или по крайней мере, близко к этому...

Компонентная архитектура

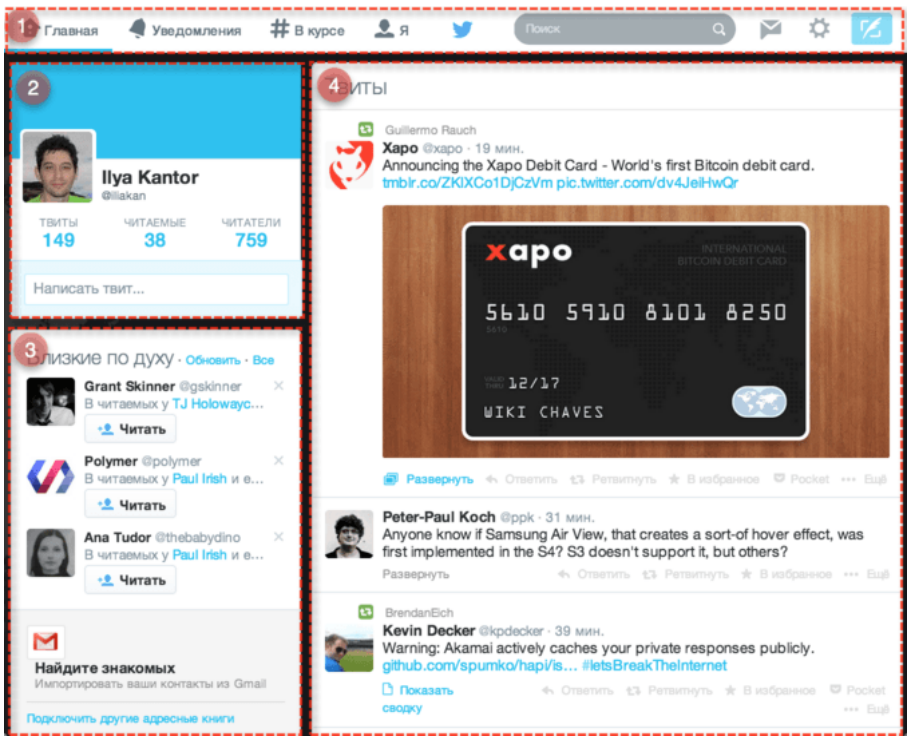
Ключевое правило при разработке сложных вещей: «Никогда не делайте сложные вещи».

Если что-то становится сложным – разбиваем это на части попроще и связываем их наиболее очевидным способом.

Хороший архитектор – это как раз тот, кто умеет делать сложное простым.

Любой сложный интерфейс мы делим на компоненты – сущности, для которых мы можем максимально чётко и понятно указать, что это такое и что оно умеет делать.

Посмотрим на какой-нибудь более-менее сложный сайт, например на Twitter.



Он естественным образом распадается на компоненты:

Для наглядности они обведены красной рамкой:

1. «Главное Меню»
2. «Об Авторе»
3. «Близкие по духу»
4. «Твиты»

Как мы решаем, что именно выделять в компонент? Это нам подсказывает опыт и здравый смысл.

В случае с твиттером разбиение на компоненты особенно очевидно. Страница «сама распадается» на чётко очерченные блоки, каждый из которых выполняет свою роль.

Если представить каждый компонент HTML-тегом, то страница будет выглядеть примерно так (выделены нестандартные теги):

```
<header>
  <top-menu/>
</header>
<aside>
  <author-info/>
  <congenial-info/>
</aside>
<main>
  <tweets-list/>
</main>
```

До недавнего времени, чтобы так описать страницу, требовались специальные JavaScript-фреймворки. Такой фреймворк позволял описывать «свои» теги, которые, после обработки фреймворком, становились JavaScript-объектами.

Веб-компоненты (Web Components) – это не один стандарт, а целая платформа, комплекс стандартов, которые вместе добавляют в браузер технологии для удобной реализации компонент.

Если глядеть «сверху», то веб-компоненты – это возможность добавлять свои элементы в браузер, например `document.createElement("tweets-list")`, которые описываются с помощью классов JavaScript, могут иметь свои методы и свойства.

Также «под капотом» кроются расширенные возможности по инкапсуляции поддерева DOM и стилей, по генерации событий и многое другое, что мы рассмотрим далее.

Пользовательские элементы: Custom Elements

Платформа «веб-компоненты» включает в себя несколько стандартов Web Components, которые находятся в разработке.

Начнём мы со стандарта Custom Elements, который позволяет создавать свои типы элементов.

Зачем Custom Elements?

Критично настроенный читатель скажет: «Зачем ещё стандарт для своих типов элементов? Я могу создать любой элемент и прямо сейчас! В любом из современных браузеров можно писать любой HTML, используя свои теги: `<mytag>`. Или создавать элементы из JavaScript при помощи `document.createElement('mytag')`.»

Однако, по умолчанию элемент с нестандартным названием (например `<mytag>`) воспринимается браузером, как нечто неопределённо-непонятное. Ему соответствует класс HTMLUnknownElement, и у него нет каких-либо особых методов.

Стандарт Custom Elements позволяет описывать для новых элементов свои свойства, методы, объявлять свой DOM, подобие конструктора и многое другое.

Давайте посмотрим это на примерах.

> **Для примеров рекомендуется Chrome**
>
> Так как спецификация не окончательна, то для запуска примеров рекомендуется использовать Google Chrome, лучше – последнюю сборку Chrome Canary, в которой, как правило, отражены последние изменения.

Новый элемент

Для описания нового элемента используется вызов `document.registerElement(имя, { prototype: прототип })`.

Здесь:

- `имя` – имя нового тега, например `"mega-select"`. Оно обязано содержать дефис `"-"`. Спецификация требует дефис, чтобы избежать в будущем конфликтов со стандартными элементами HTML. Нельзя создать элемент `timer` или `myTimer` – будет ошибка.
- `прототип` – объект-прототип для нового элемента, он должен наследовать от `HTMLElement`, чтобы у элемента были стандартные свойства и методы.

Вот, к примеру, новый элемент `<my-timer>`:

```
<script>
  // прототип с методами для нового элемента
  var MyTimerProto = Object.create(HTMLElement.prototype);
  MyTimerProto.tick = function() { // свой метод tick
    this.innerHTML++;
  };

  // регистрируем новый элемент в браузере
  document.registerElement("my-timer", {
```

```
    prototype: MyTimerProto
  });
</script>

<!-- теперь используем новый элемент -->
<my-timer id="timer">0</my-timer>

<script>
  // вызовем метод tick() на элементе
  setInterval(function() {
    timer.tick();
  }, 1000);
</script>
```

Использовать новый элемент в HTML можно и до его объявления через `registerElement`.

Для этого в браузере предусмотрен специальный режим «обновления» существующих элементов.

Если браузер видит элемент с неизвестным именем, в котором есть дефис `-` (такие элементы называются «unresolved»), то:

- Он ставит такому элементу специальный CSS-псевдокласс `:unresolved`, для того, чтобы через CSS можно было показать, что он ещё «не подгрузился».
- При вызове `registerElement` такие элементы автоматически обновятся до нужного класса.

В примере ниже регистрация элемента происходит через 2 секунды после его появления в разметке:

```
<style>
  /* стиль для :unresolved элемента (до регистрации) */
  hello-world:unresolved {
    color: white;
  }

  hello-world {
    transition: color 3s;
  }
</style>

<hello-world id="hello">Hello, world!</hello-world>

<script>
  // регистрация произойдёт через 2 сек
  setTimeout(function() {
    document.registerElement("hello-world", {
      prototype: {
        __proto__: HTMLElement.prototype,
        sayHi: function() { alert('Привет!'); }
      }
    });

    // у нового типа элементов есть метод sayHi
    hello.sayHi();
  }, 2000);
</script>
```

Можно создавать такие элементы и в JavaScript – обычным вызовом `createElement`:

```
var timer = document.createElement('my-timer');
```

Расширение встроенных элементов

Выше мы видели пример создания элемента на основе базового `HTMLElement`. Но можно расширить и другие, более конкретные HTML-элементы.

Для расширения встроенных элементов у `registerElement` предусмотрен параметр `extends`, в котором можно задать, какой тег мы расширяем.

Например, кнопку:

```
<script>
  var MyTimerProto = Object.create(HTMLButtonElement.prototype);
  MyTimerProto.tick = function() {
    this.innerHTML++;
  };

  document.registerElement("my-timer", {
    prototype: MyTimerProto,
    extends: 'button'
  });
</script>

<button is="my-timer" id="timer">0</button>

<script>
  setInterval(function() {
    timer.tick();
  }, 1000);

  timer.onclick = function() {
    alert("Текущее значение: " + this.innerHTML);
  };
</script>
```

Важные детали:

Прототип теперь наследует не от **`HTMLElement`**, а от **`HTMLButtonElement`**

Чтобы расширить элемент, нужно унаследовать прототип от его класса.

В HTML указывается при помощи атрибута **`is="..."`**

Это принципиальное отличие разметки от обычного объявления без `extends`. Теперь `<my-timer>` работать не будет, нужно использовать исходный тег и `is`.

Работают методы, стили и события кнопки.

При клике на кнопку её не отличишь от встроенной. И всё же, это новый элемент, со своими методами, в данном случае `tick`.

При создании нового элемента в JS, если используется `extends`, необходимо указать и исходный тег в том числе:

```
var timer = document.createElement("button", "my-timer");
```

Жизненный цикл

В прототипе своего элемента мы можем задать специальные методы, которые будут вызываться при создании, добавлении и удалении элемента из DOM:

`createdCallback`	Элемент создан
`attachedCallback`	Элемент добавлен в документ
`detachedCallback`	Элемент удалён из документа
`attributeChangedCallback(name, prevValue, newValue)`	Атрибут добавлен, изменён или удалён

Как вы, наверняка, заметили, `createdCallback` является подобием конструктора. Он вызывается только при создании элемента, поэтому всю дополнительную инициализацию имеет смысл описывать в нём.

Давайте используем `createdCallback`, чтобы инициализировать таймер, а `attachedCallback` – чтобы автоматически запускать таймер при вставке в документ:

```
<script>
  var MyTimerProto = Object.create(HTMLElement.prototype);

  MyTimerProto.tick = function() {
    this.timer++;
    this.innerHTML = this.timer;
  };

  MyTimerProto.createdCallback = function() {
    this.timer = 0;
  };

  MyTimerProto.attachedCallback = function() {
    setInterval(this.tick.bind(this), 1000);
  };

  document.registerElement("my-timer", {
    prototype: MyTimerProto
  });
</script>

<my-timer id="timer">0</my-timer>
```

Итого

Мы рассмотрели, как создавать свои DOM-элементы при помощи стандарта Custom Elements.

Далее мы перейдём к изучению дополнительных возможностей по работе с DOM.

Shadow DOM

Спецификация Shadow DOM является отдельным стандартом. Частично он уже используется для обычных DOM-элементов, но также применяется для создания веб-компонентов.

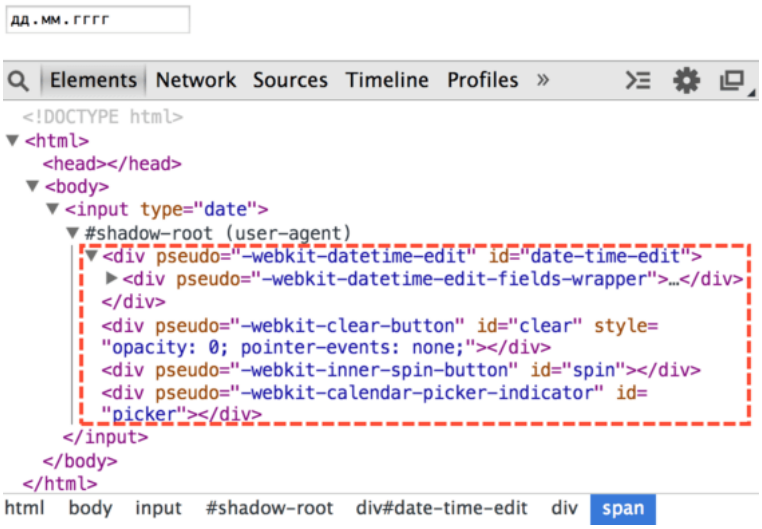
Shadow DOM – это внутренний DOM элемента, который существует отдельно от внешнего документа. В нём могут быть свои ID, свои стили и так далее. Причём снаружи его, без применения специальных техник, не видно, поэтому не возникает конфликтов.

Внутри браузера

Концепция Shadow DOM начала применяться довольно давно внутри самих браузеров. Когда браузер показывает сложные элементы управления, наподобие слайдера `<input type="range">` или календаря `<input type="date">` – внутри себя он конструирует их из самых обычных стилизованных `<div>`, `<span>` и так далее.

С первого взгляда они незаметны, но если в настройках Chrome Development Tools выбрать показ Shadow DOM, то их можно легко увидеть.

Например, вот такое содержимое будет у `<input type="date">`:



То, что находится под `#shadow-root` – это и есть Shadow DOM.

Получить элементы из Shadow DOM можно только при помощи специальных JavaScript-вызовов или селекторов. Это не обычные дети, а намного более мощное средство отделения содержимого.

В Shadow DOM выше можно увидеть полезный атрибут `pseudo`. Он нестандартный, существует по историческим причинам. С его помощью можно стилизовать подэлементы через CSS, например, сделаем поле редактирования даты красным:

```
<style>
input::-webkit-datetime-edit {
  background: red;
}
</style>

<input type="date">
```

Ещё раз заметим, что `pseudo` – нестандартный атрибут. Если говорить хронологически, то сначала браузеры начали экспериментировать внутри себя с инкапсуляцией внутренних DOM-структур, а уже потом, через некоторое время, появился стандарт Shadow DOM, который позволяет делать то же самое разработчикам.

Далее мы рассмотрим работу с Shadow DOM из JavaScript, по стандарту Shadow DOM.

Создание Shadow DOM

Shadow DOM можно создать внутри любого элемента вызовом `elem.createShadowRoot()`.

Например:

```
<p id="elem">Доброе утро, страна!</p>

<script>
  var root = elem.createShadowRoot();
  root.innerHTML = "<p>Привет из подполья!</p>";
</script>
```

Если вы запустите этот пример, то увидите, что изначальное содержимое элемента куда-то исчезло и показывается только «Привет из подполья!». Это потому, что у элемента есть Shadow DOM.

С момента создания Shadow DOM обычное содержимое (дети) элемента не отображается, а показывается только Shadow DOM.

Внутрь этого Shadow DOM, при желании, можно поместить обычное содержимое. Для этого нужно указать, куда. В Shadow DOM это делается через «точку вставки» (insertion point). Она объявляется при помощи тега `<content>`, например:

```
<p id="elem">Доброе утро, страна!</p>

<script>
  var root = elem.createShadowRoot();
  root.innerHTML = "<h3><content></content></h3> <p>Привет из подполья!</p>";
</script>
```

Теперь вы увидите две строчки: «Доброе утро, страна!» в заголовке, а затем «Привет из подполья».

Shadow DOM примера выше в инструментах разработки:

```
▼ <p id="elem">
  ▼ #shadow-root
    ▼ <h3>
        <content></content>
      </h3>
      <p>Привет из подполья!</p>
    "Доброе утро, страна!"
  </p>
```

Важные детали:

- Тег `<content>` влияет только на отображение, он не перемещает узлы физически. Как видно из картинки выше, текстовый узел «Доброе утро, страна!» остался внутри `p#elem` . Его можно даже получить при помощи `elem.firstElementChild` .
- Внутри `<content>` показывается не элемент целиком `<p id="elem">` , а его содержимое, то есть в данном случае текст «Доброе утро, страна!».

В **`<content>`** атрибутом **`select`** можно указать конкретный селектор содержимого, которое нужно переносить. Например, **`<content select="h3"></content>`** перенесёт только заголовки.

Внутри Shadow DOM можно использовать `<content>` много раз с разными значениями `select` , указывая таким образом, где конкретно какие части исходного содержимого разместить. Но при этом дублирование узлов невозможно. Если узел показан в одном `<content>` , то в следующем он будет пропущен.

Например, если сначала идёт `<content select="h3.title">` , а затем `<content select="h3">` , то в первом `<content>` будут показаны заголовки `<h3>` с классом `title` , а во втором – все остальные, кроме уже показанных.

В примере выше тег `<content></content>` внутри пуст. Если в нём указать содержимое, то оно будет показано только в том случае, если узлов для вставки нет. Например потому что ни один узел не подпал под указанный `select` , или все они уже отображены другими, более ранними `<content>` .

Например:

```
<section id="elem">
  <h1>Новости</h1>
  <article>Жили-были <i>старик со старухой</i>, но недавно...</article>
</section>

<script>
  var root = elem.createShadowRoot();

  root.innerHTML = "<content select='h1'></content> \
   <content select='.author'>Без автора.</content> \
   <content></content>";

</script>

<button onclick="alert(root.innerHTML)">root.innerHTML</button>
```

При запуске мы увидим, что:

- Первый `<content select='h1'>` выведет заголовок.
- Второй `<content select=".author">` вывел бы автора, но так как такого элемента нет – выводится содержимое самого `<content select=".author">` , то есть «Без автора».
- Третий `<content>` выведет остальное содержимое исходного элемента – уже без заголовка `<h1>` , он выведен ранее!

Ещё раз обратим внимание, что `<content>` физически не перемещает узлы по DOM. Он только показывает, где их отображать, а также, как мы увидим далее, влияет на применение стилей.

Корень shadowRoot

После создания корень внутреннего DOM-дерева доступен как `elem.shadowRoot` .

Он представляет собой специальный объект, поддерживающий основные методы CSS-запросов и подробно описанный в стандарте как ShadowRoot.

Если нужно работать с содержимым в Shadow DOM, то нужно перейти к нему через `elem.shadowRoot` . Можно и создать новое Shadow DOM-дерево из JavaScript, например:

```
<p id="elem">Доброе утро, страна!</p>

<script>
  // создать новое дерево Shadow DOM для elem
  var root = elem.createShadowRoot();

  root.innerHTML = "<h3><content></content></h3> <p>Привет из подполья!</p> <hr>";
</script>

<script>
  // прочитать данные из Shadow DOM для elem
  var root = elem.shadowRoot;
  // Привет из подполья!
  document.write("<p>p:" + root.querySelector('p').innerHTML);
  // пусто, так как физически узлы - вне content
  document.write("<p>content:" + root.querySelector('content').innerHTML);
</script>
```

⚠ Внутрь встроенных элементов так «залезть» нельзя

На момент написания статьи `shadowRoot` можно получить только для Shadow DOM, созданного описанным выше способом, но не встроенного, как в элементах типа `<input type="date">`.

Итого

Shadow DOM – это средство для создания отдельного DOM-дерева внутри элемента, которое не видно снаружи без применения специальных методов.

- Ряд браузерных элементов со сложной структурой уже имеют Shadow DOM.
- Можно создать Shadow DOM внутри любого элемента вызовом `elem.createShadowRoot()`. В дальнейшем его корень будет доступен как `elem.shadowRoot`. У встроенных элементов он недоступен.
- Как только у элемента появляется Shadow DOM, его изначальное содержимое скрывается. Теперь показывается только Shadow DOM, который может указать, какое содержимое хозяина куда вставлять, при помощи элемента `<content>`. Можно указать селектор `<content select="селектор">` и размещать разное содержимое в разных местах Shadow DOM.
- Элемент `<content>` перемещает содержимое исходного элемента в Shadow DOM только визуально, в структуре DOM оно остаётся на тех же местах.

Подробнее спецификация описана по адресу http://w3c.github.io/webcomponents/spec/shadow/.

Далее мы рассмотрим работу с шаблонами, которые также являются частью платформы Web Components и не заменяют существующие шаблонные системы, но дополняют их важными встроенными в браузер возможностями.

Шаблоны <template>

Элемент `<template>` предназначен для хранения «образца» разметки, невидимого и предназначенного для вставки куда-либо.

Конечно, есть много способов записать произвольный невидимый текст в HTML. В чём же особенность `<template>`?

Его отличие от обычных тегов в том, что его содержимое обрабатывается особым образом. Оно не только скрыто, но и считается находящимся вообще «вне документа». А при вставке автоматически «оживает», выполняются из него скрипты, начинает проигрываться видео и т.п.

Содержимое тега `<template>`, в отличие, к примеру, от шаблонов или `<script type="неизвестный тип">`, обрабатывается браузером. А значит, должно быть корректным HTML.

Оно доступно как `DocumentFragment` в свойстве тега `content`. Предполагается, что мы, при необходимости, возьмём `content` и вставим, куда надо.

Вставка шаблона

Пример вставки шаблона `tmpl` в Shadow DOM элемента `elem`:

```
<p id="elem">
  Доброе утро, страна!</p>

<template id="tmpl">
  <h3><content></content></h3>
  <p>Привет из подполья!</p>
  <script>
    document.write('...document.write:Новость!');
  </script>
</template>

<script>
  var root = elem.createShadowRoot();
  root.appendChild(tmpl.content.cloneNode(true));
</script>
```

У нас получилось, что:

1. В элементе `#elem` содержатся данные в некоторой оговорённой разметке.
2. Шаблон `#tmpl` указывает, как их отобразить, куда и в какие HTML-теги завернуть содержимое `#elem`.
3. Здесь шаблон показывается в Shadow DOM тега. Технически, это не обязательно, шаблон можно использовать и без Shadow DOM, но тогда не сработает тег `<content>`.

Можно также заметить, что скрипт из шаблона выполнился. Это важнейшее отличие вставки шаблона от вставки HTML через `innerHTML` и от обычного `DocumentFragment`.

Также мы вставили не сам `tmpl.content`, а его клон. Это обычная практика, чтобы можно было использовать один шаблон много раз.

Итого

Тег `<template>` не призван заменить системы шаблонизации. В нём нет хитрых операторов итерации, привязок к данным.

Его основная особенность – это возможность вставки «живого» содержимого, вместе со скриптами.

И, конечно, мелочь, но удобно, что он не требует никаких библиотек.

Стили и селекторы

Стилизация Shadow DOM покрывается более общей спецификацией «CSS Scoping».

По умолчанию стили внутри Shadow DOM относятся только к его содержимому.

Например:

```
<p>Жили мы тихо-мирно, и тут...</p>

<p id="elem">Доброе утро, страна!</p>

<template id="tmpl">
  <style>
    p {
      color: red;
    }
  </style>
  <h3><content></content></h3>
  <p>Привет из подполья!</p>
</template>

<script>
  var root = elem.createShadowRoot();
  root.appendChild(tmpl.content.cloneNode(true));
</script>
```

При запуске окрашенным в красный цвет окажется только `<p>` внутри Shadow DOM. Обратим внимание, окрасился именно тот элемент, который находится непосредственно в Shadow DOM. А элементы, которые отображены в Shadow DOM при помощи `<content>`, этот стиль не получили – у них есть свои, заданные на внешней странице.

Внешний стиль для Shadow DOM

Граница между Shadow DOM и основным DOM, хоть и существует, но при помощи специальных селекторов её можно переходить.

Если нужно с основной страницы стилизовать или выбрать элементы внутри Shadow DOM, то можно использовать селекторы:

- **`::shadow`** – выбирает корень Shadow DOM.

  Выбранный элемент сам по себе не создаёт CSS box, но служит отправной точкой для дальнейшей выборки уже внутри дерева Shadow DOM.

  Например, `#elem::shadow > div` найдёт внутри Shadow DOM `#elem` элементы `div` первого уровня.

- **`>>>`** – особого вида CSS-селектор для всех элементов Shadow DOM, который полностью игнорирует границы между DOM'ами, включая вложенные подэлементы, у которых тоже может быть свой Shadow DOM.

  Например, `#elem >>> span` найдёт все `span` внутри Shadow DOM `#elem`, но кроме того, если в `#elem` есть подэлементы, у которых свой Shadow DOM, то оно продолжит поиск в них.

  Вот пример, когда внутри одного Shadow DOM есть `<input type="date">`, у которого тоже есть Shadow DOM:

  ```
  <style>
    #elem::shadow span {
      /* для span только внутри Shadow DOM #elem */

      border-bottom: 1px dashed blue;
    }

    #elem >>> * {
      /* для всех элементов внутри Shadow DOM #elem и далее внутри input[type=date] */

      color: red;
    }
  </style>

  <p id="elem"></p>

  <script>
    var root = elem.createShadowRoot();
    root.innerHTML = "<span>Текущее время:</span> <input type='date'>";
  </script>
  ```

- Кроме того, на Shadow DOM действует обычное CSS-наследование, если свойство поддерживает его по умолчанию.

  В этом примере CSS-стили для `body` наследуются на внутренние элементы, включая Shadow DOM:

  ```
  <style>
    body {
      color: red;
      font-style: italic;
    }
  </style>
  <p id="elem"></p>
  <script>
    elem.createShadowRoot().innerHTML = "<span>Привет, мир!</span>";
  </script>
  ```

  Внутренний элемент станет красным курсивом.

Нельзя получить содержимое встроенных элементов

Описанные CSS-селекторы можно использовать не только в CSS, но и в `querySelector`.

Исключением являются встроенные элементы типа `<input type="date">`, для которых CSS-селекторы работают, но получить их содержимое нельзя.

Например:

```
<p id="elem"></p>

<script>
  var root = elem.createShadowRoot();
  root.innerHTML = "<span>Текущее время:</span> <input type='date'>";

  // выберет только span из #elem
  // вообще-то, должен выбрать span и из вложенных Shadow DOM,
  // но для встроенных элементов - не умеет
  alert( document.querySelectorAll('#elem::shadow span').length ); // 1
</script>
```

Стиль в зависимости от хозяина

Следующие селекторы позволяют изнутри Shadow DOM выбрать внешний элемент («элемент-хозяин»):

- `:host` выбирает элемент-хозяин, в котором, живёт Shadow DOM.

  Хозяин :host выбирается именно в контексте Shadow DOM.

  То есть, это доступ не к внешнему элементу, а, скорее, к корню текущего Shadow DOM.

  После `:host` мы можем указать селекторы и стили, которые нужно применить, если хозяин удовлетворяет тому или иному условию, например:

  ```
  <style>
    :host > p {
      color: green;
    }
  </style>
  ```

  Этот селектор сработает для `<p>` первого уровня внутри Shadow DOM.

- `:host(селектор хозяина)` выбирает элемент-хозяин, если он подходит под селектор.

  Этот селектор используется для темизации хозяина «изнутри», в зависимости от его классов и атрибутов. Он отлично добавляет просто `:host`, например:

  ```
  :host p {
    color: green;
  }

  :host(.important) p {
    color: red;
  }
  ```

  Здесь параграфы будут иметь `color:green`, но если у хозяина класс `.important`, то `color:red`.

- `:host-context(селектор хозяина)` выбирает элемент-хозяин, если какой-либо из его родителей удовлетворяет селектору, например:

  ```
  :host-context(h1) p {
    /* селектор сработает для p, если хозяин находится внутри h1 */
  }
  ```

  Это используется для расширенной темизации, теперь уже не только в зависимости от его атрибутов, но и от того, внутри каких элементов он находится.

Пример использования селектора `:host()` для разной расцветки Shadow DOM-сообщения, в зависимости от того, в каком оно `<p>`:

```
<p class="message info">Доброе утро, страна!</p>

<p class="message warning">Внимание-внимание! Говорит информбюро!</p>

<template id="tmpl">
  <style>
  .content {
    min-height: 20px;
    padding: 19px;
    margin-bottom: 20px;
    background-color: #f5f5f5;
    border: 1px solid #e3e3e3;
    border-radius: 4px;
    box-shadow: inset 0 1px 1px rgba(0, 0, 0, .05);
  }

  :host(.info) .content {
    color: green;
```

```
  }

  :host(.warning) .content {
    color: red;
  }

  </style>
  <div class="content"><content></content></div>
</template>

<script>
var elems = document.querySelectorAll('p.message');

elems[0].createShadowRoot().appendChild( tmpl.content.cloneNode(true) );
elems[1].createShadowRoot().appendChild( tmpl.content.cloneNode(true) );
</script>
```

Стиль для content

Тег `<content>` не меняет DOM, а указывает, что где показывать. Поэтому если элемент изначально находится в элементе-хозяине – внешний документ сохраняет к нему доступ.

К нему будут применены стили и сработают селекторы, всё как обычно.

Например, здесь применится стиль для `<span>`:

```
<style>
  span { text-decoration: underline; }
</style>

<p id="elem"><span>Доброе утро, страна!</span></p>

<template id="tmpl">
  <h3><content></content></h3>
  <p>Привет из подполья!</p>
</template>

<script>
  elem.createShadowRoot().appendChild( tmpl.content.cloneNode(true) );
</script>
```

В примере выше заголовок «Доброе утро, страна!», который пришёл как `<span>` из внешнего документа, будет подчёркнут,

Итак, стили основного DOM-дерева применяются, всё в порядке.

Но что, если Shadow DOM тоже «имеет виды» на `<content>` и хочет стилизовать вставленное? Это тоже возможно.

Для обращения к «содержимому» **`<content>`** из стилей внутри Shadow DOM используется псевдоэлемент **`::content`**.

Например, изнутри Shadow DOM селектор `content[select="h1"]::content span` найдёт элемент `<content select="h1">` и *в его содержимом* отыщет `<span>`.

В примере ниже селектор `::content span` стилизует все `<span>` внутри всех `<content>`:

```
<style>
  span { text-decoration: underline; }
</style>

<p id="elem"><span>Доброе утро, страна!</span></p>

<template id="tmpl">
  <style>
    ::content span { color: green; }
  </style>
  <h3><content></content></h3>
  <span>Привет из подполья!</span>
</template>

<script>
  elem.createShadowRoot().appendChild( tmpl.content.cloneNode(true) );
</script>
```

Текст внутри `<h3>` – зелёный и подчёркнутый одновременно, но стилизуется именно тот `<span>`, который показан в `<content>`, а тот, который просто в Shadow DOM – нет.

Приоритет селекторов рассчитывается по обычным правилам специфичности, если же приоритеты стилей на странице и в Shadow DOM равны, то, как описано в секции Cascading, побеждает страница, а для `!important` -стиля побеждает Shadow DOM.

Итого

По умолчанию стили и селекторы из DOM-дерева действуют только на те элементы, в которых сами находятся.

Границу можно преодолевать, причём проще, естественно, от родителя к Shadow DOM, чем наоборот:

- Снаружи можно выбирать и стилизовать элементы внутри Shadow DOM – при помощи селекторов `::shadow` и `>>>`.
- Изнутри Shadow DOM можно стилизовать не только то, что изначально в Shadow DOM, но и узлы, показываемые в `<content>`.
- Также можно ставить стиль в зависимость от хозяина при помощи селекторов `::host`, `::host-context`, но выбирать и стилизовать произвольные теги внутри хозяина нельзя.

Импорты

Новая спецификация «HTML Imports» описывает, как вставить один документ в другой при помощи HTML-тега `<link rel="import">`.

Зачем?

Мы ведь и так можем вставлять документ в документ, при помощи `<iframe>`, зачем нужен ещё какой-то импорт? Что не так с `iframe`?

...С `iframe` всё так. Однако, по своему смыслу `iframe` – это отдельный документ.

- Для `iframe` создаётся полностью своё окружение, у него свой объект `window` и свои переменные.
- Если `iframe` загружен с другого домена, то взаимодействие с ним возможно только через `postMessage`.

Это хорошо, когда нужно действительно в одной странице отобразить содержимое другой.

А что, если нужно встроить другой документ как естественную часть текущего? С единым скриптовым пространством, едиными стилями, но при этом – другой документ.

Например, это нужно для подгрузки внешних частей документа (веб-компонент) снаружи. И желательно не иметь проблем с разными доменами: если уж мы действительно хотим подключить HTML с одного домена в страницу на другом – мы должны иметь возможность это сделать без «плясок с бубном».

Иначе говоря, `<link rel="import">` – это аналог `<script>`, но для подключения полноценных документов, с шаблонами, библиотеками, веб-компонентами и т.п. Всё станет понятнее, когда мы посмотрим детали.

Пример вставки

Синтаксис:

```
<link rel="import" href="http://site.com/document.html">
```

- В отличие от `<iframe>` тег `<link rel="import">` может быть в любом месте документа, даже в `<head>`.
- При вставке через `<iframe>` документ показывается внутри фрейма. В случае с `<link rel="import">` это не так, по умолчанию документ вообще не показывается.

HTML, загруженный через **`<link rel="import">`** имеет отдельный DOM документа, но скрипты в нём выполняются в общем контексте страницы.

Файл, загруженный через `<link rel="import">`, обрабатывается, выполняются скрипты, строится DOM документа, но не показывается, а записывается в свойство `link.import`.

Мы сами решаем, где и когда его вставить.

В примере ниже `<link rel="import" href="timer.html">` подключает документ `timer.html` и, после его загрузки, вызывает функцию `show`. Эта функция через `link.import.querySelector('time')` выбирает интересующую часть подгруженного документа и вставляет её в текущий:

```
<!DOCTYPE HTML>
<html>

<body>

  <script>
    function show() {
      var time = link.import.querySelector('time')
      document.body.appendChild(time);
    };
  </script>

  <link rel="import" id="link" onload="show()" href="timer.html">

</body>

</html>
```

В файле `timer.html` находится элемент и скрипт, который его «оживляет»:

```
<!DOCTYPE HTML>
<html>

<body>

  <time id="timer">0</time>

  <script>
    var localDocument = document.currentScript.ownerDocument;
    var timer = localDocument.getElementById('timer');

    var timerId = setInterval(function() {
      timer.innerHTML++;
    }, 1000);
  </script>

</body>

</html>
```

Важные детали:

- После загрузки все скрипты в подключённом `timer.html` выполняются в контексте основной страницы, так что `timer` и другие переменные станут глобальными переменными страницы.
- Переменная `document` – это документ основной страницы. Для доступа к импортированному, то есть текущему документу изнутри `timer.html` его можно получить как `document.currentScript.ownerDocument`.
- Таймер в загруженном документе начинает работать сразу, новый документ оживает сразу после загрузки, хотя до переноса узлов в основной документ этого может быть и не видно.

В примере выше содержимым импорта управлял основной документ, но `timer.html` мог бы и показать сам себя вызовом `document.body.appendChild(timer)` или вызвать функцию с внешнего документа, так как у них единая область видимости. Тогда не понадобился бы никакой `onload`.

Ещё пример вставки, на этот раз документ только подключает `<link>`, а таймер вставляет себя сам:

Обратим внимание – стили импорта попадают в контекст страницы. В примере выше импорт добавил и стиль для `#timer` и сам элемент.

Веб-компоненты

Импорт задуман как часть платформы Web Components.

Предполагается, что главный документ может импортировать файлы-определения, в которых будут все необходимые HTML, JS и CSS для элементов, а затем использовать их.

Пример:

```
<link rel="import" href="ui-tabs.html">
<link rel="import" href="ui-dialog.html">

<ui-tabs>...</ui-tabs>
<ui-dialog>...</ui-dialog>
```

В следующей главе мы разберём расширенный пример на эту тему.

Повторное использование

Повторный импорт с тем же URL использует уже существующий документ.

Если файл `libs.html` импортирован два раза, то CSS и скрипты из него подключатся и выполнятся ровно один раз.

Это можно использовать, чтобы не подгружать одинаковые зависимости много раз. И сама страница и её импорты, и их подимпорты, и так далее, могут подключать `libs.html` без опасения лишний раз перезагрузить и выполнить скрипты.

Например:

- Главный файл `index.html` подключает документы:

  ```
  <link rel="import" href="ui-tabs.html">
  <link rel="import" href="ui-dialog.html">
  ...
  ```

- `ui-tabs.html` подключает `libs.html`:

  ```
  <link rel="import" href="libs.html">
  ...template и код для табов...
  ```

- `ui-dialog.html` также использует `libs.html`:

  ```
  <link rel="import" href="libs.html">
  ...template и код для диалогов...
  ```

Файл `libs.html` при этом будет подключен только один раз. Это позволяет не бояться лишнего дублирования библиотек, используемых при описании множества компонент.

Итого

Тег `<link rel="import">` позволяет подключить любой документ к странице, причём:

- Скриптовое пространство и стили со страницей будут общие.
- Документ DOM – отдельный, он доступен как `link.import` снаружи, а из внутреннего скрипта – через `document.currentScript.ownerDocument`. Можно без проблем переносить элементы из главного документа в импорт и наоборот.

- Импорты могут содержать другие импорты.
- Если какой-то URL импортируется повторно – подключается уже готовый документ, без повторного выполнения скриптов в нём. Это позволяет избежать дублирования при использовании одной библиотеки во множестве мест.

Веб-компонент в сборе

В этой главе мы посмотрим на итоговый пример веб-компонента, включающий в себя описанные ранее технологии: Custom Elements, Shadow DOM, CSS Scoping и, конечно же, Imports.

Компонент ui-message

Компонент `ui-message` будет описан в отдельном файле `ui-message.html`.

Его использование будет выглядеть следующим образом:

```
<link rel="import" id="link" href="ui-message.html">

<style>
  ui-message {
    width: 80%;
    margin: auto;
  }
</style>

<ui-message class="info">Доброе утро, страна!</ui-message>

<ui-message class="warning">Внимание-внимание! Говорит информбюро!</ui-message>
```

Этот код ничем не отличается от использования обычного элемента, поэтому перейдём дальше, к содержимому `ui-message.html`

Шаблон для ui-message

Файл `ui-message.html` можно начать с шаблона:

```
<template id="tmpl">
  <style>
    .content {
      min-height: 20px;
      padding: 19px;
      margin-bottom: 20px;
      background-color: #f5f5f5;
      border: 1px solid #e3e3e3;
      border-radius: 4px;
      box-shadow: inset 0 1px 1px rgba(0, 0, 0, .05);
    }

    :host {
      display: block;
    }

    :host(.info) .content {
      color: green;
    }

    :host(.warning) .content {
      color: red;
    }
  </style>
  <div class="content">
    <content></content>
  </div>
</template>
```

Этот шаблон рисует `<div class="content">` и заполняет его содержимым элемента-хозяина.

Важные детали:

- Самое важное правило здесь `:host { display:block }`.

  Оно обязательно! . Это правило задаёт, что корень DOM-дерева будет иметь `display:block`. По умолчанию `:host` не создаёт CSS-блок, а это значит, что ни ширину ни отступы указать не получится.
- Последующие правила `:host(.info) .content` и `:host(.warning) .content` стилизуют содержимое в зависимости от того, какой на хозяине класс.

Скрипт для ui-message

В файле `ui-message.html` мы создадим новый элемент `<ui-message>`:

```
// (1) получить шаблон
var localDocument = document.currentScript.ownerDocument;
var tmpl = localDocument.getElementById('tmpl');

// (2) создать элемент
var MessageProto = Object.create(HTMLElement.prototype);

MessageProto.createdCallback = function() {
  var root = this.createShadowRoot();
  root.appendChild(tmpl.content.cloneNode(true));
```

```
};

// (3) зарегистрировать в DOM
document.registerElement('ui-message', {
  prototype: MessageProto
});
```

Все компоненты этого кода мы подробно разбирали ранее:

1. Получаем шаблон из текущего документа, то есть из самого импорта.
2. Описываем элемент. Он довольно прост – при создании записывает в свой `Shadow DOM` шаблон. При этом содержимое исходного элемента будет показано в `<content>`, но делать правила на сам `content` бессмысленно – они не сработают. Нужно либо перейти внутрь `<content>` при помощи `::content` -селектора, либо указать для внешнего элемента `.content`, что в данном случае и сделано.
3. С момента регистрации все уже существующие элементы `<ui-message>` будут превращены в описанные здесь. И будущие, конечно, тоже.

Компонент в действии:

Компонент ui-slider с jQuery

Компонент может использовать и внешние библиотеки.

Для примера создадим слайдер с использованием библиотеки jQuery UI.

Компонент `ui-slider` будет показывать слайдер с минимальным и максимальным значением из атрибутов `min/max` и генерировать событие `slide` при его перемещении.

Использование:

```
<link rel="import" id="link" href="ui-slider.html">

<ui-slider min="0" max="1000" id="elem"></ui-slider>

<script>
  elem.addEventListener("slide", function(e) {
    value.innerHTML = e.detail.value;
  });
</script>

<div id="value">0</div>
```

Файл компонента ui-slider

Файл `ui-slider.html`, задающий компонент, мы разберём по частям.

Заголовок
В начале подключим jQuery и jQuery UI.

Мы импортируем в слайдер `jquery.html`, который содержит теги `<script>` вместо того, чтобы явным образом прописывать загрузку скриптов:

```
<head>
  <link rel="import" href="jquery.html">
</head>
```

Это сделано для того, чтобы другие компоненты, которым тоже могут понадобится эти библиотеки, также могли импортировать `jquery.html`. При повторном импорте ничего не произойдёт, скрипты не будут подгружены и исполнены два раза.

То есть, это средство оптимизации.

Содержимое `jquery.html`:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
<script src="https://code.jquery.com/ui/1.11.4/jquery-ui.js"></script>
```

Шаблон
Шаблон будет помещён в Shadow DOM. В нём должны быть стили и элементы, необходимые слайдеру.

Конкретно для слайдера из разметки достаточно одного элемента `<div id="slider"></div>`, который затем будет обработан jQuery UI.

Кроме того, в шаблоне должны быть стили:

```
<template id="tmpl">
  <style>
    @import url(https://code.jquery.com/ui/1.11.4/themes/ui-lightness/jquery-ui.css);
    :host {
      display: block;
    }
  </style>
  <div id="slider"></div>
</template>
```

Скрипт

Скрипт для нового элемента похож на тот, что делали раньше, но теперь он использует jQuery UI для создания слайдера внутри своего Shadow DOM.

Для его понимания желательно знать jQuery, хотя в коде ниже я намеренно свёл использование этой библиотеки к минимуму.

```
var localDocument = document.currentScript.ownerDocument;
var tmpl = localDocument.getElementById('tmpl');

var SliderProto = Object.create(HTMLElement.prototype);

SliderProto.createdCallback = function() {
  // (1) инициализировать Shadow DOM, получить из него #slider
  var root = this.createShadowRoot();
  root.appendChild(tmpl.content.cloneNode(true));

  this.$slider = $(root.getElementById('slider'));

  var self = this;

  // (2) инициализировать слайдер, пробросить параметры
  this.$slider.slider({
    min: this.getAttribute('min') || 0,
    max: this.getAttribute('max') || 100,
    value: this.getAttribute('value') || 0,
    slide: function() {
      // (3) пробросить событие
      var event = new CustomEvent("slide", {
        detail: {
          value: self.$slider.slider("option", "value")
        },
        bubbles: true
      });
      self.dispatchEvent(event);
    }
  });
};

document.registerElement('ui-slider', {
  prototype: SliderProto
});
```

Функция `createdCallback` по шагам:

1. Создаём Shadow DOM, элемент `#slider` получаем из него, он не в основном документе.
2. Используя jQuery UI, слайдер создаётся вызовом jQuery UI методом slider, который имеет вид `$elem.slider({...параметры...});` . Параметры получаем из атрибутов `<ui-slider>` (он же `this`) и отдаём библиотеке. Она делает всю работу.
3. Параметр `slide` задаёт функцию-коллбэк, которая вызывается при передвижении слайдера и будет генерировать DOM-событие на элементе, на которое можно будет поставить обработчик при помощи `addEventListener` . В его деталях мы указываем новое значение слайдера.

Полный код с примером:

Его можно далее улучшать, например добавить геттер и сеттер для значения `value` :

```
Object.defineProperty(SliderProto, 'value', {
  get: function() {
    return this.$slider.slider("option", "value");
  },
  set: function(value) {
    this.$slider.slider('option', 'value', value);
  }
});
```

Если добавить этот код, то к значению `<ui-slider>` можно будет обращаться как `elem.value` , аналогично всяким встроенным `<input>` .

Проблема с jQuery

Попробуйте пример выше. Он не совсем работает. Слайдер прокручивается первый раз, но второй раз он как-то странно «прыгает».

Чтобы понять, почему это происходит, я заглянул в исходники jQuery UI и, после отладки происходящего, натолкнулся на проблемный код.

Он был в методе offset, который предназначен для того, чтобы определять координаты элемента. Этот метод не срабатывал, поскольку в нём есть проверка, которая выглядит примерно так:

```
var box = {
  top: 0,
  left: 0
};
...
// Make sure it's not a disconnected DOM node
if(!jQuery.contains(elem.ownerDocument, elem)) {
  return box;
}
```

То есть, jQuery проверяет, находится ли элемент `elem` внутри своего документа `elem.ownerDocument` . Если нет – то считается, что элемент вне DOM, и его размеры равны нулю.

Если копнуть чуть глубже, то `jQuery.contains` в современных браузерах сводится к обычному вызову contains.

Парадокс с Shadow DOM заключается в том, что вызов `elem.ownerDocument.contains(elem)` вернёт `false`!

Получилось, что элемент не в документе и одновременно он имеет размеры. Такого разработчики jQuery не предусмотрели.

Можно, конечно, побежать исправлять jQuery, но давайте подумаем, может быть так оно и должно быть?

С точки зрения здравого смысла, Shadow DOM является частью текущего документа. Это соответствует и духу текущей спецификации, где shadow tree рассматривается в контексте document tree.

Поэтому на самом деле `document.contains(elem)` следовало бы возвращать `true`.

Почему же `false`? Причина проста – описанный в другом стандарте механизм работы `contains` по сути состоит в проходе вверх от `elem` по цепочке `parentNode`, пока либо встретим искомый элемент, тогда ответ `true`, а иначе `false`. В случае с Shadow DOM этот путь закончится на корне Shadow DOM-дерева, оно ведь не является потомком хозяина.

Метод **`contains`** описан стандартом без учёта Shadow DOM, поэтому возвратил неверный результат **`false`**.

Это один из тех небольших, но важных нюансов, которые показывают, что стандарты всё ещё в разработке.

Итого

- С использованием современных технологий можно делать компоненты. Но это, всё же, дело будущего. Все стандарты находятся в процессе доработки, готовятся новые.
- Можно использовать произвольную библиотеку, такую как jQuery, и работать с Shadow DOM с её использованием. Но возможны проблемки. Выше была продемонстрирована одна из них, могут быть и другие.

Пока веб-компоненты ещё не являются законченными стандартами, можно попробовать Polymer – это самый известный из полифиллов на тему веб-компонент.

Он старается их эмулировать по возможности кросс-браузерно, но пока что это довольно-таки сложно, в частности, необходима дополнительная разметка.

AJAX и COMET

Современные средства для обмена данными с сервером и смежные аспекты.

Введение в AJAX и COMET

В этой главе мы «обзорно», на уровне возможностей и примеров рассмотрим технологию AJAX. Пока что с минимумом технических деталей.

Она будет полезна для понимания, что такое AJAX и с чем его едят.

Что такое AJAX?

AJAX (аббревиатура от «Asynchronous Javascript And Xml») – технология обращения к серверу без перезагрузки страницы.

За счет этого уменьшается время отклика и веб-приложение по интерактивности больше напоминает десктоп.

Несмотря на то, что в названии технологии присутствует буква `X` (от слова XML), использовать XML вовсе не обязательно. Под AJAX подразумевают любое общение с сервером без перезагрузки страницы, организованное при помощи JavaScript.

Что я могу сделать с помощью AJAX?

Элементы интерфейса

В первую очередь AJAX полезен для форм и кнопок, связанных с элементарными действиями: добавить в корзину, подписаться, и т.п.

Сейчас – в порядке вещей, что такие действия на сайтах осуществляются без перезагрузки страницы.

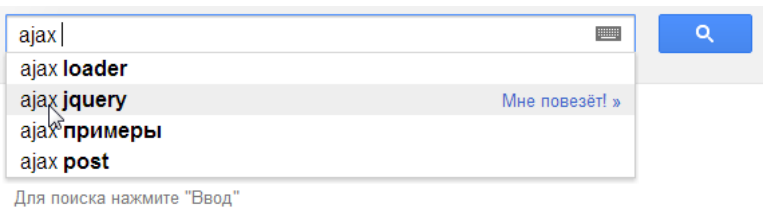
Динамическая подгрузка данных

Например, дерево, которое при раскрытии узла запрашивает данные у сервера.

Живой поиск

Живой поиск – классический пример использования AJAX, взятый на вооружение современными поисковыми системами.

Пользователь начинает печатать поисковую фразу, а JavaScript предлагает возможные варианты, получая список самых вероятных дополнений с сервера.



Код, который это обеспечивает, работает следующим образом.

1. Код активируется примерно при каждом нажатии клавиши, но не чаще чем раз в 100 мс (примерно).

2. Создается скрытый DIV и заполняется ответом сервера:
- Текущий результат подсвечен, можно перемещаться и выбирать
- При нажатии правой стрелки или при клике -- поиск в подрезультатах
3. Результаты запросов кешируются, повторных обращений к серверу не происходит.
4. В Google не только предлагаются варианты, но система тут же инициирует и сам поиск, т.е. не нужно даже нажимать `key:Enter`.
Технически, с помощью AJAX можно обмениваться любыми данными с сервером.

Обычно используются форматы:

- JSON – для отправки и получения структурированных данных, объектов.
- XML – если сервер почему-то работает в формате XML, то можно использовать и его, есть средства.
- HTML/текст – можно и просто загрузить с сервера код HTML или текст для показа на странице.
- Бинарные данные, файлы – гораздо реже, в современных браузерах есть удобные средства для них.

Что такое COMET?

COMET – общий термин, описывающий различные техники получения данных по инициативе сервера.

Можно сказать, что AJAX – это «отправил запрос – получил результат», а COMET – это «непрерывный канал, по которому приходят данные».

Примеры COMET-приложений:

- Чат – человек сидит и смотрит, что пишут другие. При этом новые сообщения приходят «сами по себе», он не должен нажимать на кнопку для обновления окна чата.
- Аукцион – человек смотрит на экран и видит, как обновляется текущая ставка за товар.
- Интерфейс редактирования – когда один редактор начинает изменять документ, другие видят информацию об этом. Возможно и совместное редактирование, когда редакторы видят изменения друг друга.

На текущий момент технология COMET удобно реализуется во всех браузерах.

Об этом разделе

Здесь мы будем говорить об AJAX и COMET на низком уровне, на уровне веб-стандартов и их использования.

Существуют библиотеки и фреймворки, добавляющие удобства, например Socket.io, CometD и другие.

В принципе, можно начать их использовать и не зная, что внутри. Но, скорее всего, вам всё равно понадобится отлаживать ошибки, смотреть детали коммуникации, выбирать наилучшее решение для конкретной задачи, и здесь обязательно разбираться, как это всё работает.

Node.JS для решения задач

В этом разделе предлагаются задачи по теме AJAX.

Конечно же, они требуют взаимодействия с сервером. Мы будем использовать серверную часть, написанную на JavaScript, на Node.JS.

Если вы не использовали Node.JS ранее – не беспокойтесь. Здесь нашей целью является преимущественно клиентская часть, поэтому прямо сейчас изучать Node.JS не обязательно. Серверные скрипты уже готовы. Нужно только поставить Node.JS и модули, чтобы их запускать.

Установка

Для настройки окружения будет достаточно сделать два шага:

1. Сначала установите сам сервер Node.JS.

   Если у вас Unix-система – рекомендуется собрать последнюю версию из исходников, а также NPM. Вы справитесь.

   Если Windows – посетите сайт http://nodejs.org или скачайте установщик (32 или 64-битный) с расширением `.msi` из http://nodejs.org/dist/latest/.

2. Выберите директорию, в которой будете решать задачи. Запустите в ней:

   ```
   npm install node-static
   ```

   Это установит в текущую директорию модуль node-static, который станет автоматически доступным для скриптов из поддиректорий.

   Если у вас Windows и команда не сработала, то скорее всего дело в том, что «не подхватились» новые пути. Перезапустите ваш файловый менеджер или консоль.

Проверка

Проверьте инсталяцию.

Для этого:

1. Создайте какую-нибудь поддиректорию и в ней файл `server.js` с таким содержимым:

```
var http = require('http');
var static = require('node-static');
var file = new static.Server('.');

http.createServer(function(req, res) {
  file.serve(req, res);
}).listen(8080);

console.log('Server running on port 8080');
```

2. Запустите его: `node server.js` .

   Должно вывести:

   ```
   Server running on port 8080
   ```

> ⚠ Нельзя запустить больше одного сервера одновременно!
>
> При попытке запуска двух серверов (например, в разных консолях) – будет конфликт портов и ошибка.

3. Откройте в браузере http://127.0.0.1:8080/server.js .

   Должно вывести код файла `server.js` .

Если всё работает – отлично, теперь вы готовы решать задачи.

Примеры

В примерах, за редким исключением, для краткости будет приводиться не полный скрипт на Node.JS, а только код обработки запроса.

Например, вместо:

```
var http = require('http');
var url = require('url');
var querystring = require('querystring');

function accept(req, res) {

  res.writeHead(200, {
    'Content-Type': 'text/plain',
    'Cache-Control': 'no-cache'
  });

  res.end("OK");
}

http.createServer(accept).listen(8080);
```

...Будет только функция `accept` , или даже только её содержимое:

```
res.writeHead(200, {
  'Content-Type': 'text/plain',
  'Cache-Control': 'no-cache'
});
```

Основные методы

В функции `accept` используются два объекта:

- `req` – объект запроса («request»), то есть то, что прислал клиент (обычно браузер), из него читаем данные.
- `res` – объект ответа («response»), в него пишем данные в ответ клиенту.
  - вызов `res.writeHead(HTTP-код, [строка статуса], {заголовки})` пишет заголовки.
  - вызов `res.write(txt)` пишет текст в ответ.
  - вызов `res.end(txt)` – завершает запрос ответом.

Демо

Кроме просмотра кода, можно будет попробовать и скачать различные демки.

Вот пример демо, можете попробовать нажать на кнопку – она работает.

Голосовать!

Если хотите посмотреть пример поближе и поиграть с ним – скачайте архив (кнопка справа-сверху в примере выше), он будет работать и на вашем Node.JS.

Больше о Node.JS

Больше о сервере Node.JS можно узнать в скринкасте по Node.JS.

Основы XMLHttpRequest

Объект `XMLHttpRequest` (или, как его кратко называют, «XHR») дает возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы.

Несмотря на слово «XML» в названии, `XMLHttpRequest` может работать с любыми данными, а не только с XML.

Использовать его очень просто.

Пример использования

Как правило, `XMLHttpRequest` используют для загрузки данных.

Для начала посмотрим на пример использования, который загружает файл `phones.json` из текущей директории и выдаёт его содержимое:

```
// 1. Создаём новый объект XMLHttpRequest
var xhr = new XMLHttpRequest();

// 2. Конфигурируем его: GET-запрос на URL 'phones.json'
xhr.open('GET', 'phones.json', false);

// 3. Отсылаем запрос
xhr.send();

// 4. Если код ответа сервера не 200, то это ошибка
if (xhr.status != 200) {
  // обработать ошибку
  alert( xhr.status + ': ' + xhr.statusText ); // пример вывода: 404: Not Found
} else {
  // вывести результат
  alert( xhr.responseText ); // responseText -- текст ответа.
}
```

В действии:

Далее мы более подробно разберём основные методы и свойства объекта `XMLHttpRequest`, в том числе те, которые были использованы в этом коде.

Настроить: open

Синтаксис:

```
xhr.open(method, URL, async, user, password)
```

Этот метод – как правило, вызывается первым после создания объекта `XMLHttpRequest`.

Задаёт основные параметры запроса:

- `method` – HTTP-метод. Как правило, используется GET либо POST, хотя доступны и более экзотические, вроде TRACE/DELETE/PUT и т.п.
- `URL` – адрес запроса. Можно использовать не только http/https, но и другие протоколы, например `ftp://` и `file://`.

  При этом есть ограничения безопасности, называемые «Same Origin Policy»: запрос со страницы можно отправлять только на тот же `протокол://домен:порт`, с которого она пришла. В следующих главах мы рассмотрим, как их можно обойти.
- `async` – если установлено в `false`, то запрос производится синхронно, если `true` – асинхронно.

«Синхронный запрос» означает, что после вызова `xhr.send()` и до ответа сервера главный поток будет «заморожен»: посетитель не сможет взаимодействовать со страницей – прокручивать, нажимать на кнопки и т.п. После получения ответа выполнение продолжится со следующей строки.

«Асинхронный запрос» означает, что браузер отправит запрос, а далее результат нужно будет получить через обработчики событий, которые мы рассмотрим далее.

- `user`, `password` – логин и пароль для HTTP-авторизации, если нужны.

> ⚠ Вызов **open** не открывает соединение
>
> Заметим, что вызов `open`, в противоположность своему названию (`open` – англ. «открыть») не открывает соединение. Он лишь настраивает запрос, а коммуникация инициируется методом `send`.

Отослать данные: send

Синтаксис:

```
xhr.send([body])
```

Именно этод метод открывает соединение и отправляет запрос на сервер.

В `body` находится *тело* запроса. Не у всякого запроса есть тело, например у GET-запросов тела нет, а у POST – основные данные как раз передаются через `body`.

Отмена: abort

Вызов `xhr.abort()` прерывает выполнение запроса.

Ответ: status, statusText, responseText

Основные свойства, содержащие ответ сервера:

`status`

HTTP-код ответа: `200`, `404`, `403` и так далее. Может быть также равен `0`, если сервер не ответил или при запросе на другой домен.

`statusText`

Текстовое описание статуса от сервера: `OK`, `Not Found`, `Forbidden` и так далее.

`responseText`

Текст ответа сервера.

Есть и ещё одно свойство, которое используется гораздо реже:

`responseXML`

Если сервер вернул XML, снабдив его правильным заголовком `Content-type: text/xml`, то браузер создаст из него XML-документ. По нему можно будет делать запросы `xhr.responseXml.querySelector("...")` и другие.

Оно используется редко, так как обычно используют не XML, а JSON. То есть, сервер возвращает JSON в виде текста, который браузер превращает в объект вызовом `JSON.parse(xhr.responseText)`.

Синхронные и асинхронные запросы

Если в методе `open` установить параметр `async` равным `false`, то запрос будет синхронным.

Синхронные вызовы используются чрезвычайно редко, так как блокируют взаимодействие со страницей до окончания загрузки. Посетитель не может даже прокручивать её. Никакой JavaScript не может быть выполнен, пока синхронный вызов не завершён – в общем, в точности те же ограничения как `alert`.

```
// Синхронный запрос
xhr.open('GET', 'phones.json', false);

// Отсылаем его
xhr.send();
// ...весь JavaScript "подвиснет", пока запрос не завершится
```

Если синхронный вызов занял слишком много времени, то браузер предложит закрыть «зависшую» страницу.

Из-за такой блокировки получается, что нельзя отослать два запроса одновременно. Кроме того, забегая вперёд, заметим, что ряд продвинутых возможностей, таких как возможность делать запросы на другой домен и указывать таймаут, в синхронном режиме не работают.

Из всего вышесказанного уже должно быть понятно, что синхронные запросы используются чрезвычайно редко, а асинхронные – почти всегда.

Для того, чтобы запрос стал асинхронным, укажем параметр `async` равным `true`.

Изменённый JS-код:

```
var xhr = new XMLHttpRequest();

xhr.open('GET', 'phones.json', true);

xhr.send(); // (1)

xhr.onreadystatechange = function() { // (3)
  if (xhr.readyState != 4) return;

  button.innerHTML = 'Готово!';

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }

}

button.innerHTML = 'Загружаю...'; // (2)
button.disabled = true;
```

Если в `open` указан третий аргумент `true` (или если третьего аргумента нет), то запрос выполняется асинхронно. Это означает, что после вызова `xhr.send()` в строке `(1)` код не «зависает», а преспокойно продолжает выполняться, выполняется строка `(2)`, а результат приходит через событие `(3)`, мы изучим его чуть позже.

Полный пример в действии:

Событие readystatechange

Событие `readystatechange` происходит несколько раз в процессе отсылки и получения ответа. При этом можно посмотреть «текущее состояние запроса» в свойстве `xhr.readyState`.

В примере выше мы использовали только состояние `4` (запрос завершён), но есть и другие.

Все состояния, по спецификации:

```
const unsigned short UNSENT = 0; // начальное состояние
const unsigned short OPENED = 1; // вызван open
const unsigned short HEADERS_RECEIVED = 2; // получены заголовки
const unsigned short LOADING = 3; // загружается тело (получен очередной пакет данных)
const unsigned short DONE = 4; // запрос завершён
```

Запрос проходит их в порядке `0` → `1` → `2` → `3` → ... → `3` → `4`, состояние `3` повторяется при каждом получении очередного пакета данных по сети.

Пример ниже демонстрирует переключение между состояниями. В нём сервер отвечает на запрос `digits`, пересылая по строке из 1000 цифр раз в секунду.

Точка разрыва пакетов не гарантирована

При состоянии `readyState=3` (получен очередной пакет) мы можем посмотреть текущие данные в `responseText` и, казалось бы, могли бы работать с этими данными как с «ответом на текущий момент».

Однако, технически мы не управляем разрывами между сетевыми пакетами. Если протестировать пример выше в локальной сети, то в большинстве браузеров разрывы будут каждые 1000 символов, но в реальности пакет может прерваться на любом байте.

Чем это опасно? Хотя бы тем, что символы русского языка в кодировке UTF-8 кодируются двумя байтами каждый – и разрыв может возникнуть *между ними*.

Получится, что при очередном `readyState` в конце `responseText` будет байт-полсимвола, то есть он не будет корректной строкой – частью ответа! Если в скрипте как-то по-особому это не обработать, то неизбежны проблемы.

HTTP-заголовки

`XMLHttpRequest` умеет как указывать свои заголовки в запросе, так и читать присланные в ответ.

Для работы с HTTP-заголовками есть 3 метода:

`setRequestHeader(name, value)`

Устанавливает заголовок `name` запроса со значением `value`.

Например:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Ограничения на заголовки

Нельзя установить заголовки, которые контролирует браузер, например `Referer` или `Host` и ряд других (полный список тут).

Это ограничение существует в целях безопасности и для контроля корректности запроса.

Поставленный заголовок нельзя снять

Особенностью `XMLHttpRequest` является то, что отменить `setRequestHeader` невозможно.

Повторные вызовы лишь добавляют информацию к заголовку, например:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// в результате будет заголовок:
// X-Auth: 123, 456
```

`getResponseHeader(name)`

Возвращает значение заголовка ответа `name`, кроме `Set-Cookie` и `Set-Cookie2`.

Например:

```
xhr.getResponseHeader('Content-Type')
```

`getAllResponseHeaders()`

Возвращает все заголовки ответа, кроме `Set-Cookie` и `Set-Cookie2`.

Заголовки возвращаются в виде единой строки, например:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

Между заголовками стоит перевод строки в два символа `"\r\n"` (не зависит от ОС), значение заголовка отделено двоеточием с пробелом `": "`. Этот формат задан стандартом.

Таким образом, если хочется получить объект с парами заголовок-значение, то эту строку необходимо разбить и обработать.

Таймаут

Максимальную продолжительность асинхронного запроса можно задать свойством `timeout`:

```
xhr.timeout = 30000; // 30 секунд (в миллисекундах)
```

При превышении этого времени запрос будет оборван и сгенерировано событие `ontimeout`:

```
xhr.ontimeout = function() {
  alert( 'Извините, запрос превысил максимальное время' );
}
```

Полный список событий

Современная спецификация предусматривает следующие события по ходу обработки запроса:

- `loadstart` – запрос начат.
- `progress` – браузер получил очередной пакет данных, можно прочитать текущие полученные данные в `responseText`.
- `abort` – запрос был отменён вызовом `xhr.abort()`.
- `error` – произошла ошибка.
- `load` – запрос был успешно (без ошибок) завершён.
- `timeout` – запрос был прекращён по таймауту.
- `loadend` – запрос был завершён (успешно или неуспешно)

Используя эти события можно более удобно отслеживать загрузку (`onload`) и ошибку (`onerror`), а также количество загруженных данных (`onprogress`).

Ранее мы видели ещё одно событие – `readystatechange`. Оно появилось гораздо раньше, ещё до появления текущего стандарта.

В современных браузерах от него можно отказаться в пользу других, необходимо лишь, как мы увидим далее, учесть особенности IE8-9.

IE8,9: XDomainRequest

В IE8 и IE9 поддержка `XMLHttpRequest` ограничена:

- Не поддерживаются события, кроме `onreadystatechange`.
- Некорректно поддерживается состояние `readyState = 3`: браузер может сгенерировать его только один раз во время запроса, а не при каждом пакете данных. Кроме того, он не даёт доступ к ответу `responseText` до того, как он будет до конца получен.

Дело в том, что, когда создавались эти браузеры, спецификации были не до конца проработаны. Поэтому разработчики браузера решили добавить свой объект `XDomainRequest`, который реализовывал часть возможностей современного стандарта.

А обычный `XMLHttpRequest` решили не трогать, чтобы ненароком не сломать существующий код.

Мы подробнее поговорим про `XDomainRequest` в главе XMLHttpRequest: кросс-доменные запросы. Пока лишь заметим, что для того, чтобы получить некоторые из современных возможностей в IE8,9 – вместо `new XMLHttpRequest()` нужно использовать `new XDomainRequest`.

Кросс-браузерно:

```
var XHR = ("onload" in new XMLHttpRequest()) ? XMLHttpRequest : XDomainRequest;
var xhr = new XHR();
```

Теперь в IE8,9 поддерживаются события `onload`, `onerror` и `onprogress`. Это именно для IE8,9. Для IE10 обычный `XMLHttpRequest` уже является полноценным.

IE9- и кеширование

Обычно ответы на запросы `XMLHttpRequest` кешируются, как и обычные страницы.

Но IE9- по умолчанию кеширует все ответы, не снабжённые антикеш-заголовком. Другие браузеры этого не делают. Чтобы этого избежать, сервер должен добавить в ответ соответствующие антикеш-заголовки, например `Cache-Control: no-cache`.

Впрочем, использовать заголовки типа `Expires`, `Last-Modified` и `Cache-Control` рекомендуется в любом случае, чтобы дать понять браузеру (не обязательно IE), что ему следует делать.

Альтернативный вариант – добавить в URL запроса случайный параметр, предотвращающий кеширование.

Например, вместо `xhr.open('GET', 'service', false)` написать:

```
xhr.open('GET', 'service?r=' + Math.random(), false);
```

По историческим причинам такой способ предотвращения кеширования можно увидеть много где, так как старые браузеры плохо обрабатывали кеширующие заголовки. Сейчас серверные заголовки поддерживаются хорошо.

Итого

Типовой код для GET-запроса при помощи `XMLHttpRequest`:

```
var xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url', true);

xhr.send();

xhr.onreadystatechange = function() {
  if (this.readyState != 4) return;

  // по окончании запроса доступны:
  // status, statusText
  // responseText, responseXML (при content-type: text/xml)

  if (this.status != 200) {
    // обработать ошибку
    alert( 'ошибка: ' + (this.status ? this.statusText : 'запрос не удался') );
    return;
  }

  // получить результат из this.responseText или this.responseXML
}
```

Мы разобрали следующие методы `XMLHttpRequest`:

- `open(method, url, async, user, password)`
- `send(body)`
- `abort()`
- `setRequestHeader(name, value)`
- `getResponseHeader(name)`
- `getAllResponseHeaders()`

Свойства `XMLHttpRequest`:

- `timeout`
- `responseText`
- `responseXML`
- `status`
- `statusText`

События:

- `onreadystatechange`
- `ontimeout`
- `onerror`
- `onload`
- `onprogress`
- `onabort`
- `onloadstart`
- `onloadend`

Задачи

Выведите телефоны

важность: 5

Создайте код, который загрузит файл `phones.json` из текущей директории и выведет все названия телефонов из него в виде списка.

Демо результата:

Загрузить phones.json!

Исходный код просто выводит содержимое файла (скачайте к себе):

К решению

XMLHttpRequest POST, формы и кодировка

Во время обычной отправки формы `<form>` браузер собирает значения её полей, делает из них строку и составляет тело GET/POST-запроса для посылки на сервер.

При отправке данных через `XMLHttpRequest`, это нужно делать самим, в JS-коде. Большинство проблем и вопросов здесь связано с непониманием, где и какое кодирование нужно осуществлять.

Кодировка urlencoded

Основной способ кодировки запросов – это *urlencoded*, то есть – стандартное кодирование URL.

Например, форма:

```
<form action="/submit" method="GET">
  <input name="name" value="Ivan">
  <input name="surname" value="Ivanov">
</form>
```

Здесь есть два поля: `name=Ivan` и `surname=Ivanov`.

Браузер перечисляет такие пары «имя=значение» через символ амперсанда `&` и, так как метод GET, итоговый запрос выглядит как `/submit?name=Ivan&surname=Ivanov`.

Все символы, кроме английских букв, цифр и `- _ . ! ~ * ' ( )` заменяются на их цифровой код в UTF-8 со знаком %.

Например, пробел заменяется на `%20`, символ `/` на `%2F`, русские буквы кодируются двумя байтами в UTF-8, поэтому, к примеру, `Ц` заменится на `%D0%A6`.

Например, форма:

```
<form action="/submit" method="GET">
  <input name="name" value="Виктор">
  <input name="surname" value="Цой">
</form>
```

Будет отправлена так: `/submit?name=%D0%92%D0%B8%D0%BA%D1%82%D0%BE%D1%80&surname=%D0%A6%D0%BE%D0%B9`.

в JavaScript есть функция encodeURIComponent для получения такой кодировки «вручную»:

```
alert( encodeURIComponent(' ') ); // %20
alert( encodeURIComponent('/') ); // %2F
alert( encodeURIComponent('В') ); // %D0%92
alert( encodeURIComponent('Виктор') ); // %D0%92%D0%B8%D0%BA%D1%82%D0%BE%D1%80
```

Эта кодировка используется в основном для метода GET, то есть для передачи параметра в строке запроса. По стандарту строка запроса не может содержать произвольные Unicode-символы, поэтому они кодируются как показано выше.

GET-запрос

Формируя XMLHttpRequest, мы должны формировать запрос «руками», кодируя поля функцией `encodeURIComponent`.

Например, для посылки GET-запроса с параметрами `name` и `surname`, аналогично форме выше, их необходимо закодировать так:

```
// Передаём name и surname в параметрах запроса
```

```
var xhr = new XMLHttpRequest();

var params = 'name=' + encodeURIComponent(name) +
  '&surname=' + encodeURIComponent(surname);

xhr.open("GET", '/submit?' + params, true);

xhr.onreadystatechange = ...;

xhr.send();
```

Прочие заголовки

Браузер автоматически добавит к запросу важнейшие HTTP-заголовки, такие как `Content-Length` и `Connection`.

По спецификации браузер запрещает их явную установку, как и некоторых других низкоуровневых HTTP-заголовков, которые могли бы ввести в заблуждение сервер относительно того, кто и сколько данных ему прислал, например `Referer`. Это сделано в целях контроля правильности запроса и для безопасности.

Сообщаем про AJAX

Запрос, отправленный кодом выше через `XMLHttpRequest`, никак не отличается от обычной отправки формы. Сервер не в состоянии их отличить.

Поэтому в некоторых фреймворках, чтобы сказать серверу, что это AJAX, добавляют специальный заголовок, например такой:

```
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
```

POST с urlencoded

В методе POST параметры передаются не в URL, а в теле запроса. Оно указывается в вызове `send(body)`.

В стандартных HTTP-формах для метода POST доступны три кодировки, задаваемые через атрибут `enctype`:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text-plain`

В зависимости от `enctype` браузер кодирует данные соответствующим способом перед отправкой на сервер.

В случае с `XMLHttpRequest` мы, вообще говоря, не обязаны использовать ни один из этих способов. Главное, чтобы сервер наш запрос понял. Но обычно проще всего выбрать какой-то из стандартных.

В частности, при POST обязателен заголовок `Content-Type`, содержащий кодировку. Это указание для сервера – как обрабатывать (раскодировать) пришедший запрос.

Для примера отправим запрос в кодировке `application/x-www-form-urlencoded`:

```
var xhr = new XMLHttpRequest();

var body = 'name=' + encodeURIComponent(name) +
  '&surname=' + encodeURIComponent(surname);

xhr.open("POST", '/submit', true)
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')

xhr.onreadystatechange = ...;

xhr.send(body);
```

Только UTF-8

Всегда используется только кодировка UTF-8, независимо от языка и кодировки страницы.

Если сервер вдруг ожидает данные в другой кодировке, к примеру windows-1251, то их нужно будет перекодировать.

Кодировка multipart/form-data

Кодировка urlencoded за счёт замены символов на `%код` может сильно «раздуть» общий объём пересылаемых данных. Поэтому для пересылки файлов используется другая кодировка: multipart/form-data.

В этой кодировке поля пересылаются одно за другим, через строку-разделитель.

Чтобы использовать этот способ, нужно указать его в атрибуте `enctype` и метод должен быть POST:

```
<form action="/submit" method="POST" enctype="multipart/form-data">
  <input name="name" value="Виктор">
  <input name="surname" value="Цой">
</form>
```

Форма при такой кодировке будет выглядеть примерно так:

```
...Заголовки...
Content-Type: multipart/form-data; boundary=RaNdOmDeLiMiTeR

--RaNdOmDeLiMiTeR
Content-Disposition: form-data; name="name"

Виктор
--RaNdOmDeLiMiTeR
Content-Disposition: form-data; name="surname"

Цой
--RaNdOmDeLiMiTeR--
```

...То есть, поля передаются одно за другим, значения не кодируются, а чтобы было чётко понятно, какое значение где – поля разделены случайно сгенерированной строкой, которую называют «boundary» (англ. граница), в примере выше это `RaNdOmDeLiMiTeR` :

Сервер видит заголовок `Content-Type: multipart/form-data` , читает из него границу и раскодирует поля формы.

Такой способ используется в первую очередь при пересылке файлов, так перекодировка мегабайтов через urlencoded существенно загрузила бы браузер. Да и объём данных после неё сильно вырос бы.

Однако, никто не мешает использовать эту кодировку всегда для POST запросов. Для GET доступна только urlencoded.

POST с multipart/form-data

Сделать POST-запрос в кодировке `multipart/form-data` можно и через XMLHttpRequest.

Достаточно указать в заголовке `Content-Type` кодировку и границу, и далее сформировать тело запроса, удовлетворяющее требованиям кодировки.

Пример кода для того же запроса, что и раньше, теперь в кодировке `multipart/form-data` :

```
var data = {
  name: 'Виктор',
  surname: 'Цой'
};

var boundary = String(Math.random()).slice(2);
var boundaryMiddle = '--' + boundary + '\r\n';
var boundaryLast = '--' + boundary + '--\r\n'

var body = ['\r\n'];
for (var key in data) {
  // добавление поля
  body.push('Content-Disposition: form-data; name="' + key + '"\r\n\r\n' + data[key] + '\r\n');
}

body = body.join(boundaryMiddle) + boundaryLast;

// Тело запроса готово, отправляем

var xhr = new XMLHttpRequest();
xhr.open('POST', '/submit', true);

xhr.setRequestHeader('Content-Type', 'multipart/form-data; boundary=' + boundary);

xhr.onreadystatechange = function() {
  if (this.readyState != 4) return;

  alert( this.responseText );
}

xhr.send(body);
```

Тело запроса будет иметь вид, описанный выше, то есть поля через разделитель.

Отправка файла

Можно создать запрос, который сервер воспримет как загрузку файла.

Для добавления файла нужно использовать тот же код, что выше, модифицировав заголовки перед полем, которое является файлом, так:

```
Content-Disposition: form-data; name="myfile"; filename="pic.jpg"
Content-Type: image/jpeg
(пустая строка)
содержимое файла
```

FormData

Современные браузеры, исключая IE9- (впрочем, есть полифилл), поддерживают встроенный объект FormData, который кодирует формы для отправки на сервер.

Это очень удобно. Например:

```
<form name="person">
  <input name="name" value="Виктор">
  <input name="surname" value="Цой">
</form>
```

```
<script>
  // создать объект для формы
  var formData = new FormData(document.forms.person);

  // добавить к пересылке ещё пару ключ - значение
  formData.append("patronym", "Робертович");

  // отослать
  var xhr = new XMLHttpRequest();
  xhr.open("POST", "/url");
  xhr.send(formData);
</script>
```

Этот код отправит на сервер форму с полями `name`, `surname` и `patronym`.

Интерфейс:

- Конструктор `new FormData([form])` вызывается либо без аргументов, либо с DOM-элементом формы.
- Метод `formData.append(name, value)` добавляет данные к форме.

Объект `formData` можно сразу отсылать, интеграция `FormData` с `XMLHttpRequest` встроена в браузер. Кодировка при этом будет `multipart/form-data`.

Другие кодировки

XMLHttpRequest сам по себе не ограничивает кодировку и формат пересылаемых данных.

Поэтому для обмена данными часто используется формат JSON:

```
var xhr = new XMLHttpRequest();

var json = JSON.stringify({
  name: "Виктор",
  surname: "Цой"
});

xhr.open("POST", '/submit', true)
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.onreadystatechange = ...;

// Отсылаем объект в формате JSON и с Content-Type application/json
// Сервер должен уметь такой Content-Type принимать и раскодировать
xhr.send(json);
```

Итого

- У форм есть две основные кодировки: `application/x-www-form-urlencoded` – по умолчанию и `multipart/form-data` – для POST запросов, если явно указана в `enctype`. Вторая кодировка обычно используется для больших данных и только для тела запроса.
- Для составления запроса в `application/x-www-form-urlencoded` используется функция `encodeURIComponent`.
- Для отправки запроса в `multipart/form-data` – объект `FormData`.
- Для обмена данными JS ↔ сервер можно использовать и просто JSON, желательно с указанием кодировки в заголовке `Content-Type`.

В XMLHttpRequest можно использовать и другие HTTP-методы, например PUT, DELETE, TRACE. К ним применимы все те же принципы, что описаны выше.

XMLHttpRequest: кросс-доменные запросы

Обычно запрос `XMLHttpRequest` может делать запрос только в рамках текущего сайта. При попытке использовать другой домен/порт/протокол – браузер выдаёт ошибку.

Существует современный стандарт XMLHttpRequest, он ещё в состоянии черновика, но предусматривает кросс-доменные запросы и многое другое.

Большинство возможностей этого стандарта уже поддерживаются всеми браузерами, но увы, не в IE9-.

Впрочем, частично кросс-доменные запросы поддерживаются, начиная с IE8, только вместо `XMLHttpRequest` нужно использовать объект XDomainRequest.

Кросс-доменные запросы

Разберём кросс-доменные запросы на примере кода:

```
// (1)
var XHR = ("onload" in new XMLHttpRequest()) ? XMLHttpRequest : XDomainRequest;

var xhr = new XHR();

// (2) запрос на другой домен :)
xhr.open('GET', 'http://anywhere.com/request', true);

xhr.onload = function() {
  alert( this.responseText );
}

xhr.onerror = function() {
```

```
  alert( 'Ошибка ' + this.status );
}

xhr.send();
```

1. Мы создаём `XMLHttpRequest` и проверяем, поддерживает ли он событие `onload`. Если нет, то это старый `XMLHttpRequest`, значит это IE8,9, и используем `XDomainRequest`.
2. Запрос на другой домен отсылается просто указанием соответствующего URL в `open`. Он обязательно должен быть асинхронным, в остальном – никаких особенностей.

Контроль безопасности

Кросс-доменные запросы проходят специальный контроль безопасности, цель которого – не дать злым хакерам™ завоевать интернет.

Серьёзно. Разработчики стандарта предусмотрели все заслоны, чтобы «злой хакер» не смог, воспользовавшись новым стандартом, сделать что-то принципиально отличное от того, что и так мог раньше и, таким образом, «сломать» какой-нибудь сервер, работающий по-старому стандарту и не ожидающий ничего принципиально нового.

Давайте, на минуточку, вообразим, что появился стандарт, который даёт, без ограничений, возможность делать любой странице HTTP-запросы куда угодно, какие угодно.

Как сможет этим воспользоваться злой хакер?

Он сделает свой сайт, например `http://evilhacker.com` и заманит туда посетителя (а может посетитель попадёт на «злонамеренную» страницу и по ошибке – не так важно).

Когда посетитель зайдёт на `http://evilhacker.com`, он автоматически запустит JS-скрипт на странице. Этот скрипт сделает HTTP-запрос на почтовый сервер, к примеру, `http://gmail.com`. А ведь обычно HTTP-запросы идут с куками посетителя и другими авторизующими заголовками.

Поэтому хакер сможет написать на `http://evilhacker.com` код, который, сделав GET-запрос на `http://gmail.com`, получит информацию из почтового ящика посетителя. Проанализирует её, сделает ещё пачку POST-запросов для отправки писем от имени посетителя. Затем настанет очередь онлайн-банка и так далее.

Спецификация CORS налагает специальные ограничения на запросы, которые призваны не допустить подобного апокалипсиса.

Запросы в ней делятся на два вида.

Простыми считаются запросы, если они удовлетворяют следующим двум условиям:

1. Простой метод: GET, POST или HEAD
2. Простые заголовки – только из списка:

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Content-Type` со значением `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`.

«Непростыми» считаются все остальные, например, запрос с методом `PUT` или с заголовком `Authorization` не подходит под ограничения выше.

Принципиальная разница между ними заключается в том, что «простой» запрос можно сформировать и отправить на сервер и без XMLHttpRequest, например при помощи HTML-формы.

То есть, злой хакер на странице `http://evilhacker.com` и до появления CORS мог отправить произвольный GET-запрос куда угодно. Например, если создать и добавить в документ элемент `<script src="любой url">`, то браузер сделает GET-запрос на этот URL.

Аналогично, злой хакер и ранее мог на своей странице объявить и, при помощи JavaScript, отправить HTML-форму с методом GET/POST и кодировкой `multipart/form-data`. А значит, даже старый сервер наверняка предусматривает возможность таких атак и умеет от них защищаться.

А вот запросы с нестандартными заголовками или с методом `DELETE` таким образом не создать. Поэтому старый сервер может быть к ним не готов. Или, к примеру, он может полагать, что такие запросы веб-страница в принципе не умеет присылать, значит они пришли из привилегированного приложения, и дать им слишком много прав.

Поэтому при посылке «непростых» запросов нужно специальным образом спросить у сервера, согласен ли он в принципе на подобные кросс-доменные запросы или нет? И, если сервер не ответит, что согласен – значит, нет.

> В спецификации CORS, как мы увидим далее, есть много деталей, но все они объединены единым принципом: новые возможности доступны только с явного согласия сервера (по умолчанию – нет).

CORS для простых запросов

В кросс-доменный запрос браузер автоматически добавляет заголовок `Origin`, содержащий домен, с которого осуществлён запрос.

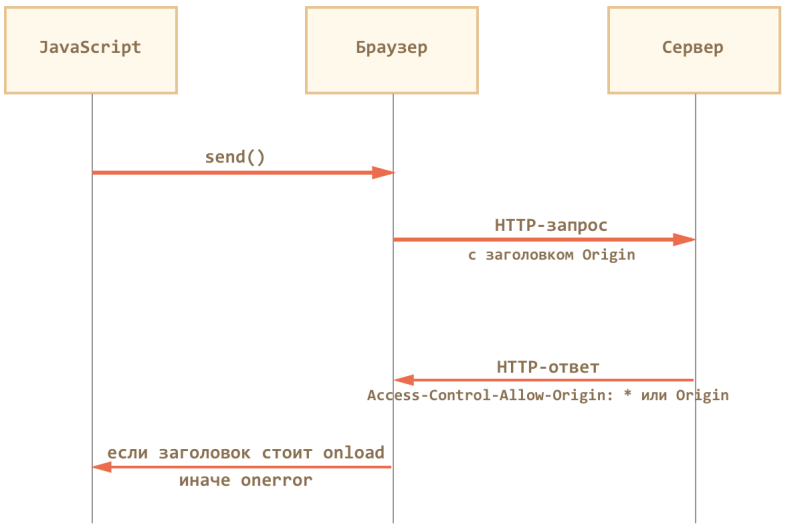
В случае запроса на `http://anywhere.com/request` с `http://javascript.ru/page` заголовки будут примерно такие:

```
GET /request
Host:anywhere.com
Origin:http://javascript.ru
...
```

Сервер должен, со своей стороны, ответить специальными заголовками, разрешает ли он такой запрос к себе.

Если сервер разрешает кросс-доменный запрос с этого домена – он должен добавить к ответу заголовок `Access-Control-Allow-Origin`, содержащий домен запроса (в данном случае «javascript.ru») или звёздочку `*`.

Только при наличии такого заголовка в ответе – браузер сочтёт запрос успешным, а иначе JavaScript получит ошибку.



То есть, ответ сервера может быть примерно таким:

```
HTTP/1.1 200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: http://javascript.ru
```

Если `Access-Control-Allow-Origin` нет, то браузер считает, что разрешение не получено, и завершает запрос с ошибкой.

При таких запросах не передаются куки и заголовки HTTP-авторизации. Параметры `user` и `password` в методе `open` игнорируются. Мы рассмотрим, как разрешить их передачу, чуть далее.

⚠ Что может сделать хакер, используя такие запросы?

Описанные выше ограничения приводят к тому, что запрос полностью безопасен.

Действительно, злая страница может сформировать любой GET/POST-запрос и отправить его, но без разрешения сервера ответа она не получит.

А без ответа такой запрос, по сути, эквивалентен отправке формы GET/POST, причём без авторизации.

Ограничения IE9-

В IE9- используется `XDomainRequest`, который представляет собой урезанный `XMLHttpRequest`.

На него действуют ограничения:

- Протокол нужно сохранять: запросы допустимы с HTTP на HTTP, с HTTPS на HTTPS. Другие протоколы запрещены.
- Метод `open(method, url)` имеет только два параметра. Он всегда асинхронный.
- Ряд возможностей современного стандарта недоступны, в частности:
  - Недоступны методы, кроме GET или POST.
  - Нельзя добавлять свои заголовки, даже нельзя указать свой `Content-Type` для запроса, он всегда `text/plain`.
  - Нельзя включить передачу кук и данных HTTP-авторизации.
- В IE8 в режиме просмотра InPrivate кросс-доменные запросы не работают.

Современный стандарт XMLHttpRequest предусматривает средства для преодоления этих ограничений, но на момент выхода IE8 они ещё не были проработаны, поэтому их не реализовали. А IE9 исправил некоторые ошибки, но в общем не добавил ничего нового.

Поэтому на сайтах, которые хотят поддерживать IE9-, то на практике кросс-доменные запросы редко используют, предпочитая другие способы кросс-доменной коммуникации. Например, динамически создаваемый тег `SCRIPT` или вспомогательный `IFRAME` с другого домена. Мы разберём эти подходы в последующих главах.

Как разрешить кросс-доменные запросы от доверенного сайта в IE9-?

Разрешить кросс-доменные запросы для «доверенных» сайтов можно в настройках IE, во вкладке «Безопасность», включив пункт «Доступ к источникам данных за пределами домена».

Обычно это делается для зоны «Надёжные узлы», после чего в неё вносится доверенный сайт. Теперь он может делать кросс-доменные запросы `XMLHttpRequest`.

Этот способ можно применить для корпоративных сайтов, а также в тех случаях, когда посетитель заведомо вам доверяет, но почему-то (компьютер на работе, админ запрещает ставить другой браузер?) хочет использовать именно IE. Например, он может предлагаться в качестве дополнительной инструкции «как заставить этот сервис работать под IE».

В IE разрешён другой порт

В кросс-доменные ограничения IE не включён порт.

То есть, можно сделать запрос с `http://javascript.ru` на `http://javascript.ru:8080`, и в IE он не будет считаться кросс-доменным.

Это позволяет решить некоторые задачи, связанные с взаимодействием различных сервисов в рамках одного сайта. Но только для IE.

Расширенные возможности, описанные далее, поддерживаются всеми современными браузерами, кроме IE9-.

Заголовки ответа

Чтобы JavaScript мог прочитать HTTP-заголовок ответа, сервер должен указать его имя в `Access-Control-Expose-Headers`.

Например:

```
HTTP/1.1 200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: http://javascript.ru
X-Uid: 123
X-Authorization: 2c9de507f2c54aa1
Access-Control-Expose-Headers: X-Uid, X-Authentication
```

По умолчанию скрипт может прочитать из ответа только «простые» заголовки:

```
Cache-Control
Content-Language
Content-Type
Expires
Last-Modified
Pragma
```

...То есть, `Content-Type` получить всегда можно, а доступ к специфическим заголовкам нужно открывать явно.

Запросы от имени пользователя

По умолчанию браузер не передаёт с запросом куки и авторизующие заголовки.

Чтобы браузер передал вместе с запросом куки и HTTP-авторизацию, нужно поставить запросу `xhr.withCredentials = true`:

```
var xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.open('POST', 'http://anywhere.com/request', true)
  ...
```

Далее – всё как обычно, дополнительных действий со стороны клиента не требуется.

Такой `XMLHttpRequest` с куками, естественно, требует от сервера больше разрешений, чем «анонимный».

Поэтому для запросов с **`withCredentials`** предусмотрено дополнительное подтверждение со стороны сервера.

При запросе с `withCredentials` сервер должен вернуть уже не один, а два заголовка:

- `Access-Control-Allow-Origin: домен`
- `Access-Control-Allow-Credentials: true`

Пример заголовков:

```
HTTP/1.1 200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: http://javascript.ru
Access-Control-Allow-Credentials: true
```

Использование звёздочки `*` в `Access-Control-Allow-Origin` при этом запрещено.

Если этих заголовков не будет, то браузер не даст JavaScript'у доступ к ответу сервера.

«Непростые» запросы

В кросс-доменном `XMLHttpRequest` можно указать не только `GET/POST`, но и любой другой метод, например `PUT`, `DELETE`.

Когда-то никто и не думал, что страница сможет сделать такие запросы. Поэтому ряд веб-сервисов написаны в предположении, что «если метод – нестандартный, то это не браузер». Некоторые веб-сервисы даже учитывают это при проверке прав доступа.

Чтобы пресечь любые недопонимания, браузер использует предзапрос в случаях, когда:

- Если метод – не GET / POST / HEAD.
- Если заголовок `Content-Type` имеет значение отличное от `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`, например `application/xml`.
- Если устанавливаются другие HTTP-заголовки, кроме `Accept`, `Accept-Language`, `Content-Language`.

...Любое из условий выше ведёт к тому, что браузер сделает два HTTP-запроса.

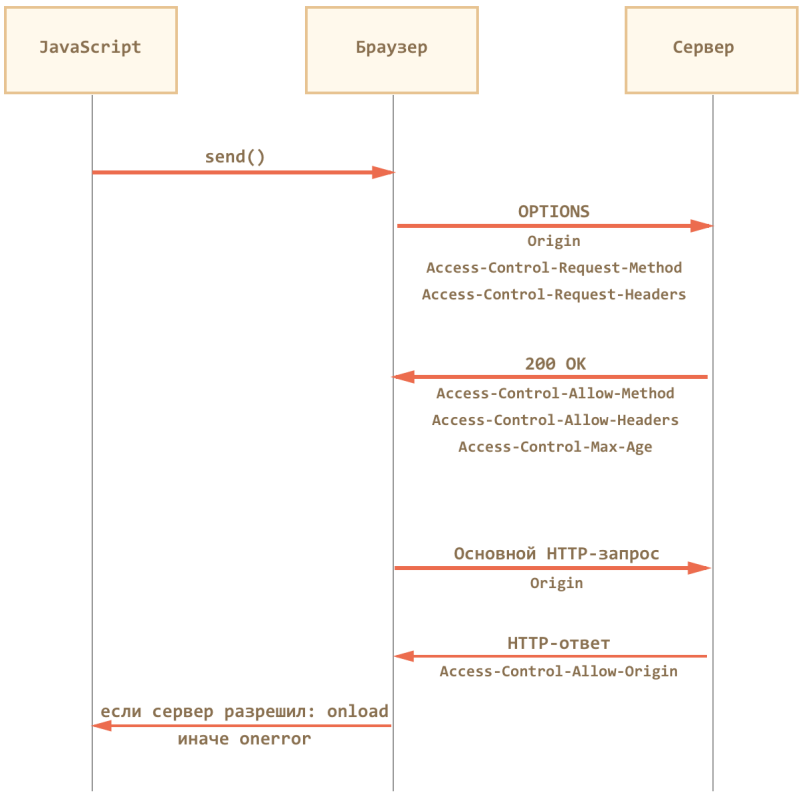
Первый запрос называется «предзапрос» (английский термин «preflight»). Браузер делает его целиком по своей инициативе, из JavaScript мы о нём ничего не знаем, хотя можем увидеть в инструментах разработчика.

Этот запрос использует метод `OPTIONS`. Он не содержит тела и содержит название желаемого метода в заголовке `Access-Control-Request-Method`, а если добавлены особые заголовки, то и их тоже – в `Access-Control-Request-Headers`.

Его задача – спросить сервер, разрешает ли он использовать выбранный метод и заголовки.

На этот запрос сервер должен ответить статусом 200, без тела ответа, указав заголовки `Access-Control-Allow-Method: метод` и, при необходимости, `Access-Control-Allow-Headers: разрешённые заголовки`.

Дополнительно он может указать `Access-Control-Max-Age: sec`, где `sec` – количество секунд, на которые нужно закэшировать разрешение. Тогда при последующих вызовах метода браузер уже не будет делать предзапрос.



Давайте рассмотрим предзапрос на конкретном примере.

Пример запроса COPY

Рассмотрим запрос `COPY`, который используется в протоколе WebDAV для управления файлами через HTTP:

```
var xhr = new XMLHttpRequest();

xhr.open('COPY', 'http://site.com/~ilya', true);
xhr.setRequestHeader('Destination', 'http://site.com/~ilya.bak');

xhr.onload = ...
xhr.onerror = ...

xhr.send();
```

Этот запрос «непростой» по двум причинам (достаточно было бы одной из них):

1. Метод `COPY`.

2. Заголовок `Destination`.

Поэтому браузер, по своей инициативе, шлёт предварительный запрос `OPTIONS`:

```
OPTIONS /~ilya HTTP/1.1
Host: site.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: http://javascript.ru
Access-Control-Request-Method: COPY
Access-Control-Request-Headers: Destination
```

Обратим внимание на детали:

- Адрес – тот же, что и у основного запроса: `http://site.com/~ilya`.
- Стандартные заголовки запроса `Accept`, `Accept-Encoding`, `Connection` присутствуют.
- Кросс-доменные специальные заголовки запроса:
  - `Origin` – домен, с которого сделан запрос.
  - `Access-Control-Request-Method` – желаемый метод.
  - `Access-Control-Request-Headers` – желаемый «непростой» заголовок.

На этот запрос сервер должен ответить статусом 200, указав заголовки `Access-Control-Allow-Method: COPY` и `Access-Control-Allow-Headers: Destination`.

Но в протоколе WebDav разрешены многие методы и заголовки, которые имеет смысл сразу перечислить в ответе:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Access-Control-Allow-Methods: PROPFIND, PROPPATCH, COPY, MOVE, DELETE, MKCOL, LOCK, UNLOCK, PUT, GETLIB, VERSION-CONTROL, CHECKIN, CHECKOUT, UNCHECKOUT, REPORT, U
Access-Control-Allow-Headers: Overwrite, Destination, Content-Type, Depth, User-Agent, X-File-Size, X-Requested-With, If-Modified-Since, X-File-Name, Cache-Contro
Access-Control-Max-Age: 86400
```

Ответ должен быть без тела, то есть только заголовки.

Браузер видит, что метод `COPY` – в числе разрешённых и заголовок `Destination` – тоже, и дальше он шлёт уже основной запрос.

При этом ответ на предзапрос он закэширует на 86400 сек (сутки), так что последующие аналогичные вызовы сразу отправят основной запрос, без `OPTIONS`.

Основной запрос браузер выполняет уже в «обычном» кросс-доменном режиме:

```
COPY /~ilya HTTP/1.1
Host: site.com
Content-Type: text/html; charset=UTF-8
Destination: http://site.com/~ilya.bak
Origin: http://javascript.ru
```

Ответ сервера, согласно спецификации WebDav COPY, может быть примерно таким:

```
HTTP/1.1 207 Multi-Status
Content-Type: text/xml; charset="utf-8"
Content-Length: ...
Access-Control-Allow-Origin: http://javascript.ru

<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
  ...
</d:multistatus>
```

Так как `Access-Control-Allow-Origin` содержит правильный домен, то браузер вызовет `xhr.onload` и запрос будет завершён.

Итого

- Все современные браузеры умеют делать кросс-доменные XMLHttpRequest.
- В IE8,9 для этого используется объект `XDomainRequest`, ограниченный по возможностям.
- Кросс-доменный запрос всегда содержит заголовок `Origin` с доменом запроса.

Порядок выполнения:

1. Для запросов с «непростым» методом или особыми заголовками браузер делает предзапрос `OPTIONS`, указывая их в `Access-Control-Request-Method` и `Access-Control-Request-Headers`.

   Браузер ожидает ответ со статусом `200`, без тела, со списком разрешённых методов и заголовков в `Access-Control-Allow-Method` и `Access-Control-Allow-Headers`. Дополнительно можно указать `Access-Control-Max-Age` для кеширования предзапроса.

2. Браузер делает запрос и проверяет, есть ли в ответе `Access-Control-Allow-Origin`, равный `*` или `Origin`.

   Для запросов с `withCredentials` может быть только `Origin` и дополнительно `Access-Control-Allow-Credentials: true`.

3. Если проверки пройдены, то вызывается `xhr.onload`, иначе `xhr.onerror`, без деталей ответа.
4. Дополнительно: названия нестандартных заголовков ответа сервер должен указать в `Access-Control-Expose-Headers`, если хочет, чтобы клиент мог их прочитать.

Детали и примеры мы разобрали выше.

Задачи

Зачем нужен Origin?

важность: 5

Как вы, наверняка, знаете, существует HTTP-заголовок `Referer`, в котором обычно указан адрес страницы, с которой инициирован запрос.

Например, при отправке `XMLHttpRequest` со страницы `http://javascript.ru/some/url` на `http://google.ru`, заголовки будут примерно такими:

```
Accept:*/*
Accept-Charset:windows-1251,utf-8;q=0.7,*;q=0.3
Accept-Encoding:gzip,deflate,sdch
Accept-Language:ru-RU,ru;q=0.8,en-US;q=0.6,en;q=0.4
Connection:keep-alive
Host:google.ru
Origin:http://javascript.ru
Referer:http://javascript.ru/some/url
```

Как видно, здесь присутствуют и `Referer` и `Origin`.

Итак, вопросы:

1. Зачем нужен `Origin`, если `Referer` содержит даже более полную информацию?
2. Может ли быть такое, что заголовка `Referer` нет или он неправильный?

К решению

XMLHttpRequest: индикация прогресса

Запрос `XMLHttpRequest` состоит из двух фаз:

1. Стадия закачки (upload). На ней данные загружаются на сервер. Эта фаза может быть долгой для POST-запросов. Для отслеживания прогресса на стадии закачки существует объект типа XMLHttpRequestUpload, доступный как `xhr.upload` и события на нём.
2. Стадия скачивания (download). После того, как данные загружены, браузер скачивает ответ с сервера. Если он большой, то это может занять существенное время. На этой стадии используется обработчик `xhr.onprogress`.

Далее – обо всём по порядку.

Стадия закачки

На стадии закачки для получения информации используем объект `xhr.upload`. У этого объекта нет методов, он только генерирует события в процессе закачки. А они-то как раз и нужны.

Вот полный список событий:

- `loadstart`
- `progress`
- `abort`
- `error`
- `load`
- `timeout`
- `loadend`

Пример установки обработчиков на стадию закачки:

```
xhr.upload.onprogress = function(event) {
  alert( 'Загружено на сервер ' + event.loaded + ' байт из ' + event.total );
}

xhr.upload.onload = function() {
  alert( 'Данные полностью загружены на сервер!' );
}

xhr.upload.onerror = function() {
  alert( 'Произошла ошибка при загрузке данных на сервер!' );
}
```

Стадия скачивания

После того, как загрузка завершена, и сервер соизволит ответить на запрос, `XMLHttpRequest` начнёт скачивание ответа сервера.

На этой фазе `xhr.upload` уже не нужен, а в дело вступают обработчики событий на самом объекте `xhr`. В частности, событие `xhr.onprogress` содержит информацию о количестве принятых байт ответа.

Пример обработчика:

```
xhr.onprogress = function(event) {
  alert( 'Получено с сервера ' + event.loaded + ' байт из ' + event.total );
}
```

Все события, возникающие в этих обработчиках, имеют тип ProgressEvent, то есть имеют свойства `loaded` – количество уже пересланных данных в байтах и `total` – общее количество данных.

Демо: загрузка файла с индикатором прогресса

Современный `XMLHttpRequest` позволяет отправить на сервер всё, что угодно. Текст, файл, форму.

Мы, для примера, рассмотрим загрузку файла с индикацией прогресса. Это требует от браузера поддержки File API, то есть исключает IE9-.

File API позволяет получить доступ к содержимому файла, который перенесён в браузер при помощи Drag'n'Drop или выбран в поле формы, и отправить его при помощи `XMLHttpRequest`.

Форма для выбора файла с обработчиком `submit`:

```
<form name="upload">
  <input type="file" name="myfile">
  <input type="submit" value="Загрузить">
</form>

<script>
  document.forms.upload.onsubmit = function() {
    var input = this.elements.myfile;
    var file = input.files[0];
    if (file) {
      upload(file);
    }
    return false;
  }
</script>
```

Мы получаем файл из формы через свойство `files` элемента `<input>` и передаём его в функцию `upload`:

```
function upload(file) {

  var xhr = new XMLHttpRequest();

  // обработчик для закачки
  xhr.upload.onprogress = function(event) {
    log(event.loaded + ' / ' + event.total);
  }

  // обработчики успеха и ошибки
  // если status == 200, то это успех, иначе ошибка
  xhr.onload = xhr.onerror = function() {
    if (this.status == 200) {
      log("success");
    } else {
      log("error " + this.status);
    }
  };

  xhr.open("POST", "upload", true);
  xhr.send(file);

}
```

Этот код отправит файл на сервер и будет сообщать о прогрессе при его закачке (`xhr.upload.onprogress`), а также об окончании запроса (`xhr.onload` , `xhr.onerror`).

Полный пример индикации прогресса при загрузке, основанный на коде выше:

Событие onprogress в деталях

При обработке события `onprogress` есть ряд важных тонкостей.

Можно, конечно, их игнорировать, но лучше бы знать.

Заметим, что событие, возникающее при `onprogress` , имеет одинаковый вид на стадии закачки (в обработчике `xhr.upload.onprogress`) и при получении ответа (в обработчике `xhr.onprogress`).

Оно представляет собой объект типа ProgressEvent со свойствами:

`loaded`

Сколько байт уже переслано.

Имеется в виду только тело запроса, заголовки не учитываются.

`lengthComputable`

Если `true`, то известно полное количество байт для пересылки, и оно хранится в свойстве `total`.

`total`

Общее количество байт для пересылки, если известно.

А может ли оно быть неизвестно?

- При закачке на сервер браузер всегда знает полный размер пересылаемых данных, так что `total` всегда содержит конкретное количество байт, а значение `lengthComputable` всегда будет `true`.
- При скачивании данных – обычно сервер в начале сообщает их общее количество в HTTP-заголовке `Content-Length`. Но он может и не делать этого, например если сам не знает, сколько данных будет или если генерирует их динамически. Тогда `total` будет равно `0`. А чтобы отличить нулевой размер данных от неизвестного – как раз служит `lengthComputable`, которое в данном случае равно `false`.

Ещё особенности, которые необходимо учитывать при использовании `onprogress`:

- Событие происходит при каждом полученном/отправленном байте, но не чаще чем раз в 50 мс.

  Это обозначено в спецификации progress notifications.

- В процессе получения данных, ещё до их полной передачи, доступен **`xhr.responseText`**, но он не обязательно содержит корректную строку.

  Можно до окончания запроса заглянуть в него и прочитать текущие полученные данные. Важно, что при пересылке строки в кодировке UTF-8 кириллические символы, как, впрочем, и многие другие, кодируются 2 байтами. Возможно, что в конце одного пакета данных окажется первая половинка символа, а в начале следующего – вторая. Поэтому полагаться на то, что до окончания запроса в `responseText` находится корректная строка нельзя. Она может быть обрезана посередине символа.

  Исключение – заведомо однобайтные символы, например цифры или латиница.

- Сработавшее событие **`xhr.upload.onprogress`** не гарантирует, что данные дошли.

  Событие `xhr.upload.onprogress` срабатывает, когда данные отправлены браузером. Но оно не гарантирует, что сервер получил, обработал и записал данные на диск. Он говорит лишь о самом факте отправки.

  Поэтому прогресс-индикатор, получаемый при его помощи, носит приблизительный и оптимистичный характер.

Файлы и формы

Выше мы использовали `xhr.send(file)` для передачи файла непосредственно в теле запроса.

При этом посылается только *содержимое* файла.

Если нужно дополнительно передать имя файла или что-то ещё – это можно удобно сделать через форму, при помощи объекта FormData:

Создадим форму `formData` и прибавим к ней поле с файлом `file` и именем `"myfile"`:

```
var formData = new FormData();
formData.append("myfile", file);
xhr.send(formData);
```

Данные будут отправлены в кодировке `multipart/form-data`. Серверный фреймворк увидит это как обычную форму с файлом, практически все серверные технологии имеют их встроенную поддержку. Индикация прогресса реализуется точно так же.

XMLHttpRequest: возобновляемая закачка

Современный `XMLHttpRequest` даёт возможность загружать файл как угодно: во множество потоков, с догрузкой, с подсчётом контрольной суммы и т.п.

Здесь мы рассмотрим общий подход к организации загрузки, а его уже можно расширять, адаптировать к своему фреймворку и так далее.

Поддержка – все браузеры кроме IE9-.

Неточный upload.onprogress

Ранее мы рассматривали загрузку с индикатором прогресса. Казалось бы, сделать возобновляемую загрузку на его основе очень просто.

Есть же `xhr.upload.onprogress` – ставим на него обработчик, по свойству `loaded` события `onprogress` смотрим, сколько байт загрузилось. А при обрыве – возобновляем загрузку с последнего байта.

К счастью, отослать на сервер не весь файл, а только нужную часть его – не проблема, File API позволяет прочитать выбранный участок из файла и отправить его.

Примерно так:

```
var slice = file.slice(10, 100); // прочитать байты с 10-го по 99-й включительно

xhr.send(slice); // ... и отправить эти байты в запросе.
```

...Но такая модель не жизнеспособна!

Всё дело в том, что `upload.onprogress` срабатывает, когда байты *отправлены*, но были ли они получены сервером – браузер не знает. Может, их прокси-сервер забуферизовал, может серверный процесс «упал» в процессе обработки, может соединение порвалось и байты так и не дошли до получателя.

Поэтому **`onprogress`** годится лишь для красивенького рисования прогресса.

Для загрузки нам нужно точно знать количество загруженных байт. Это может сообщить только сервер.

Алгоритм возобновляемой загрузки

Загрузкой файла будет заведовать объект `Uploader`, его примерный общий вид:

```
function Uploader(file, onSuccess, onFail, onProgress) {

  var fileId = file.name + '-' + file.size + '-' + +file.lastModifiedDate;

  var errorCount = 0;

  var MAX_ERROR_COUNT = 6;

  function upload() {
    ...
  }

  function pause() {
    ...
  }

  this.upload = upload;
  this.pause = pause;
}
```

- Аргументы для `new Uploader`:

`file`

Объект File API. Может быть получен из формы, либо как результат Drag'n'Drop.

`onSuccess`, `onFail`, `onProgress`

Функции-коллбэки, которые будут вызываться в процессе (`onProgress`) и при окончании загрузки.

- Подробнее про важные данные, с которыми мы будем работать в процессе загрузки:

`fileId`

Уникальный идентификатор файла, генерируется по имени, размеру и дате модификации. По нему мы всегда сможем возобновить загрузку, в том числе и после закрытия и открытия браузера.

`startByte`

С какого байта загружать. Изначально – с нулевого.

`errorCount / MAX_ERROR_COUNT`

Текущее число ошибок / максимальное число ошибок подряд, после которого загрузка считается проваленной.

Алгоритм загрузки:

1. Генерируем `fileId` из названия, размера, даты модификации файла. Можно добавить и идентификатор посетителя.
2. Спрашиваем сервер, есть ли уже такой файл, и если да – сколько байт уже загружено?
3. Отсылаем файл с позиции, которую сказал сервер.

При этом загрузку можно прервать в любой момент, просто оборвав все запросы.

Демо ниже, к сожалению, работает лишь частично, так как на этом сайте Node.JS стоит за сервером Nginx, который буферизует все закачки, не передавая их в Node.JS до полного завершения.

Вы можете скачать пример и запустить локально для полноценной демонстрации:

Полный код включает также сервер на Node.JS с функциям `onUpload` – начало и возобновление загрузки, а также `onStatus` – для получения состояния загрузки.

Итого

Мы рассмотрели довольно простой алгоритм возобновляемой загрузки.

Его можно усложнить:

- добавить подсчёт контрольных сумм, проверку целостности пересылаемых файлов,
- для индикации прогресса вместо неточного `xhr.upload.onprogress` – сделать дополнительный запрос к серверу, в который тот будет отдавать текущий прогресс.
- разбивать файл на части и грузить в несколько потоков, несколькими параллельными запросами.

Как можно видеть, возможности современного XMLHttpRequest в плане загрузки файлов приближаются к полноценному файловому менеджеру – полный контроль над заголовками, индикатор прогресса и т.п.

COMET с XMLHttpRequest: длинные опросы

В этой главе мы рассмотрим способ организации COMET, то есть непрерывного получения данных с сервера, который очень прост и подходит в 90% реальных случаев.

Частые опросы

Первое решение, которое приходит в голову для непрерывного получения событий с сервера – это «частые опросы» (polling), т.е периодические запросы на сервер: «эй, я тут, изменилось ли что-нибудь?». Например, раз в 10 секунд.

В ответ сервер во-первых помечает у себя, что клиент онлайн, а во-вторых посылает сообщение, в котором в специальном формате содержится весь пакет событий, накопившихся к данному моменту.

При этом, однако, возможна задержка между появлением и получением данных, как раз в размере этих 10 секунд между запросами.

Другой минус – лишний входящий трафик на сервер. При каждом запросе браузер передает множество заголовков и в ответ получает, кроме данных, также заголовки. Для некоторых приложений трафик заголовков может в 10 и более раз превосходить трафик реальных данных.

Недостатки

- Задержки между событием и уведомлением.
- Лишний трафик и запросы на сервер.

Достоинства

- Простота реализации.

Причём, простота реализации тут достаточно условная. Клиентская часть – довольно проста, а вот сервер получает сразу большой поток запросов.

Даже если клиент ушёл пить чай – его браузер каждые 10 секунд будет «долбить» сервер запросами. Готов ли сервер к такому?

Длинные опросы

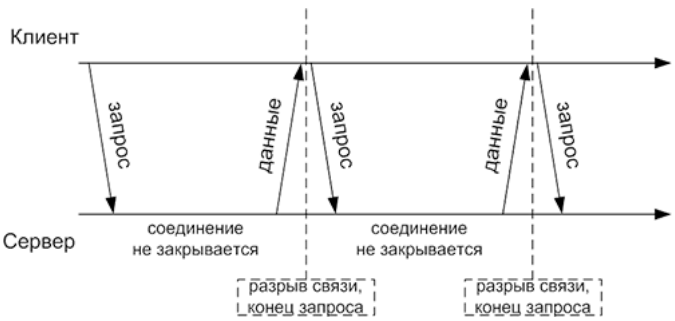
Длинные опросы – отличная альтернатива частым опросам. Они также удобны в реализации, и при этом сообщения доставляются без задержек.

Схема:

1. Отправляется запрос на сервер.
2. Соединение не закрывается сервером, пока не появится сообщение.
3. Когда сообщение появилось – сервер отвечает на запрос, пересылая данные.
4. Браузер тут же делает новый запрос.

Ситуация, когда браузер отправил запрос и держит соединение с сервером, ожидая ответа, является стандартной и прерывается только доставкой сообщений.

Схема коммуникации:



При этом если соединение рвётся само, например, из-за ошибки в сети, то браузер тут же отсылает новый запрос.

Примерный код клиентской части:

```
function subscribe(url) {
  var xhr = new XMLHttpRequest();

  xhr.onreadystatechange = function() {
    if (this.readyState != 4) return;

    if (this.status == 200) {
      onMessage(this.responseText);
    } else {
      onError(this);
    }

    subscribe(url);
  }
  xhr.open("GET", url, true);
  xhr.send();
}
```

Функция `subscribe` делает запрос, при ответе обрабатывает результат, и тут же запускает процесс по новой.

Сервер, конечно же, должен уметь работать с большим количеством таких «ожидающих» соединений.

Демо: чат

Демо:

Область применения

Длинные опросы отлично работают в тех случаях, когда сообщения приходят редко.

При большом количестве частых сообщений график приёма-отправки, приведённый выше, превращается в «пилу». Каждое сообщение – это новый запрос, дополнительный трафик заголовков.

В этих случаях используются другие способы получения данных, подразумевающие непрерывное соединение с сервером. Мы рассмотрим их в следующих главах.

WebSocket

Протокол `WebSocket` (стандарт RFC 6455) предназначен для решения любых задач и снятия ограничений обмена данными между браузером и сервером.

Он позволяет пересылать любые данные, на любой домен, безопасно и почти без лишнего сетевого трафика.

Пример браузерного кода

Для открытия соединения достаточно создать объект `WebSocket`, указав в нём специальный протокол `ws`:

```
var socket = new WebSocket("ws://javascript.ru/ws");
```

У объекта `socket` есть четыре коллбэка: один при получении данных и три – при изменениях в состоянии соединения:

```
socket.onopen = function() {
  alert("Соединение установлено.");
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert('Соединение закрыто чисто');
  } else {
    alert('Обрыв соединения'); // например, "убит" процесс сервера
  }
  alert('Код: ' + event.code + ' причина: ' + event.reason);
};

socket.onmessage = function(event) {
  alert("Получены данные " + event.data);
};

socket.onerror = function(error) {
  alert("Ошибка " + error.message);
};
```

Для посылки данных используется метод **`socket.send(data)`**. Пересылать можно любые данные.

Например, строку:

```
socket.send("Привет");
```

...Или файл, выбранный в форме:

```
socket.send(form.elements[0].file);
```

Просто, не правда ли? Выбираем, что переслать, и `socket.send()`.

Для того, чтобы коммуникация была успешной, сервер должен поддерживать протокол WebSocket.

Чтобы лучше понимать происходящее – посмотрим, как он устроен.

Установление WebSocket-соединения

Протокол `WebSocket` работает *над* HTTP.

Это означает, что при соединении браузер отправляет специальные заголовки, спрашивая: «поддерживает ли сервер WebSocket?».

Если сервер в ответных заголовках отвечает «да, поддерживаю», то дальше HTTP прекращается и общение идёт на специальном протоколе WebSocket, который уже не имеет с HTTP ничего общего.

Установление соединения

Пример запроса от браузера при создании нового объекта `new WebSocket("ws://server.example.com/chat")`:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://javascript.ru
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

Описания заголовков:

GET, Host

Стандартные HTTP-заголовки из URL запроса

Upgrade, Connection

Указывают, что браузер хочет перейти на websocket.

Origin

Протокол, домен и порт, откуда отправлен запрос.

Sec-WebSocket-Key

Случайный ключ, который генерируется браузером: 16 байт в кодировке Base64.

Sec-WebSocket-Version

Версия протокола. Текущая версия: 13.

Все заголовки, кроме `GET` и `Host`, браузер генерирует сам, без возможности вмешательства JavaScript.

> **Такой XMLHttpRequest создать нельзя**
>
> Создать подобный XMLHttpRequest-запрос (подделать `WebSocket`) невозможно, по одной простой причине: указанные выше заголовки запрещены к установке методом `setRequestHeader`.

Сервер может проанализировать эти заголовки и решить, разрешает ли он **`WebSocket`** с данного домена **`Origin`**.

Ответ сервера, если он понимает и разрешает `WebSocket`-подключение:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

Здесь строка `Sec-WebSocket-Accept` представляет собой перекодированный по специальному алгоритму ключ `Sec-WebSocket-Key`. Браузер использует её для проверки, что ответ предназначается именно ему.

Затем данные передаются по специальному протоколу, структура которого («фреймы») изложена далее. И это уже совсем не HTTP.

Расширения и подпротоколы

Также возможны дополнительные заголовки `Sec-WebSocket-Extensions` и `Sec-WebSocket-Protocol`, описывающие расширения и подпротоколы (subprotocol), которые поддерживает данный клиент.

Посмотрим разницу между ними на двух примерах:

- Заголовок `Sec-WebSocket-Extensions: deflate-frame` означает, что браузер поддерживает модификацию протокола, обеспечивающую сжатие данных.

  Это говорит не о самих данных, а об улучшении способа их передачи. Браузер сам формирует этот заголовок.

- Заголовок `Sec-WebSocket-Protocol: soap, wamp` говорит о том, что по WebSocket браузер собирается передавать не просто какие-то данные, а данные в протоколах SOAP или WAMP («The WebSocket Application Messaging Protocol»). Стандартные подпротоколы регистрируются в специальном каталоге IANA.

Этот заголовок браузер поставит, если указать второй необязательный параметр `WebSocket` :

```
var socket = new WebSocket("ws://javascript.ru/ws", ["soap", "wamp"]);
```

При наличии таких заголовков сервер может выбрать расширения и подпротоколы, которые он поддерживает, и ответить с ними.

Например, запрос:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://javascript.ru
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

Ответ:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

В ответе выше сервер указывает, что поддерживает расширение `deflate-frame` , а из запрошенных подпротоколов – только SOAP.

WSS

Соединение `WebSocket` можно открывать как `WS://` или как `WSS://` . Протокол `WSS` представляет собой WebSocket над HTTPS.

Кроме большей безопасности, у **`WSS`** есть важное преимущество перед обычным **`WS`** – большая вероятность соединения.

Дело в том, что HTTPS шифрует трафик от клиента к серверу, а HTTP – нет.

Если между клиентом и сервером есть прокси, то в случае с HTTP все WebSocket-заголовки и данные передаются через него. Прокси имеет к ним доступ, ведь они никак не шифруются, и может расценить происходящее как нарушение протокола HTTP, обрезать заголовки или оборвать передачу.

А в случае с `WSS` весь трафик сразу кодируется и через прокси проходит уже в закодированном виде. Поэтому заголовки гарантированно пройдут, и общая вероятность соединения через `WSS` выше, чем через `WS` .

Формат данных

Полное описание протокола содержится в RFC 6455 .

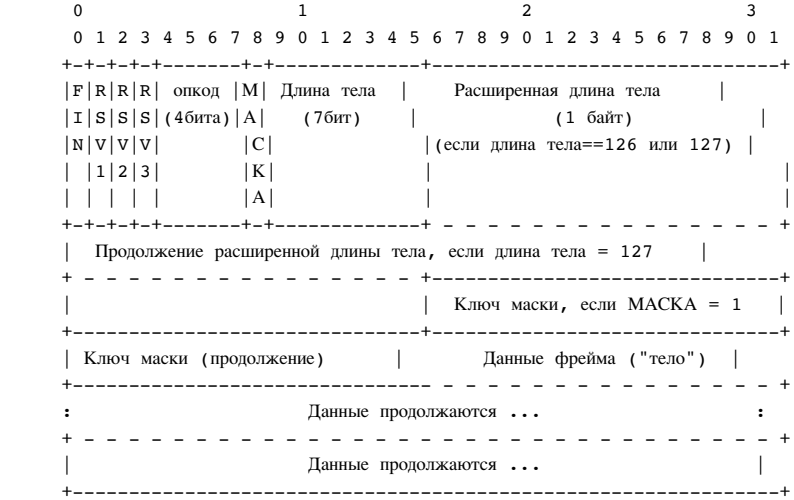
Здесь представлено частичное описание с комментариями самых важных его частей. Если вы хотите понять стандарт, то рекомендуется сначала прочитать это описание.

Описание фрейма

В протоколе WebSocket предусмотрены несколько видов пакетов («фреймов»).

Они делятся на два больших типа: фреймы с данными («data frames») и управляющие («control frames»), предназначенные для проверки связи (PING) и закрытия соединения.

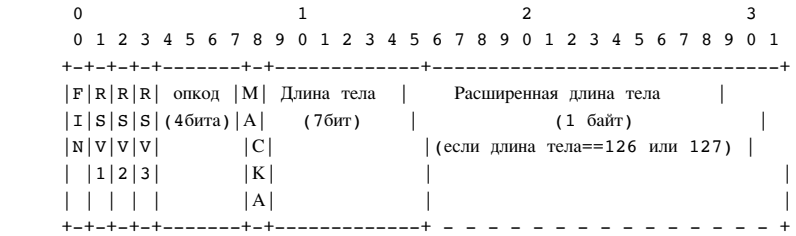
Фрейм, согласно стандарту, выглядит так:



С виду – не очень понятно, во всяком случае, для большинства людей.

Позвольте пояснить: читать следует слева-направо, сверху-вниз, каждая горизонтальная полоска это 32 бита.

То есть, вот первые 32 бита:



Сначала идёт бит FIN (вертикальная надпись на рисунке), затем биты RSV1, RSV2, RSV3 (их смысл раскрыт ниже), затем «опкод», «MACKA» и, наконец, «Длина тела», которая занимает 7 бит. Затем, если «Длина тела» равна 126 или 127, идёт «Расширенная длина тела», потом (на следующей строке, то есть после первых 32 бит) будет её продолжение, ключ маски, и потом данные.

А теперь – подробное описание частей фрейма, то есть как именно передаются сообщения:

FIN: 1 бит

Одно сообщение, если оно очень длинное (вызовом `send` можно передать хоть целый файл), может состоять из множества фреймов («быть фрагментированным»).

У всех фреймов, кроме последнего, этот фрагмент установлен в `0`, у последнего – в `1`.

Если сообщение состоит из одного-единственного фрейма, то он в нём `FIN` равен `1`.

RSV1, RSV2, RSV3: 1 бит каждый

В обычном WebSocket равны `0`, предназначены для расширений протокола. Расширение может записать в эти биты свои значения.

Опкод: 4 бита

Задаёт тип фрейма, который позволяет интерпретировать находящиеся в нём данные. Возможные значения:

- `0x1` обозначает текстовый фрейм.
- `0x2` обозначает двоичный фрейм.
- `0x3-7` зарезервированы для будущих фреймов с данными.
- `0x8` обозначает закрытие соединения этим фреймом.
- `0x9` обозначает PING.
- `0xA` обозначает PONG.
- `0xB-F` зарезервированы для будущих управляющих фреймов.
- `0x0` обозначает фрейм-продолжение для фрагментированного сообщения. Он интерпретируется, исходя из ближайшего предыдущего ненулевого типа.

Маска: 1 бит

Если этот бит установлен, то данные фрейма маскированы. Более подробно маску и маскирование мы рассмотрим далее.

Длина тела: 7 битов, 7+16 битов, или 7+64 битов

Если значение поле «Длина тела» лежит в интервале `0-125`, то оно обозначает длину тела (используется далее). Если `126`, то следующие 2 байта интерпретируются как 16-битное беззнаковое целое число, содержащее длину тела. Если `127`, то следующие 8 байт интерпретируются как 64-битное беззнаковое целое, содержащее длину.

Такая хитрая схема нужна, чтобы минимизировать накладные расходы. Для сообщений длиной `125` байт и меньше хранение длины потребует всего 7 битов, для бóльших (до 65536) – 7 битов + 2 байта, ну а для ещё бóльших – 7 битов и 8 байт. Этого хватит для хранения длины сообщения размером в гигабайт и более.

Ключ маски: 4 байта.

Если бит `Маска` установлен в 0, то этого поля нет. Если в `1` то эти байты содержат маску, которая налагается на тело (см. далее).

Данные фрейма (тело)

Состоит из «данных расширений» и «данных приложения», которые идут за ними. Данные расширений определяются конкретными расширениями протокола и по умолчанию отсутствуют. Длина тела должна быть равна указанной в заголовке.

Примеры

Некоторые примеры сообщений:

- Нефрагментированное текстовое сообщение `Hello` без маски:

```
0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f (содержит "Hello")
```

  В заголовке первый байт содержит `FIN=1` и `опкод=0x1` (получается `10000001` в двоичной системе, то есть `0x81` – в 16-ричной), далее идёт длина `0x5`, далее текст.

- Фрагментированное текстовое сообщение `Hello World` из трёх частей, без маски, может выглядеть так:

```
0x01 0x05 0x48 0x65 0x6c 0x6c 0x6f (содержит "Hello")
0x00 0x01 0x20 (содержит " ")
```

```
0x80 0x05 0x57 0x6f 0x72 0x6c 0x64 (содержит "World")
```

- У первого фрейма `FIN=0` и текстовый опкод `0x1`.
- У второго `FIN=0` и опкод `0x0`. При фрагментации сообщения, у всех фреймов, кроме первого, опкод пустой (он один на всё сообщение).
- У третьего, последнего фрейма `FIN=1`.

А теперь посмотрим на все те замечательные возможности, которые даёт этот формат фрейма.

Фрагментация

Позволяет отправлять сообщения в тех случаях, когда на момент начала посылки полный размер ещё неизвестен.

Например, идёт поиск в базе данных и что-то уже найдено, а что-то ещё может быть позже.

- У всех сообщений, кроме последнего, бит `FIN=0`.
- Опкод указывается только у первого, у остальных он должен быть равен `0x0`.

PING / PONG

В протокол встроена проверка связи при помощи управляющих фреймов типа PING и PONG.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

Этот функционал встроен в браузерную реализацию, так что браузер ответит на PING сервера, но управлять им из JavaScript нельзя.

Иначе говоря, сервер всегда знает, жив ли посетитель или у него проблема с сетью.

Чистое закрытие

При закрытии соединения сторона, желающая это сделать (обе стороны в WebSocket равноправны) отправляет закрывающий фрейм (опкод `0x8`), в теле которого указывает причину закрытия.

В браузерной реализации эта причина будет содержаться в свойстве `reason` события `onclose`.

Наличие такого фрейма позволяет отличить «чистое закрытие» от обрыва связи.

В браузерной реализации событие `onclose` при чистом закрытии имеет `event.wasClean = true`.

Коды закрытия

Коды закрытия вебсокета `event.code`, чтобы не путать их с HTTP-кодами, состоят из 4 цифр:

`1000`

Нормальное закрытие.

`1001`

Удалённая сторона «исчезла». Например, процесс сервера убит или браузер перешёл на другую страницу.

`1002`

Удалённая сторона завершила соединение в связи с ошибкой протокола.

`1003`

Удалённая сторона завершила соединение в связи с тем, что она получила данные, которые не может принять. Например, сторона, которая понимает только текстовые данные, может закрыть соединение с таким кодом, если приняла бинарное сообщение.

Атака «отравленный кэш»

В ранних реализациях WebSocket существовала уязвимость, называемая «отравленный кэш» (cache poisoning).

Она позволяла атаковать кэширующие прокси-сервера, в частности, корпоративные.

Атака осуществлялась так:

1. Хакер заманивает доверчивого посетителя (далее Жертва) на свою страницу.
2. Страница открывает `WebSocket`-соединение на сайт хакера. Предполагается, что Жертва сидит через прокси. Собственно, на прокси и направлена эта атака.
3. Страница формирует специального вида WebSocket-запрос, который (и здесь самое главное!) ряд прокси серверов не понимают.

   Они пропускают начальный запрос через себя (который содержит `Connection: upgrade`) и думают, что далее идёт уже следующий HTTP-запрос.

   ...Но на самом деле там данные, идущие через вебсокет! И обе стороны вебсокета (страница и сервер) контролируются Хакером. Так что хакер может передать в них нечто похожее на GET-запрос к известному ресурсу, например `http://code.jquery.com/jquery.js`, а сервер ответит «якобы кодом jQuery» с кэширующими заголовками.

   Прокси послушно проглотит этот ответ и закэширует «якобы jQuery».
4. В результате при загрузке последующих страниц любой пользователь, использующий тот же прокси, что и Жертва, получит вместо `http://code.jquery.com/jquery.js` хакерский код.

Поэтому эта атака и называется «отравленный кэш».

Такая атака возможна не для любых прокси, но при анализе уязвимости было показано, что она не теоретическая, и уязвимые прокси действительно есть.

Поэтому придумали способ защиты – «маску».

Маска для защиты от атаки

Для того, чтобы защититься от атаки, и придумана маска.

Ключ маски – это случайное 32-битное значение, которое варьируется от пакета к пакету. Тело сообщения проходит через XOR `^` с маской, а получатель восстанавливает его повторным XOR с ней (можно легко доказать, что `(x ^ a) ^ a == x`).

Маска служит двум целям:

1. Она генерируется браузером. Поэтому теперь хакер не сможет управлять реальным содержанием тела сообщения. После накладывания маски оно превратится в бинарную мешанину.
2. Получившийся пакет данных уже точно не может быть воспринят промежуточным прокси как HTTP-запрос.

Наложение маски требует дополнительных ресурсов, поэтому протокол WebSocket не требует её.

Если по этому протоколу связываются два клиента (не обязательно браузеры), доверяющие друг другу и посредникам, то можно поставить бит `Маска` в `0`, и тогда ключ маски не указывается.

Пример

Рассмотрим прототип чата на WebSocket и Node.JS.

HTML: посетитель отсылает сообщения из формы и принимает в `div`

```
<!-- форма для отправки сообщений -->
<form name="publish">
  <input type="text" name="message">
  <input type="submit" value="Отправить">
</form>

<!-- здесь будут появляться входящие сообщения -->
<div id="subscribe"></div>
```

Код на клиенте:

```
// создать подключение
var socket = new WebSocket("ws://localhost:8081");

// отправить сообщение из формы publish
document.forms.publish.onsubmit = function() {
  var outgoingMessage = this.message.value;

  socket.send(outgoingMessage);
  return false;
};

// обработчик входящих сообщений
socket.onmessage = function(event) {
  var incomingMessage = event.data;
  showMessage(incomingMessage);
};

// показать сообщение в div#subscribe
function showMessage(message) {
  var messageElem = document.createElement('div');
  messageElem.appendChild(document.createTextNode(message));
  document.getElementById('subscribe').appendChild(messageElem);
}
```

Серверный код можно писать на любой платформе. В нашем случае это будет Node.JS, с использованием модуля ws:

```
var WebSocketServer = new require('ws');

// подключенные клиенты
var clients = {};

// WebSocket-сервер на порту 8081
var webSocketServer = new WebSocketServer.Server({
  port: 8081
});
webSocketServer.on('connection', function(ws) {

  var id = Math.random();
  clients[id] = ws;
  console.log("новое соединение " + id);

  ws.on('message', function(message) {
    console.log('получено сообщение ' + message);

    for (var key in clients) {
      clients[key].send(message);
    }
  });

  ws.on('close', function() {
    console.log('соединение закрыто ' + id);
    delete clients[id];
  });

});
```

Рабочий пример можно скачать: websocket.zip. Понадобится поставить два модуля: `npm install node-static && npm install ws`.

Итого

WebSocket – современное средство коммуникации. Кросс-доменное, универсальное, безопасное.

На текущий момент он работает в браузерах IE10+, FF11+, Chrome 16+, Safari 6+, Opera 12.5+. В более старых версиях FF, Chrome, Safari, Opera есть поддержка черновых редакций протокола.

Там, где вебсокеты не работают – обычно используют другие транспорты, например `IFRAME`. Вы найдёте их в других статьях этого раздела.

Есть и готовые библиотеки, реализующие функционал COMET с использованием сразу нескольких транспортов, из которых вебсокет имеет приоритет. Как правило, библиотеки состоят из двух частей: клиентской и серверной.

Например, для Node.JS одной из самых известных библиотек является Socket.IO.

К недостаткам библиотек следует отнести то, что некоторые продвинутые возможности WebSocket, такие как двухсторонний обмен бинарными данными, в них недоступны. С другой – в большинстве случаев стандартного текстового обмена вполне достаточно.

Протокол JSONP

Если создать тег `<script src>`, то при добавлении в документ запустится процесс загрузки `src`. В ответ сервер может прислать скрипт, содержащий нужные данные.

Таким образом можно запрашивать данные с любого сервера, в любом браузере, без каких-либо разрешений и дополнительных проверок.

Протокол JSONP – это «надстройка» над таким способом коммуникации. Здесь мы рассмотрим его использование в деталях.

Запрос

Простейший пример запроса:

```
function addScript(src) {
  var elem = document.createElement("script");
  elem.src = src;
  document.head.appendChild(elem);
}

addScript('user?id=123');
```

Такой вызов добавит в `<head>` документа тег:

```
<script src="/user?id=123"></script>
```

При добавлении тега `<script>` с внешним `src` в документ браузер тут же начинает его скачивать, а затем – выполняет.

В данном случае браузер запросит скрипт с URL `/user?id=123` и выполнит.

Обработка ответа, JSONP

В примере выше рассмотрено создание запроса, но как получить ответ? Допустим, сервер хочет прислать объект с данными.

Конечно, он может присвоить её в переменную, например так:

```
// ответ сервера
var user = {name: "Вася", age: 25 };
```

...А браузер по `script.onload` отловит окончание загрузки и прочитает значение `user`.

Но что, если одновременно делается несколько запросов? Получается, нужно присваивать в разные переменные.

Протокол JSONP как раз и призван облегчить эту задачу.

Он очень простой:

1. Вместе с запросом клиент в специальном, заранее оговорённом, параметре передаёт название функции.

   Обычно такой параметр называется `callback`. Например:

   ```
   addScript('user?id=123&callback=onUserData');
   ```

2. Сервер кодирует данные в JSON и оборачивает их в вызов функции, название которой получает из параметра `callback`:

   ```
   // ответ сервера
   onUserData({
     name: "Вася",
     age: 25
   });
   ```

Это и называется JSONP («JSON with Padding»).

> ⚠ Аспект безопасности
>
> Клиентский код должен доверять серверу при таком запросе. Ведь серверу ничего не стоит добавить в скрипт любые команды.

Реестр CallbackRegistry

В примере выше функция `onUserData` должна быть глобальной, ведь `<script src>` выполняется в глобальной области видимости.

Хотелось бы не загрязнять глобальное пространство имён, или по крайней мере свести загрязнение к минимуму.

Как правило, для этого создают один глобальный объект «реестр», который мы назовём `CallbackRegistry`. Далее для каждого запроса в нём генерируется временная функция.

Тег будет выглядеть так:

```
<script src="user?id=123&callback=CallbackRegistry.func12345"></script>
```

Сервер обернёт ответ в функцию `CallbackRegistry.func12345`, она вызывает нужный обработчик и очищает память, удаляя себя.

Далее мы посмотрим более полный код всего этого, но перед этим – важный момент! Нужно предусмотреть обработку ошибок.

Обнаружение ошибок

При запросе данных при помощи `SCRIPT` возможны различные ошибки:

1. Скрипт может не загрузиться: отказ в соединении, разрыв связи...
2. Ошибка HTTP, например 500.
3. Скрипт загрузился, но внутри некорректен и не вызывает функцию. Например, на сервере произошла ошибка и в ответе передан её текст, а вовсе не данные.

Чтобы отловить их все «одним махом», используем следующий алгоритм:

1. Создаётся `<script>`.
2. На `<script>` ставятся обработчики `onreadystatechange` (для старых IE) и `onload/onerror` (для остальных браузеров).
3. При загрузке скрипт выполняет функцию-коллбэк `CallbackRegistry...`. Пусть она при запуске ставит флажок «все ок». А мы в обработчиках проверим – если флага нет, то функция не вызывалась – стало быть, ошибка при загрузке или содержимое скрипта некорректно.

Полный пример

Итак, код функции, которая вызывается с `url` и коллбэками.

Он совсем небольшой, а без комментариев был бы ещё меньше:

```
var CallbackRegistry = {}; // реестр

// при успехе вызовет onSuccess, при ошибке onError
function scriptRequest(url, onSuccess, onError) {

  var scriptOk = false; // флаг, что вызов прошел успешно

  // сгенерировать имя JSONP-функции для запроса
  var callbackName = 'cb' + String(Math.random()).slice(-6);

  // укажем это имя в URL запроса
  url += ~url.indexOf('?') ? '&' : '?';
  url += 'callback=CallbackRegistry.' + callbackName;

  // ..и создадим саму функцию в реестре
  CallbackRegistry[callbackName] = function(data) {
    scriptOk = true; // обработчик вызвался, указать что всё ок
    delete CallbackRegistry[callbackName]; // можно очистить реестр
    onSuccess(data); // и вызвать onSuccess
  };

  // эта функция сработает при любом результате запроса
  // важно: при успешном результате - всегда после JSONP-обработчика
  function checkCallback() {
    if (scriptOk) return; // сработал обработчик?
    delete CallbackRegistry[callbackName];
    onError(url); // нет - вызвать onError
  }

  var script = document.createElement('script');

  // в старых IE поддерживается только событие, а не onload/onerror
  // в теории 'readyState=loaded' означает "скрипт загрузился",
  // а 'readyState=complete' -- "скрипт выполнился", но иногда
  // почему-то случается только одно из них, поэтому проверяем оба
  script.onreadystatechange = function() {
    if (this.readyState == 'complete' || this.readyState == 'loaded') {
      this.onreadystatechange = null;
      setTimeout(checkCallback, 0); // Вызвать checkCallback - после скрипта
    }
  }
```

```
  // события script.onload/onerror срабатывают всегда после выполнения скрипта
  script.onload = script.onerror = checkCallback;
  script.src = url;

  document.body.appendChild(script);
}
```

Пример использования:

```
function ok(data) {
  alert( "Загружен пользователь " + data.name );
}

function fail(url) {
  alert( 'Ошибка при запросе ' + url );
}

// Внимание! Ответы могут приходить в любой последовательности!
scriptRequest("user?id=123", ok, fail); // Загружен
scriptRequest("/badurl.js", ok, fail); // fail, 404
scriptRequest("/", ok, fail); // fail, 200 но некорректный скрипт
```

Демо, по нажатию на кнопке запускаются запросы выше:

COMET

COMET через `SCRIPT` реализуется при помощи длинных опросов, также как мы обсуждали в главе COMET с XMLHttpRequest: длинные опросы.

То есть, создаётся тег `<script>`, браузер запрашивает скрипт у сервера и... Сервер оставляет соединение висеть, пока не появится, что сказать. Когда сервер хочет отправить сообщение – он отвечает, используя формат JSONP. И, тут же, новый запрос...

Server Side Events -- события с сервера

Сразу заметим, что на текущий момент этот способ поддерживают все современные браузеры, кроме IE.

Современный стандарт Server-Sent Events позволяет браузеру создавать специальный объект `EventSource`, который сам обеспечивает соединение с сервером, делает пересоединение в случае обрыва и генерирует события при поступлении данных.

Он, по дизайну, может меньше, чем WebSocket'ы.

С другой стороны, Server Side Events проще в реализации, работают по обычному протоколу HTTP и сразу поддерживают ряд возможностей, которые для WebSocket ещё надо реализовать.

Поэтому в тех случаях, когда нужна преимущественно односторонняя передача данных от сервера к браузеру, они могут быть удачным выбором.

Получение сообщений

При создании объекта `new EventSource(src)` браузер автоматически подключается к адресу `src` и начинает получать с него события:

```
var eventSource = new EventSource("/events/subscribe");

eventSource.onmessage = function(e) {
  console.log("Пришло сообщение: " + e.data);
};
```

Чтобы соединение успешно открылось, сервер должен ответить с заголовком `Content-Type: text/event-stream`, а затем оставить соединение висящим и писать в него сообщения в специальном формате:

```
data: Сообщение 1

data: Сообщение 2

data: Сообщение 3
data: из двух строк
```

- Каждое сообщение пишется после `data:`. Если после двоеточия есть пробел, то он игнорируется.
- Сообщения разделяются двумя строками `\n\n`.
- Если нужно переслать перевод строки, то сообщение разделяется. Каждая следующая строка пересылается отдельным `data:`.

  В частности, две последние строки в примере выше составляют одно сообщение: `"Сообщение 3\nиз двух строк"`.

Здесь все очень просто и удобно, кроме разделения сообщения при переводе строки. Но, если подумать – это не так уж страшно: на практике сложные сообщения обычно передаются в формате JSON. А перевод строки в нём кодируется как `\n`.

Соответственно, многострочные данные будут пересылаться так:

```
data: {"user":"Вася","message":"Сообщение 3\n из двух строк"}
```

...То есть, строка `data:` будет одна, и никаких проблем с разделением сообщения нет.

Восстановление соединения

При создании объекта браузер автоматически подключается к серверу, а при обрыве – пытается его возобновить.

Это очень удобно, никакой другой транспорт не обладает такой встроенной способностью.

> **Как серверу полностью закрыть соединение?**
>
> При любом закрытии соединения, в том числе если сервер ответит на запрос и закроет соединение сам – браузер через короткое время повторит свой запрос.
>
> Есть лишь два способа, которыми сервер может «отшить» надоедливый `EventSource`:
>
> - Ответить со статусом не 200.
> - Ответить с `Content-Type`, не совпадающим с `text/event-stream`.

Между попытками возобновить соединение будет пауза, начальное значение которой зависит от браузера (1-3 секунды) и может быть изменено сервером через указание `retry:` в ответе:

```
retry: 15000
data: Поставлена задержка 15 секунд
```

Браузер, со своей стороны, может закрыть соединение вызовом `close()`:

```
var eventSource = new EventSource(...);

eventSource.close();
```

При этом дальнейших попыток соединения не будет. Открыть обратно этот объект тоже нельзя, можно создать новый `EventSource`.

Идентификатор id

Для того, чтобы продолжить получение событий с места разрыва, стандарт предусматривает идентификацию событий через `id`.

Сервер может указать его в ответе:

```
data: Сообщение 1
id: 1

data: Сообщение 2
id: 2

data: Сообщение 3
data: из двух строк
id: 3
```

При получении `id:` браузер:

- Устанавливает свойство `eventSource.lastEventId` в его значение.
- При пересоединении пошлёт заголовок `Last-Event-ID` с этим `id`, так что сервер сможет переслать последующие, пропущенные, сообщения.

Обратим внимание: `id` шлётся *не перед сообщением, а после него*, чтобы обновление `lastEventId` произошло, когда браузер всё уже точно получил.

Статус соединения readyState

У объекта `EventSource` есть свойство `readyState`, которое содержит одно из значений (выдержка из стандарта):

```
const unsigned short CONNECTING = 0; // в процессе (пере-)соединения
const unsigned short OPEN = 1;       // соединение установлено
const unsigned short CLOSED = 2;     // соединение закрыто
```

При создании объекта и при разрыве оно автоматически равно `CONNECTING`.

События

Событий всего три:

- `onmessage` – пришло сообщение, доступно как `event.data`
- `onopen` – при успешном установлении соединения
- `onerror` – при ошибке соединения.

Например:

```
var eventSource = new EventSource('digits');

eventSource.onopen = function(e) {
  console.log("Соединение открыто");
};

eventSource.onerror = function(e) {
  if (this.readyState == EventSource.CONNECTING) {
    console.log("Соединение порвалось, пересоединяемся...");
  } else {
    console.log("Ошибка, состояние: " + this.readyState);
  }
};

eventSource.onmessage = function(e) {
  console.log("Пришли данные: " + e.data);
};
```

Своё имя события: event

По умолчанию на события срабатывает обработчик `onmessage`, но можно сделать и свои события. Для этого сервер должен указать перед событием его имя после `event:`.

Например:

```
event: join
data: Вася

data: Привет

event: leave
data: Вася
```

Сообщение по умолчанию имеет имя `message`.

Для обработки своих имён событий необходимо ставить обработчик при помощи **`addEventListener`**.

Пример кода для обработки:

```
eventSource.addEventListener('join', function(e) {
  alert( 'Пришёл ' + e.data );
});

eventSource.addEventListener('message', function(e) {
  alert( 'Сообщение ' + e.data );
});

eventSource.addEventListener('leave', function(e) {
  alert( 'Ушёл ' + e.data );
});
```

Демо

В примере ниже сервер посылает в соединение числа от 1 до 3, а затем – событие `bye` и закрывает соединение. Браузер автоматически откроет его заново.

Кросс-доменность

`EventSource` поддерживает кросс-доменные запросы, аналогично `XMLHttpRequest`. Для этого у конструктора есть второй аргумент – объект, который нужно передать так:

```
var source = new EventSource("http://pupkin.ru/stream", {
  withCredentials: true
});
```

Второй аргумент сделан объектом с расчётом на будущее. Пока что никаких других свойств там не поддерживается, только `withCredentials`.

Сервер при этом получит заголовок `Origin` с доменом запроса и должен ответить с заголовком `Access-Control-Allow-Origin` (и `Access-Control-Allow-Credentials`, если стоит `withCredentials`), в точности как в главе XMLHttpRequest: кросс-доменные запросы.

При кросс-доменных запросах у событий `event` также появится дополнительное свойство `origin`, содержащее адрес источника, откуда пришли данные. Его можно использовать для дополнительной проверки со стороны браузера:

```
eventSource.addEventListener('message', function(e) {
  if (e.origin != 'http://javascript.ru') return;
  alert( 'Сообщение ' + e.data );
});
```

Итого

Объект `EventSource` предназначен для передачи текстовых сообщений с сервера, используя обычный протокол HTTP.

Он предлагает не только передачу сообщений, но и встроенную поддержку важных вспомогательных функций:

- События `event`.
- Автоматическое пересоединение, с настраиваемой задержкой `retry`.
- Проверка текущего состояния подключения по `readyState`.
- Идентификаторы сообщений `id` для точного возобновления потока данных, последний полученный идентификатор передаётся в заголовке `Last-Event-ID`.
- Кросс-доменность CORS.

Этот набор функций делает EventSource достойной альтернативой WebSocket, которые хоть и потенциально мощнее, но требуют реализации всех этих функций на клиенте и сервере, поверх протокола.

Поддержка – все браузеры, кроме IE.

- Синтаксис:

  ```
  var source = new EventSource(src[, credentials]); // src - адрес с любого домена
  ```

  Второй необязательный аргумент, если указан в виде `{ withCredentials: true }`, инициирует отправку Cookie и данных авторизации при кросс-доменных запросах.

  Безопасность при кросс-доменных запросах обеспечивается аналогично `XMLHttpRequest`.

- Свойства объекта:

  `readyState`

  Текущее состояние соединения, одно из `EventSource.CONNECTING (=0)`, `EventSource.OPEN (=1)` или `EventSource.CLOSED (=2)`.

  `lastEventId`

  Последнее полученное `id`, если есть. При возобновлении соединения браузер указывает это значение в заголовке `Last-Event-ID`.

  `url`, **`withCredentials`**

  Параметры, переданные при создании объекта. Менять их нельзя.

- Методы:

  `close()`

  Закрывает соединение.

- События:

  `onmessage`

  При сообщении, данные – в `event.data`.

  `onopen`

  При установлении соединения.

  `onerror`

  При ошибке, в том числе – закрытии соединения по инициативе сервера.

  Эти события можно ставить напрямую через свойство: `source.onmessage = ...`.

  Если сервер присылает имя события в `event:`, то такие события нужно обрабатывать через `addEventListener`.

- Формат ответа сервера:

  Сервер присылает пустые строки, либо строки, начинающиеся с:

  - `data:` – сообщение, несколько таких строк подряд склеиваются и образуют одно сообщение.
  - `id:` – обновляет `lastEventId`.
  - `retry:` – указывает паузу между пересоединениями, в миллисекундах. JavaScript не может указать это значение, только сервер.
  - `event:` – имя события, должен быть перед `data:`.

IFRAME для AJAX и COMET

Эта глава посвящена `IFRAME` – самому древнему и кросс-браузерному способу AJAX-запросов.

Сейчас он используется, разве что, для поддержки кросс-доменных запросов в IE7- и, что чуть более актуально, для реализации COMET в IE9-.

Для общения с сервером создается невидимый `IFRAME`. В него отправляются данные, и в него же сервер пишет ответ.

Введение

Сначала – немного вспомогательных функций и особенности работы с `IFRAME`.

Двуличность IFRAME: окно+документ

Что такое IFRAME? На этот вопрос у браузера два ответа

```
1. IFRAME -- это HTML-тег: <code>&lt;iframe&gt;</code> со стандартным набором свойств.

- Тег можно создавать в JavaScript
- У тега есть стили, можно менять.
- К тегу можно обратиться через `document.getElementById` и другие методы.
2. IFRAME -- это окно браузера, вложенное в основное

- IFRAME -- такое же по функционалу окно браузера, как и основное, с адресом и т.п.
    - Если документ в `IFRAME` и внешнее окно находятся на разных доменах, то прямой вызов методов друг друга невозможен.
- Ссылку на это окно можно получить через `window.frames['имя фрейма']`.
```

Для достижения цели мы будем работать как с тегом, так и с окном. Они, конечно же, взаимосвязаны.

В теге **<iframe>** свойство **contentWindow** хранит ссылку на окно.

Окна также содержатся в коллекции `window.frames`.

Например:

```
// Окно из ифрейма
var iframeWin = iframe.contentWindow;

// Можно получить и через frames, если мы знаем имя ифрейма (и оно у него есть)
var iframeWin = window.frames[iframe.name];
iframeWin.parent == window; // parent из iframe указывает на родительское окно

// Документ не будет доступен, если iframe с другого домена
var iframeDoc = iframe.contentWindow.document;
```

Больше информации об ифреймах вы можете получить в главе Общение между окнами и фреймами.

IFRAME и история посещений

`IFRAME` – полноценное окно, поэтому навигация в нём попадает в историю посещений.

Это означает, что при нажатии кнопки «Назад» браузер вернёт посетителя назад не в основном окне, а в ифрейме. В лучшем случае – браузер возьмёт предыдущее состояние ифрейма из кэша и посетитель просто подумает, что кнопка не сработала. В худшем – в ифрейм будет сделан предыдущий запрос, а это уже точно ни к чему.

Наши запросы в ифрейм – служебные и для истории не предназначены. К счастью, есть ряд техник, которые позволяют обойти проблему.

- Ифрейм нужно создавать динамически, через JavaScript.
- Когда ифрейм уже создан, то единственный способ поменять его `src` без попадания запроса в историю посещений:

  ```
  // newSrc - новый адрес
  iframeDoc.location.replace(newSrc);
  ```

  Вы можете возразить: «но ведь `iframeDoc` не всегда доступен! `iframe` может быть с другого домена – как быть тогда?». Ответ: вместо смены `src` этого ифрейма – создать новый, с новым `src`.
- POST-запросы в `iframe` всегда попадают в историю посещений.
- ... Но если `iframe` удалить, то лишняя история тоже исчезнет :). Сделать это можно по окончании запроса.

Таким образом, общий принцип использования **IFRAME**: динамически создать, сделать запрос, удалить.

Бывает так, что удалить по каким-то причинам нельзя, тогда возможны проблемы с историей, описанные выше.

Функция createIframe

Приведенная ниже функция `createIframe(name, src, debug)` кросс-браузерно создаёт ифрейм с данным именем и `src`.

Аргументы:

name

Имя и `id` ифрейма

src

Исходный адрес ифрейма. Необязательный параметр.

debug

Если параметр задан, то ифрейм после создания не прячется.

```
function createIframe(name, src, debug) {
  src = src || 'javascript:false'; // пустой src

  var tmpElem = document.createElement('div');

  // в старых IE нельзя присвоить name после создания iframe
```

```
  // поэтому создаём через innerHTML
  tmpElem.innerHTML = '<iframe name="' + name + '" id="' + name + '" src="' + src + '">';
  var iframe = tmpElem.firstChild;

  if (!debug) {
    iframe.style.display = 'none';
  }

  document.body.appendChild(iframe);

  return iframe;
}
```

Ифрейм здесь добавляется к `document.body`. Конечно, вы можете исправить этот код и добавлять его в любое другое место документа.

Кстати, при вставке, если не указан `src`, тут же произойдёт событие `iframe.onload`. Пока обработчиков нет, поэтому оно будет проигнорировано.

Функция postToIframe

Функция `postToIframe(url, data, target)` отправляет POST-запрос в ифрейм с именем `target`, на адрес `url` с данными `data`.

Аргументы:

`url`

URL, на который отправлять запрос.

`data`

Объект содержит пары `ключ:значение` для полей формы. Значение будет приведено к строке.

`target`

Имя ифрейма, в который отправлять данные.

```
// Например: postToIframe('/vote', {mark:5}, 'frame1')

function postToIframe(url, data, target) {
  var phonyForm = document.getElementById('phonyForm');
  if (!phonyForm) {
    // временную форму создаем, если нет
    phonyForm = document.createElement("form");
    phonyForm.id = 'phonyForm';
    phonyForm.style.display = "none";
    phonyForm.method = "POST";
    document.body.appendChild(phonyForm);
  }

  phonyForm.action = url;
  phonyForm.target = target;

  // заполнить форму данными из объекта
  var html = [];
  for (var key in data) {
    var value = String(data[key]).replace(/"/g, "&quot;");
    // в старых IE нельзя указать name после создания input
    // поэтому используем innerHTML вместо DOM-методов
    html.push("<input type='hidden' name=\"" + key + "\" value=\"" + value + "\">");
  }
  phonyForm.innerHTML = html.join('');

  phonyForm.submit();
}
```

Эта функция формирует форму динамически, но, конечно, это лишь один из возможных сценариев использования.

В `IFRAME` можно отправлять и существующую форму, включающую файловые и другие поля.

Запросы GET и POST

Общий алгоритм обращения к серверу через ифрейм:

1. Создаём `iframe` со случайным именем `iframeName`.
2. Создаём в основном окне объект `CallbackRegistry`, в котором в `CallbackRegistry[iframeName]` сохраняем функцию, которая будет обрабатывать результат.
3. Отправляем GET или POST-запрос в него.
4. Сервер отвечает как-то так:

   ```
   <script>
     parent.CallbackRegistry[window.name]({данные});
   </script>
   ```

   ...То есть, вызывает из основного окна функцию обработки (`window.name` в ифрейме – его имя).
5. Дополнительно нужен обработчик `iframe.onload` – он сработает и проверит, выполнилась ли функция `CallbackRegistry[window.name]`. Если нет, значит какая-то ошибка. Сервер при нормальном потоке выполнения всегда отвечает её вызовом.

Подробнее можно понять процесс, взглянув на код.

Мы будем использовать в нём две функции – одну для GET, другую – для POST:

- `iframeGet(url, onSuccess, onError)` – для GET-запросов на `url`. При успешном запросе вызывается `onSuccess(result)`, при неуспешном: `onError()`.
- `iframePost(url, data, onSuccess, onError)` – для POST-запросов на `url`. Значением `data` должен быть объект `ключ:значение` для пересылаемых данных, он конвертируется в поля формы.

Пример в действии, возвращающий дату сервера при GET и разницу между датами клиента и сервера при POST:

Прямой вызов функции внешнего окна из ифрейма отлично работает, потому что они с одного домена. Если с разных, то нужны дополнительные действия, например:

- В IE8+ есть интерфейс postMessage для общения между окнами с разных доменов.
- В любых, даже самых старых IE, можно обмениваться данными через `window.name`. Эта переменная хранит «имя» окна или фрейма, которое не меняется при перезагрузке страницы.

  Поэтому если мы сделали `POST` в `<iframe>` на другой домен и он поставил `window.name = "Вася"`, а затем сделал редирект на основной домен, то эти данные станут доступны внешней странице.
- Также в совсем старых IE можно обмениваться данными через хеш, то есть фрагмент URL после `#`. Его изменение доступно между ифреймами с разных доменов и не приводит к перезагрузке страницы. Таким образом они могут передавать данные друг другу. Есть готовые библиотеки, которые реализуют этот подход, например Porthole.

IFRAME для COMET

Бесконечный IFRAME – самый старый способ организации COMET. Когда-то он был основой AJAX-приложений, а сейчас – используется лишь в случаях, когда браузер не поддерживает современный стандарт WebSocket, то есть для IE9-.

Этот способ основан на том, что браузер читает страницу последовательно и обрабатывает все новые теги по мере того, как сервер их присылает.

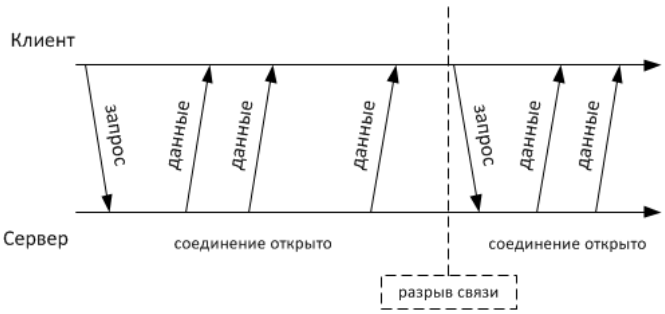
Классическая реализация – это когда клиент создает невидимый IFRAME, ведущий на служебный URL. Сервер, получив соединение на этот URL, не закрывает его, а время от времени присылает блоки сообщений `<script>…javascript…</script>`. Появившийся в IFRAME'е javascript тут же выполняется браузером, передавая информацию на основную страницу.

Таким образом, для передачи данных используется «бесконечный» ифрейм, через который сервер присылает все новые данные.

Схема работы:

1. Создаётся `<iframe src="COMET_URL">`, по адресу `COMET_URL` расположен сервер.
2. Сервер выдаёт начало («шапку») документа и останавливается, оставляя соединение активным.
3. Когда сервер хочет что-то отправить – он пишет в соединение `<script>parent.onMessage(сообщение)</script>` Браузер тут же выполняет этот скрипт – так сообщение приходит на клиент.
4. Ифрейм, в теории, грузится бесконечно. Его завершение означает обрыв канала связи. Его можно поймать по `iframe.onload` и заново открыть соединение (создать новый `iframe`).

Также ифрейм можно пересоздавать время от времени, для очистки памяти от старых сообщений.



Ифрейм при этом работает только на получение данных с сервера, как альтернатива Server Sent Events. Для запросов используется обычный `XMLHttpRequest`.

Обход проблем с IE

Такое использование ифреймов является хаком. Поэтому есть ряд проблем:

1. Показывается индикатор загрузки, «курсор-часики».
2. При POST в `<iframe>` раздаётся звук «клика».
3. Браузер буферизует начало страницы.

Мы должны эти проблемы решить, прежде всего, в IE, поскольку в других браузерах есть WebSocket и Server Sent Events.

Проще всего решить последнюю – IE не начинает обработку страницы, пока она не загрузится до определенного размера.

Поэтому в таком `IFRAME` первые несколько сообщений задержатся:

```
<!DOCTYPE HTML>
<html>
  <body>
  <script>parent.onMessage("привет");</script>
  <script>parent.onMessage("от сервера");</script>
  ...
```

Решение – забить начало ифрейма чем-нибудь, поставить, например, килобайт пробелов в начале:

```
<!DOCTYPE HTML>
<html>

<body>
  ******* 1 килобайт пробелов, а потом уже сообщения ******
  <script>
    parent.onMessage("привет");
  </script>
  <script>
    parent.onMessage("от сервера");
  </script>
  ...
```

Для решения проблемы с индикацией загрузки и клика мы можем использовать безопасный ActiveX-объект `htmlfile`. IE не требует разрешений на его создание. Фактически, это независимый HTML-документ.

Оказывается, если `iframe` создать в нём, то никакой анимации и звуков не будет.

Итак, схема:

1. Основное окно `main` создаёт вспомогательный объект: `new ActiveXObject("htmlfile")`. Это HTML-документ со своим `window`, похоже на встроенный `iframe`.
2. В `htmlfile` записывается `iframe`.
3. Цепочка общения: основное окно – `htmlfile` – ифрейм.

iframeActiveXGet

На самом деле всё еще проще, если посмотреть на код:

Метод `iframeActiveXGet` по существу идентичен обычному `iframeGet`, которое мы рассмотрели. Единственное отличие – вместо `createIframe` используется особый метод `createActiveXFrame`:

```
function iframeActiveXGet(url, onSuccess, onError) {

  var iframeOk = false;

  var iframeName = Math.random();
  var iframe = createActiveXFrame(iframeName, url);

  CallbackRegistry[iframeName] = function(data) {
    iframeOk = true;
    onSuccess(data);
  }

  iframe.onload = function() {
    iframe.parentNode.removeChild(iframe); // очистка
    delete CallbackRegistry[iframeName];
    if (!iframeOk) onError(); // если коллбэк не вызвался - что-то не так
  }

}
```

createActiveXFrame

В этой функции творится вся IE-магия:

```
function createActiveXFrame(name, src) {
  // (1)
  var htmlfile = window.htmlfile;
  if (!htmlfile) {
    htmlfile = window.htmlfile = new ActiveXObject("htmlfile");
    htmlfile.open();
    // (2)
    htmlfile.write("<html><body></body></html>");
    htmlfile.close();
    // (3)
    htmlfile.parentWindow.CallbackRegistry = CallbackRegistry;
  }

  // (4)
  src = src || 'javascript:false';
  htmlfile.body.insertAdjacentHTML('beforeEnd',
    "<iframe name='" + name + "' src='" + src + "'></iframe>");
  return htmlfile.body.lastChild;
}
```

1. Вспомогательный объект `htmlfile` будет один и он будет глобальным. Можно и спрятать переменную в замыкании. Смысл в том, что в один `htmlfile` можно записать много ифреймов, так что не будем множить сущности и занимать ими лишнюю память.
2. В `htmlfile` можно записать любой текст и, при необходимости, через `document.write('<script>...<\/script>)`. Здесь мы делаем пустой документ.

3. Когда загрузится `iframe`, он сделает вызов:

```
<script>
  parent.CallbackRegistry[window.name](объект с данными);
</script>
```

   Здесь `parent'ом для iframe'a` будет `htmlfile`, т.е. `CallbackRegistry` будет искаться среди переменных соответствующего ему окна, а вовсе не верхнего `window`.

   Окно для `htmlfile` доступно как `htmlfile.parentWindow`, копируем в него ссылку на реестр коллбэков `CallbackRegistry`. Теперь ифрейм его найдёт.

4. Далее вставляем ифрейм в документ. В старых `IE` нельзя поменять `name` ифрейму через DOM, поэтому вставляем строкой через `insertAdjacentHTML`.

Пример в действии (только IE):

Запрос, который происходит, полностью незаметен.

Метод POST делается аналогично, только форму нужно добавлять не в основное окно, а в `htmlfile`, через вызов `htmlfile.appendChild`. В остальном – всё так же, как и при обычной отправке через ифрейм.

Впрочем, для COMET нужен именно GET.

Можно и сочетать эти способы: если есть ActiveX: `if ("ActiveXObject" in window)` – используем методы для IE, описанные выше, а иначе – обычные методы.

Вот мини-приложение с сервером на Node.JS, непрерывно получающее текущее время с сервера через `<iframe>`, сочетающее эти подходы:

Ещё раз заметим, что обычно такое сочетание не нужно, так как если не IE9-, то можно использовать более современные средства для COMET.

Итого

- Iframe позволяет делать «AJAX»-запросы и хитро обходить кросс-доменные ограничения в IE7-. Обычно для этого используют либо `window.name` с редиректом, либо хеш с библиотекой типа Porthole.
- В IE9- iframe можно использовать для COMET. В IE10 уже есть WebSocket.

Существует ряд уже готовых клиент-серверных библиотек, которые реализуют AJAX/COMET, в том числе и через iframe, мы рассмотрим их позже. Поэтому совсем не обязательно делать «с нуля». Хотя, как можно видеть из главы, это совсем несложно.

Атака CSRF

Нельзя говорить про AJAX и не упомянуть про важнейшую деталь его реализации – защиту от CSRF-атак.

CSRF (Cross-Site Request Forgery, также XSRF) – опаснейшая атака, которая приводит к тому, что хакер может выполнить на неподготовленном сайте массу различных действий от имени других, зарегистрированных посетителей.

Какие это действия – отправка ли сообщений, перевод денег со счёта на счёт или смена паролей – зависят от сайта, но в любом случае эта атака входит в образовательный минимум веб-разработчика.

Злая форма

«Классический» сценарий атаки таков:

- Вася является залогиненным на сайт, допустим, `mail.com`. У него есть сессия в куках.
- Вася попал на «злую страницу», например хакер пригласил его сделать это письмом или как-то иначе.
- На злой странице находится форма такого вида:

```
<form action="http://mail.com/send" method="POST">
  <input type="hidden" name="message" value="Сообщение">
  ...
</form>
```

- При заходе на злую страницу JavaScript вызывает `form.submit`, отправляя таким образом форму на `mail.com`.
- Сайт `mail.com` проверяет куки, видит, что посетитель авторизован и обрабатывает форму. В данном примере форма предполагает посылку сообщения.

Итог атаки – Вася, зайдя на злую страницу, ненароком отправил письмо от своего имени. Содержимое письма сформировано хакером.

Защита

В примере выше атака использовала слабое звено авторизации.

Куки позволяют сайту **mail.com** проверить, что пришёл именно Вася, но ничего не говорят про данные, которые он отправляет.

Иначе говоря, куки не гарантируют, что форму создал именно Вася. Они только удостоверяют личность, но не данные.

Типичный способ защиты сайтов – это «секретный ключ» (`secret`), специальное значение, которое генерируется случайным образом и сохраняется в сессии посетителя. Его знает только сервер, посетителю мы его даже не будем показывать.

Затем на основе ключа генерируется «токен» (`token`). Токен делается так, чтобы с одной стороны он был отличен от ключа, в частности, может быть много токенов для одного ключа, с другой – чтобы было легко проверить по токену, сгенерирован ли он на основе данного ключа или нет.

Для каждого токена нужно дополнительное случайное значение, которое называют «соль» `salt` .

Формула вычисления токена:

```
token = salt + ":" + MD5(salt + ":" + secret)
```

Например:

1. В сессии хранится `secret="abcdef"` , это значение создаётся один раз.
2. Для нового токена сгенерируем `salt` , например пусть `salt="1234"` .
3. `token = "1234" + ":" + MD5("1234" + ":" + "abcdef") = "1234:5ad02792a3285252e524ccadeeda3401"` .

Это значение – с одной стороны, случайное, с другой – имея такой `token` , мы можем взять его первую часть `1234` в качестве `salt` и, зная `secret` , проверить по формуле, верно ли он вычислен.

Не зная `secret` , невозможно сгенерировать token, который сервер воспримет как правильный.

Далее, токен добавляется в качестве скрытого поля к каждой форме, генерируемой на сервере.

То есть, «честная» форма для отсылки сообщений, созданная на `http://mail.com` , будет выглядеть так:

```
<form action="http://mail.com/send" method="POST">
  <input type="hidden" name="csrf" value="1234:5ad02792a3285252e524ccadeeda3401">
  <textarea name="message">
    ...
  </textarea>
</form>
```

При её отправке сервер проверит поле `csrf` , удостоверится в правильности токена, и лишь после этого отошлёт сообщение.

«Злая страница» при всём желании не сможет сгенерировать подобную форму, так как не владеет `secret` , и токен будет неверным.

Такой токен также называют «подписью» формы, которая удостоверяет, что форма сгенерирована именно на сервере.

Подпись с полями формы

Эта подпись говорит о том, что автор формы – сервер, но ничего не гарантирует относительно её содержания.

Есть ситуации, когда мы хотим быть уверены, что некоторые из полей формы посетитель не изменил самовольно. Тогда мы можем включить в MD5 для формулы токена эти поля, например:

```
token = salt + ":" + MD5(salt + ":" + secret + ":" + fields.money)
```

При отправке формы сервер проверит подпись, подставив в неё известный ему `secret` и присланное значение `fields.money` . При несовпадении либо `secret` не тот (хакер), либо `fields.money` изменено.

Токен и AJAX

Теперь перейдём к AJAX-запросам.

Что если посылка сообщений в нашем интерфейсе реализуется через XMLHttpRequest?

Как и в случае с формой, мы должны «подписать» запрос токеном, чтобы гарантировать, что его содержимое прислано на сервер именно интерфейсом сайта, а не «злой страницей».

Здесь возможны варианты, самый простой – это дополнительная кука.

1. При авторизации сервер устанавливает куку с именем `CSRF-TOKEN` , и пишет в неё токен.
2. Код, осуществляющий XMLHttpRequest, получает куку и ставит заголовок `X-CSRF-TOKEN` с ней:

```
var request = new XMLHttpRequest();

var csrfCookie = document.cookie.match(/CSRF-TOKEN=([\w-]+)/);
if (csrfCookie) {
  request.setRequestHeader("X-CSRF-TOKEN", csrfCookie[1]);
}
```

3. Сервер проверяет, есть ли заголовок и содержит ли он правильный токен.

Защита действует потому, что прочитать куку может только JavaScript с того же домена. «Злая страница» не сможет «переложить» куку в заголовок.

Если нужно сделать не XMLHttpRequest, а, к примеру, динамически сгенерировать форму из JavaScript – она также подписывается аналогичным образом, скрытое поле или дополнительный URL-параметр генерируется по куке.

Итого

- CSRF-атака – это когда «злая страница» отправляет форму или запрос на сайт, где посетитель, предположительно, залогинен.

  Если сайт проверяет только куки, то он такую форму принимает. А делать это не следует, так как её сгенерировал злой хакер.
- Для защиты от атаки формы, которые генерирует `mail.com`, подписываются специальным токеном. Можно не все формы, а только те, которые осуществляют действия от имени посетителя, то есть могут служить объектом атаки.
- Для подписи XMLHttpRequest токен дополнительно записывается в куку. Тогда JavaScript с домена `mail.com` сможет прочитать её и добавить в заголовок, а сервер – проверить, что заголовок есть и содержит корректный токен.
- Динамически сгенерированные формы подписываются аналогично: токен из куки добавляется как URL-параметр или дополнительное поле.

Метод fetch: замена XMLHttpRequest

Метод fetch – это `XMLHttpRequest` нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на промисах.

Поддержка в браузерах пока не очень распространена, но есть полифилл и не один.

Синтаксис

Синтаксис метода `fetch`:

```
let promise = fetch(url[, options]);
```

- `url` – URL, на который сделать запрос,
- `options` – необязательный объект с настройками запроса.

Свойства `options`:

- `method` – метод запроса,
- `headers` – заголовки запроса (объект),
- `body` – тело запроса: `FormData`, `Blob`, строка и т.п.
- `mode` – одно из: «same-origin», «no-cors», «cors», указывает, в каком режиме кросс-доменности предполагается делать запрос.
- `credentials` – одно из: «omit», «same-origin», «include», указывает, пересылать ли куки и заголовки авторизации вместе с запросом.
- `cache` – одно из «default», «no-store», «reload», «no-cache», «force-cache», «only-if-cached», указывает, как кешировать запрос.
- `redirect` – можно поставить «follow» для обычного поведения при коде 30x (следовать редиректу) или «error» для интерпретации редиректа как ошибки.

Как видно, всевозможных настроек здесь больше, чем в `XMLHttpRequest`. Вместе с тем, надо понимать, что если мы используем полифилл, то ничего более гибкого, чем оригинальный `XMLHttpRequest` мы из этого не получим.

Разве что, `fetch`, возможно, будет удобнее пользоваться.

Использование

При вызове `fetch` возвращает промис, который, когда получен ответ, выполняет коллбэки с объектом Response или с ошибкой, если запрос не удался.

Пример использования:

```
'use strict';

fetch('/article/fetch/user.json')
  .then(function(response) {
    alert(response.headers.get('Content-Type')); // application/json; charset=utf-8
    alert(response.status); // 200

    return response.json();
   })
  .then(function(user) {
    alert(user.name); // iliakan
  })
  .catch( alert );
```

Объект `response` кроме доступа к заголовкам `headers`, статусу `status` и некоторым другим полям ответа, даёт возможность прочитать его тело, в желаемом формате.

Варианты описаны в спецификации Body, они включают в себя:

- `response.arrayBuffer()`
- `response.blob()`
- `response.formData()`
- `response.json()`
- `response.text()`

Соответствующий вызов возвращает промис, который, когда ответ будет получен, вызовет коллбэк с результатом.

В примере выше мы можем в первом `.then` проанализировать ответ и, если он нас устроит – вернуть промис с нужным форматом. Следующий `.then` уже будет содержать полный ответ сервера.

Больше примеров вы можете найти в описании полифилла для fetch.

Итого

Метод `fetch` – уже сейчас удобная альтернатива `XMLHttpRequest` для тех, кто не хочет ждать и любит промисы.

Детальное описание этого метода есть в стандарте Fetch, а простейшие примеры запросов – в описании к полифиллу.

Таблица транспортов и их возможностей

Здесь мы подведём итог раздела, сравним транспорты и их возможности.

Способы опроса сервера

Основные способы опроса сервера:

1. Частые опросы – регулярно к серверу отправляется запрос за данными. Сервер тут же отвечает на него, возвращая данные, если они есть. Если нет – получается, что запрос был зря.

   Этот способ очень лёгок в реализации, но приводит к большому количеству лишних запросов, поэтому мы его далее не рассматриваем.

2. Длинные опросы – к серверу отправляется запрос за данными. Сервер не отвечает на него, пока данные не появятся. Когда данные появились – ответ с ними отправляется в браузер, и тот тут же делает новый запрос.

   Способ хорош, пока сообщений не слишком много. В идеальном случае соединение почти всё время висит открытым, лишь иногда сервер отвечает на него, доставляя данные в браузер.

   Также удобен в реализации, но даёт большое количество висящих соединений на сервере. Не все сервера хорошо поддерживают это. Например, `Apache` будет есть очень много памяти.

3. Потоковое соединение – открыто соединение к серверу, и через него непрерывно поступают данные.

Таблица транспортов

Основные характеристики всех транспортов, которые мы обсуждали в деталях, собраны в этой таблице.

Они были детально рассмотрены в предыдущих главах раздела.

	XMLHttpRequest	**IFRAME**	**SCRIPT**	**EventSource**	**WebSocket**
Кросс-доменность	да, кроме IE9-[x1]	да[i1]	да	да	да
Методы	Любые	GET / POST	GET	GET	Свой протокол
COMET	Длинные опросы[x2]	Непрерывное соединение	Длинные опросы	Непрерывное соединение	Непрерывное соединение в обе стороны
Поддержка	Все браузеры, ограничения в IE9-[x3]	Все браузеры	Все браузеры	Кроме IE	IE 10, FF11, Chrome 16, Safari 6, Opera 12.5[w1]

Пояснения:

`XMLHttpRequest`

- В IE8-9 поддерживаются кросс-доменные GET/POST запросы с ограничениями через `XDomainRequest`.
- Можно говорить об ограниченной поддержке непрерывного соединения через `onprogress`, но это событие вызывается не чаще чем в 50ms и не гарантирует получение полного пакета данных. Например, сервер может записать "Привет!", а событие вызовется один раз, когда браузер получил "При". Поэтому наладить обмен пакетами сообщений с его помощью затруднительно.
- Многие возможности современного стандарта включены в IE лишь с версии 10.

`IFRAME`

- Во всех современных браузерах и IE8 кросс-доменность обеспечивает `postMessage`. В более старых браузерах возможны решения через `window.name` и хэш.

`WebSocket`

- Имеется в виду поддержка окончательной редакции протокола RFC 6455. Более старые браузеры могут поддерживать черновики протокола. IE9- не поддерживает `WebSocket`.

Существует также нестандартный транспорт, не рассмотренный здесь:

- XMLHttpRequest с флагом `multipart`, только для Firefox.

  При указании свойства `xhr.multipart = true` и специального multipart-формата ответа сервера, Firefox инициирует `onload` при получении очередной части ответа. Ответ может состоять из любого количества частей, досылаемых по инициативе сервера. Мы не рассматривали его, так как Firefox поддерживает другие, более кросс-браузерные и стандартные транспорты.

В современных браузерах поддерживается новый метод fetch, в качестве замены XMLHttpRequest (полифилл).

Анимация

CSS анимации. Контроль над ними из JavaScript. Анимации на чистом JavaScript.

Кривые Безье

Кривые Безье используются в компьютерной графике для рисования плавных изгибов, в CSS-анимации и много где ещё.

Несмотря на «умное» название – это очень простая штука.

В принципе, можно создавать анимацию и без знания кривых Безье, но стоит один раз изучить эту тему хотя бы для того, чтобы в дальнейшем с комфортом пользоваться этим замечательным инструментом. Тем более что в мире векторной графики и продвинутых анимаций без них никак.

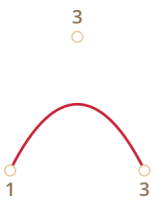
Виды кривых Безье

Кривая Безье задаётся опорными точками.

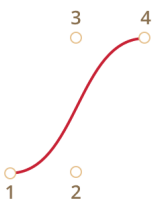
Их может быть две, три, четыре или больше. Например:

По двум точкам:

По трём точкам:

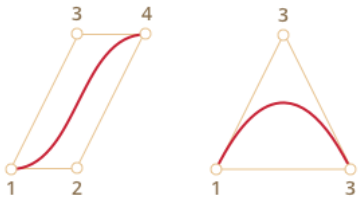


По четырём точкам:



Если вы посмотрите внимательно на эти кривые, то «на глазок» заметите:

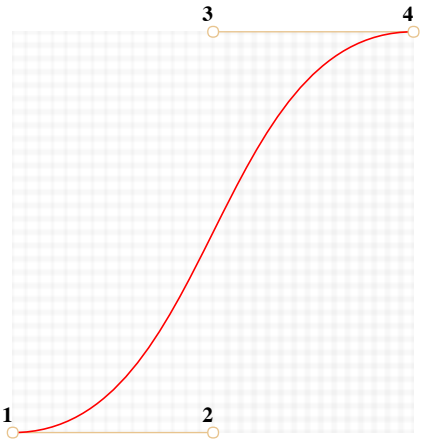
1. Точки не всегда на кривой. Это совершенно нормально, как именно строится кривая мы рассмотрим чуть позже.
2. Степень кривой равна числу точек минус один. Для двух точек – это линейная кривая (т.е. прямая), для трёх точек – квадратическая кривая (парабола), для четырёх – кубическая.
3. Кривая всегда находится внутри выпуклой оболочки, образованной опорными точками:



Благодаря последнему свойству в компьютерной графике можно оптимизировать проверку пересечений двух кривых. Если их выпуклые оболочки не пересекаются, то и кривые тоже не пересекутся.

Основная ценность кривых Безье для рисования – в том, что, двигая точки, кривую можно менять, причём кривая при этом *меняется интуитивно понятным образом*.

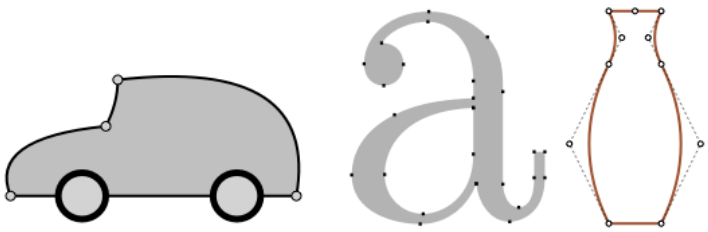
Попробуйте двигать точки мышью в примере ниже:



Как можно заметить, кривая натянута по касательным 1 → 2 и 3 → 4.

После небольшой практики становится понятно, как расположить точки, чтобы получить нужную форму. А, соединяя несколько кривых, можно получить практически что угодно.

Вот некоторые примеры:



Математика

У кривых Безье есть математическая формула.

Как мы увидим далее, для пользования кривыми Безье знать её нет особенной необходимости, но для полноты картины – вот она.

Координаты кривой описываются в зависимости от параметра $t \in [0,1]$

- Для двух точек:

  $P = (1-t)P_1 + tP_2$

- Для трёх точек:

  $P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$

- Для четырёх точек:

  $P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$

Вместо P_i нужно подставить координаты i-й опорной точки (x_i, y_i).

Эти уравнения векторные, то есть для каждой из координат:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$

- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

Вместо x_1, y_1, x_2, y_2, x_3, y_3 подставляются координаты трёх опорных точек, и в то время как t пробегает множество от 0 до 1, соответствующие значения (x, y) как раз и образуют кривую.

Впрочем, это чересчур наукообразно, не очень понятно, почему кривые именно такие, и как зависят от опорных точек. С этим нам поможет разобраться другой, более наглядный алгоритм.

Рисование «де Кастельжо»

Метод де Кастельжо идентичен математическому определению кривой и наглядно показывает, как она строится.

Посмотрим его на примере трёх точек (точки можно двигать). Нажатие на кнопку «play» запустит демонстрацию.

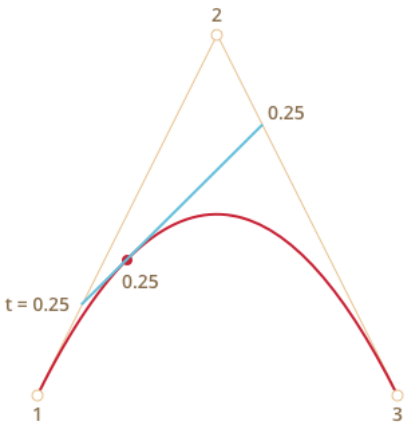
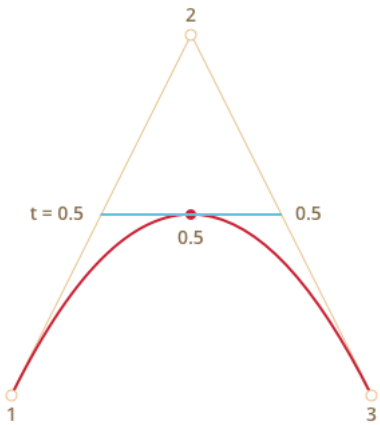
Алгоритм построения кривой по «методу де Кастельжо»:

1. Рисуем опорные точки. В примере выше это 1, 2, 3.
2. Строятся отрезки между опорными точками 1 → 2 → 3. На рисунке выше они коричневые.
3. Параметр t пробегает значения от 0 до 1. В примере выше использован шаг 0.05, т.е. в цикле $0, 0.05, 0.1, 0.15, \ldots 0.95, 1$.

   Для каждого из этих значений t:

   - На каждом из коричневых отрезков берётся точка, находящаяся от начала на расстоянии от 0 до t пропорционально длине. Так как коричневых отрезков – два, то и точек две штуки.

     Например, при $t=0$ – точки будут в начале, при $t=0.25$ – на расстоянии в 25% от начала отрезка, при $t=0.5$ – 50%(на середине), при $t=1$ – в конце отрезков.
   - Эти точки соединяются. На рисунке ниже соединяющий их отрезок изображён синим.

При **t=0.25**	При **t=0.5**
	

4. На получившемся отрезке берётся точка на расстоянии, соответствующем t. То есть, для $t=0.25$ (первый рисунок) получаем точку в конце первой четверти отрезка, для $t=0.5$ (второй рисунок) – в середине отрезка. На рисунках выше эта точка отмечена красным.
5. По мере того как t пробегает последовательность от 0 до 1, каждое значение t добавляет к красной кривой точку. Совокупность таких точек для всех значений t образуют кривую Безье.

Это был процесс для построения по трём точкам. Но то же самое происходит и с четырьмя точками.

Демо для четырёх точек (точки можно двигать):

Алгоритм:

- Точки по порядку соединяются отрезками: 1 → 2, 2 → 3, 3 → 4. Получается три коричневых отрезка.
- На отрезках берутся точки, соответствующие текущему `t`, соединяются. Получается два зелёных отрезка.
- На этих отрезках берутся точки, соответствующие текущему `t`, соединяются. Получается один синий отрезок.
- На синем отрезке берётся точка, соответствующая текущему `t`. При запуске примера выше она красная.
- Эти точки описывают кривую.

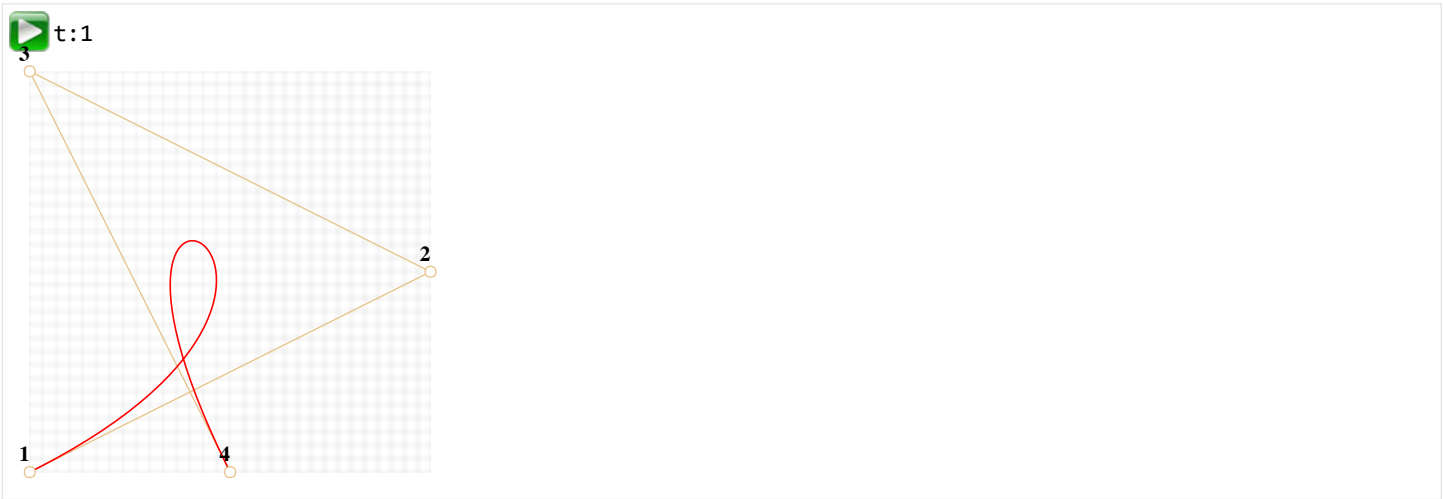
Этот алгоритм рекурсивен. Для каждого `t` из интервала от `0` до `1` по этому правилу, соединяя точки на соответствующем расстоянии, из 4 отрезков делается 3, затем из 3 так же делается 2, затем из 2 отрезков – точка, описывающая кривую для данного значения `t`.

Нажмите на кнопку «play» в примере выше, чтобы увидеть это в действии.

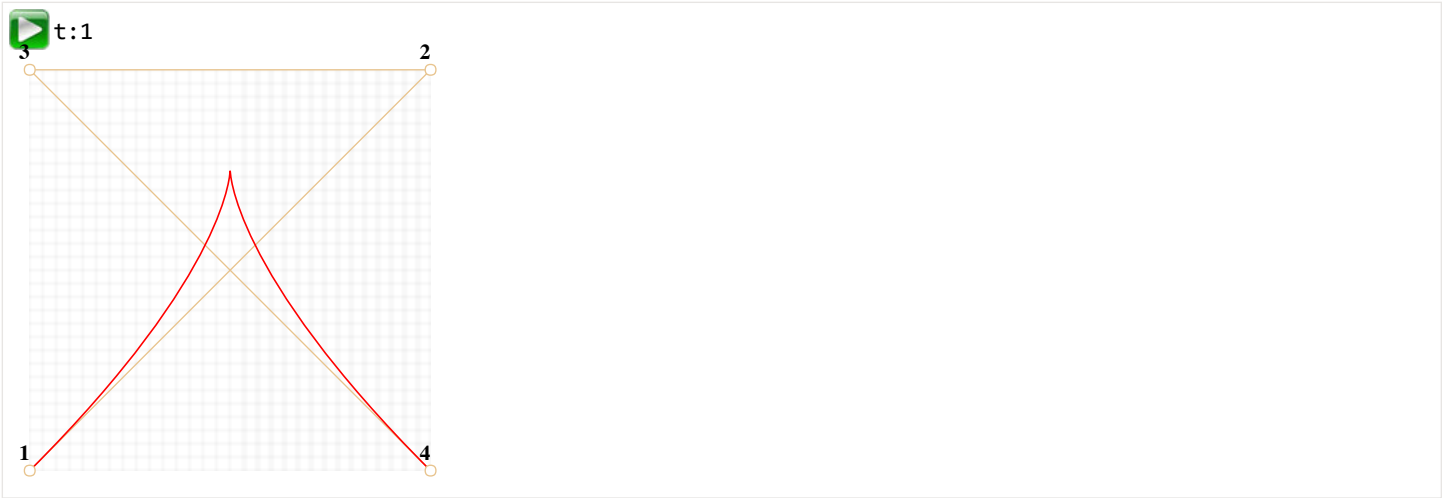
Ещё примеры кривых:

С другими точками:

Петелька:



Пример негладкой кривой Безье:



Так как алгоритм рекурсивен, то аналогичным образом могут быть построены кривые Безье и более высокого порядка: по пяти точкам, шести и так далее. Однако на практике они менее полезны. Обычно используются 2-3 точки, а для сложных линий несколько кривых соединяются. Это гораздо проще с точки зрения поддержки и расчётов.

Как провести кривую Безье *через* нужные точки?

В задаче построения кривой Безье используются «опорные точки». Они, как можно видеть из примеров выше, не лежат на кривой. Точнее говоря, только первая и последняя лежат на кривой, а промежуточные – нет.

Иногда возникает другая задача: провести кривую именно *через нужные точки*, чтобы все они лежали на некой плавной кривой, удовлетворяющей определённым требованиям. Такая задача называется интерполяцией, и здесь мы её не рассматриваем.

Существуют математические формулы для таких построений, например многочлен Лагранжа.

Как правило, в компьютерной графике для построения плавных кривых, проходящих через несколько точек, используют кубические кривые, плавно переходящие одна в другую. Это называется интерполяция сплайнами.

Итого

Кривые Безье задаются опорными точками.

Мы рассмотрели два определения кривых:

1. Через математическую формулу.
2. Через процесс построения де Кастельжо.

Их удобство в том, что:

- Можно легко нарисовать плавные линии вручную, передвигая точки мышкой.
- Более сложные изгибы и линии можно составить, если соединить несколько кривых Безье.

Применение:

- В компьютерной графике, моделировании, в графических редакторах. Шрифты описываются с помощью кривых Безье.
- В веб-разработке – для графики на Canvas или в формате SVG. Кстати, все живые примеры выше написаны на SVG. Фактически, это один SVG-документ, к которому точки передаются параметрами. Вы можете открыть его в отдельном окне и посмотреть исходник: demo.svg.
- В CSS-анимации, для задания траектории или скорости передвижения.

CSS-анимации

Все современные браузеры, кроме IE9- поддерживают CSS transitions и CSS animations, которые позволяют реализовать анимацию средствами CSS, без привлечения JavaScript.

Однако, как мы увидим далее, для более тонкого контроля анимации JavaScript вовсе не будет лишним.

CSS transitions

Идея проста. Мы указываем, что некоторое свойство будет анимироваться при помощи специальных CSS-правил. Далее, при изменении этого свойства, браузер сам обработает анимацию.

Например, CSS, представленный ниже, 3 секунды анимирует свойство `background-color`.

```
.animated {
  transition-property: background-color;
  transition-duration: 3s;
}
```

Теперь любое изменение фонового цвета будет анимироваться в течение 3х секунд.

При клике на эту кнопку происходит анимация её фона:

```
<button id="color">Кликни меня</button>

<style>
  #color {
    transition-property: background-color;
    transition-duration: 3s;
  }
</style>

<script>
  color.onclick = function() {
    this.style.backgroundColor = 'red';
  }
</script>
```

Кликни меня

Есть всего 5 свойств, задающих анимацию:

- `transition-property`
- `transition-duration`

- `transition-timing-function`
- `transition-delay`

Далее мы изучим их все, пока лишь заметим, что общее свойство `transition` может перечислять их все, в порядке: `property duration timing-function delay`, а также задавать анимацию нескольких свойств сразу.

Например, при клике на эту кнопку анимируются одновременно цвет и размер шрифта:

```
<button id="growing">Кликни меня</button>

<style>
#growing {
  transition: font-size 3s, color 2s;
}
</style>

<script>
growing.onclick = function() {
  this.style.fontSize='36px';
  this.style.color='red';
}
</script>
```

Кликни меня

Далее мы рассмотрим свойства анимации по отдельности.

transition-property

Список свойств, которые будут анимироваться, например: `left`, `margin-left`, `height`, `color`.

Анимировать можно не все свойства, но многие. Значение `all` означает «анимировать все свойства».

transition-duration

Продолжительность анимации, задаётся в формате CSS time, то есть в секундах `s` или `ms`.

transition-delay

Задержка до анимации. Например, если `transition-delay: 1s`, то анимация начнётся через 1 секунду после смены свойства.

Возможны отрицательные значения, при этом анимация начнётся с середины.

Например, вот анимация цифр от `0` до `9`:

Она осуществляется сменой `margin-left` у элемента с цифрами, примерно так:

```
#stripe.animate {
  margin-left: -174px;
  transition-property: margin-left;
  transition-duration: 9s;
}
```

В примере выше JavaScript просто добавляет элементу класс – и анимация стартует:

```
digit.classList.add('animate');
```

Можно стартовать её «с середины», с нужной цифры, например соответствующей текущей секунде, при помощи отрицательного `transition-delay`.

В примере ниже при клике на цифру она начнёт двигаться с текущей секунды:

В JavaScript это делается дополнительной строкой:

```
stripe.onclick = function() {
  var sec = new Date().getSeconds() % 10;
  // например, значение -3s начнёт анимацию с 3-й секунды
  stripe.style.transitionDelay = '-' + sec + 's';
  stripe.classList.add('animate');
};
```

transition-timing-function

Временна́я функция, которая задаёт, как процесс анимации будет распределён во времени, например начнётся ли анимация медленно, чтобы потом ускориться или наоборот.

Самое сложное, но при небольшом изучении – вполне очевидное свойство.

У него есть два основных вида значения: кривая Безье и по шагам. Начнём с первого.

Кривая Безье

В качестве временной функции можно выбрать любую кривую Безье с 4 опорными точками, удовлетворяющую условиям:

1. Начальная точка `(0,0)`.
2. Конечная точка `(1,1)`.
3. Для промежуточных точек значения `x` должны быть в интервале `0..1`, `y` – любыми.

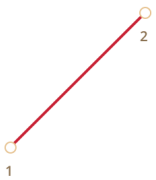
Синтаксис для задания кривой Безье в CSS: `cubic-bezier(x2, y2, x3, y3)`. В нём указываются координаты только двух точек: второй и третьей, так как первая и последняя фиксированы.

Она указывает, как быстро развивается процесс анимации во времени.

- По оси `x` идёт время: `0` – начальный момент, `1` – конец времени `transition-duration`.
- По оси `y` – завершённость процесса: `0` – начальное значение анимируемого свойства, `1` – конечное.

Самый простой вариант – это когда процесс развивается равномерно, «линейно» по времени. Это можно задать кривой Безье `cubic-bezier(0, 0, 1, 1)`.

График этой «кривой» таков:



...Как видно, это просто прямая. По мере того, как проходит время `x`, завершённость анимации `y` равномерно приближается от `0` к `1`.

Поезд в примере ниже с постоянной скоростью «едет» слева направо, используя такую временную функцию:

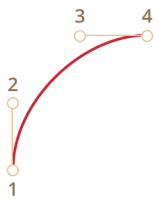
CSS для анимации:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
  /* JavaScript ставит значение left: 450px */
}
```

Как нам показать, что поезд тормозит?

Для этого используем кривую Безье: `cubic-bezier(0.0, 0.5, 0.5 ,1.0)`.

График этой кривой:



Как видно, процесс вначале развивается быстро – кривая резко идёт вверх, а затем всё медленнее, медленнее.

Вы можете увидеть эту временную функцию в действии, кликнув на поезд:

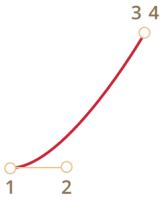
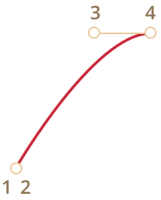
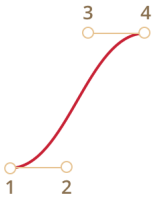
CSS для анимации:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, .5, .5, 1);
  /* JavaScript ставит значение left: 450px */
}
```

Существует несколько стандартных обозначений кривых: `linear`, `ease`, `ease-in`, `ease-out` и `ease-in-out`.

Значение `linear` – это прямая, мы её уже видели.

Остальные кривые являются короткой записью следующих `cubic-bezier`:

ease *	ease-in	ease-out	ease-in-out
(0.25, 0.1, 0.25, 1.0)	(0.42, 0, 1.0, 1.0)	(0, 0, 0.58, 1.0)	(0.42, 0, 0.58, 1.0)
			

`*` – По умолчанию, если никакой временной функции не указано, используется `ease`.

Кривая Безье может заставить анимацию «выпрыгивать» за пределы диапазона.

Допустимо указывать для кривой Безье как отрицательные `y`, так и сколь угодно большие. При этом кривая Безье будет также по `y` выскакивать за пределы диапазона `0..1`, представляющего собой начало-конец значения.

В примере ниже CSS-код анимации таков:

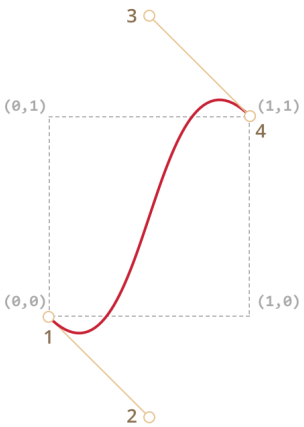
```
.train {
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
  /* JavaScript поменяет left на 400px */
}
```

Свойство `left` должно меняться от `100px` до `400px`.

Однако, если кликнуть на поезд, то мы увидим, что:

- Он едет сначала назад, то есть `left` становится меньше `100px`.
- Затем вперёд, причём выезжает за назначенные `400px`.
- А затем опять назад – до `400px`.

Почему так происходит – отлично видно, если взглянуть на кривую Безье с указанными опорными точками:



Мы вынесли координату `y` для второй опорной точки на 1 ниже нуля, а для третьей опорной точки – на 1 выше единицы, поэтому и кривая вышла за границы «обычного» квадрата. Её значения по `y` вышли из стандартного диапазона `0..1`.

Как мы помним, значению `y = 0` соответствует «нулевое» положение анимации, а `y = 1` – конечное. Получается, что значения `y<0` двинули поезд назад, меньше исходного `left`, а значения `y>1` – больше итогового `left`.

Это, конечно, «мягкий» вариант. Если поставить значения `y` порядка `-99`, `99`, то поезд будет куда более сильно выпрыгивать за диапазон.

Итак, кривая Безье позволяет задавать «плавное»" течение анимации. Подобрать кривую Безье вручную можно на сайте http://cubic-bezier.com/.

Шаги steps

Временная функция `steps(количество шагов[, start/end])` позволяет разбить анимацию на чёткое количество шагов.

Проще всего это увидеть на примере. Выше мы видели плавную анимацию цифр от `0` до `9` при помощи смены `margin-left` у элемента, содержащего `0123456789`.

Чтобы цифры сдвигались не плавно, а шли чётко и раздельно, одна за другой – мы разобьём анимацию на 9 шагов:

```
#stripe.animate {
  margin-left: -174px;
```

```
  transition: margin-left 9s steps(9, start);
}
```

В действии `step(9, start)`:

Первый аргумент `steps` – количество шагов, то есть изменение `margin-left` разделить на 9 частей, получается примерно по `19px`. На то же количество частей делится и временной интервал, то есть по `1s`.

`start` – означает, что при начале анимации нужно сразу применить первое изменение. Это проявляется тем, что при нажатии на цифру она меняется на `1` (первое изменение `margin-left`) мгновенно, а затем в начале каждой следующей секунды.

То есть, процесс развивается так:

- `0s` – `-19px` (первое изменение в начале 1-й секунды, сразу при нажатии)
- `1s` – `-38px`
- ...
- `8s` – `-174px`
- (на протяжении последней секунды видно окончательное значение).

Альтернативное значение `end` означало бы, что изменения нужно применять не в начале, а в конце каждой секунды, то есть так:

- `0s` – `0`
- `1s` – `-19px` (первое изменение в конце 1-й секунды)
- `2s` – `-38px`
- ...
- `9s` – `-174px`

В действии `step(9, end)`:

Также есть сокращённые значения:

- `step-start` – то же, что `steps(1, start)`, то есть завершить анимацию в 1 шаг сразу.
- `step-end` – то же, что `steps(1, end)`, то есть завершить анимацию в 1 шаг по истечении `transition-duration`.

Такие значения востребованы редко, так как это даже и не анимация почти, но тоже бывают полезны.

Событие transitionend

На конец CSS-анимации можно повесить обработчик на событие `transitionend`.

Это широко используется, чтобы после анимации сделать какое-то действие или объединить несколько анимаций в одну.

Например, лодочка в примере ниже при клике начинает плавать туда-обратно, с каждым разом уплывая всё дальше вправо:

Её анимация осуществляется функцией `go`, которая перезапускается по окончании, с переворотом через CSS:

```
boat.onclick = function() {
  //...
  var times = 1;

  function go() {
    if (times % 2) {
      // плывём вправо
      boat.classList.remove('back');
      boat.style.marginLeft = 100 * times + 200 + 'px';
    } else {
      // плывём влево
      boat.classList.add('back');
      boat.style.marginLeft = 100 * times - 200 + 'px';
```

```
    }

  }

  go();

  boat.addEventListener('transitionend', function() {
    times++;
    go();
  });
};
```

Объект события `transitionend` содержит специфические свойства:

`propertyName`

Свойство, анимация которого завершилась.

`elapsedTime`

Время (в секундах), которое заняла анимация, без учета `transition-delay`.

Свойство `propertyName` может быть полезно при одновременной анимации нескольких свойств. Каждое свойство даст своё событие, и можно решить, что с ним делать дальше.

CSS animations

Более сложные анимации делаются объединением простых при помощи CSS-правила `@keyframes`.

В нём задаётся «имя» анимации и правила: что, откуда и куда анимировать. Затем при помощи свойства `animation: имя параметры` эта анимация подключается к элементу, задаётся время анимации и дополнительные параметры, как её применять.

Пример с пояснениями в комментарии:

```
<div class="progress"></div>

<style>
  /* Современные браузеры, кроме Chrome, Opera, Safari */
  @keyframes go-left-right {   /* назовём анимацию: "go-left-right" */
    from {
      left: 0px;               /* от: left: 0px */
    }
    to {
      left: calc(100% - 50px); /* до: left: 100%-50px */
    }
  }

  /* Префикс для Chrome, Opera, Safari */
  @-webkit-keyframes go-left-right {
    from {
      left: 0px;
    }
    to {
      left: calc(100% - 50px);
    }
  }

  .progress {
    /* применить анимацию go-left-right */
    /* продолжительность 3s */
    /* количество раз: бесконечное (infinite) */
    /* менять направление анимации каждый раз (alternate) */
    animation: go-left-right 3s infinite alternate;
    -webkit-animation: go-left-right 3s infinite alternate;

    position: relative;
    border: 2px solid green;
    width: 50px;
    height: 20px;
    background: lime;
  }
</style>
```

Этот стандарт пока в черновике, поэтому в Chrome, Safari, Opera нужен префикс `-webkit`.

Статей про CSS animations достаточно много, посмотрите, например:

- Статья про CSS Animation.
- Пример бесконечной подпрыгивающей анимации на CSS Animation и кривых Безье.

Итого

CSS-анимации позволяют плавно или не очень менять одно или несколько свойств.

Альтернатива им – плавное изменение значений свойств через JavaScript, мы рассмотрим подробности далее.

Ограничения и достоинства CSS-анимаций по сравнению с JavaScript:

Недостатки	Достоинства

- Временная функция может быть задана кривой Безье или через шаги. Более сложные анимации, состоящие из нескольких кривых, реализуются их комбинацией при помощи CSS animations, но JavaScript-функции всегда гибче.
- CSS-анимации касаются только свойств, а в JavaScript можно делать всё, что угодно, удалять элементы, создавать новые.
- Отсутствует поддержка в IE9-

- Простые вещи делаются просто.
- «Легче» для процессора, чем анимации JavaScript, лучше используется графический ускоритель. Это очень важно для мобильных устройств.

Подавляющее большинство анимаций делается через CSS.

При этом JavaScript запускает их начало – как правило, добавлением класса, в котором задано новое свойство, и может отследить окончание через событие `transitionend` .

Задачи

Анимировать самолёт (CSS)

важность: 5

Реализуйте анимацию, как в демке ниже (клик на картинку):

- Изображение растёт при клике с 40x24px до 400x240px .
- Продолжительность анимации: 3 секунды.
- По окончании вывести «Готово!».
- Если в процессе анимации были дополнительные клики – они не должны ничего «сломать».

Открыть песочницу для задачи.

К решению

Анимировать самолёт с перелётом (CSS)

важность: 5

Модифицируйте решение предыдущей задачи Анимировать самолёт (CSS), чтобы в процессе анимации изображение выросло больше своего стандартного размера 400x240px («выпрыгнуло»), а затем вернулось к нему.

Должно получиться как здесь (клик на картинку)

В качестве исходного документа возьмите решение предыдущей задачи.

К решению

JS-Анимация

JavaScript-анимация применяется там, где не подходит CSS.

Например, по сложной траектории, с временной функцией, выходящей за рамки кривых Безье, на canvas. Иногда её используют для анимации в старых IE.

setInterval

С точки зрения HTML/CSS, анимация – это постепенное изменение стиля DOM-элемента. Например, увеличение координаты `style.left` от `0px` до `100px` сдвигает элемент.

Если увеличивать `left` от `0` до `100` при помощи `setInterval`, делая по 50 изменений в секунду, то это будет выглядеть как плавное перемещение. Тот же принцип, что и в кино: для непрерывной анимации достаточно 24 или больше вызовов `setInterval` в секунду.

Псевдо-код для анимации выглядит так:

```
var fps = 50; // 50 кадров в секунду
var timer = setInterval(function() {
  if (время вышло) clearInterval(timer);
  else немного увеличить left
}, 1000 / fps)
```

Более полный пример кода анимации:

```
var start = Date.now(); // сохранить время начала

var timer = setInterval(function() {
  // вычислить сколько времени прошло с начала анимации
  var timePassed = Date.now() - start;

  if (timePassed >= 2000) {
    clearInterval(timer); // конец через 2 секунды
    return;
  }

  // рисует состояние анимации, соответствующее времени timePassed
  draw(timePassed);

}, 20);

// в то время как timePassed идёт от 0 до 2000
// left принимает значения от 0 до 400px
function draw(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}
```

Кликните для демонстрации:

requestAnimationFrame

Если у нас не один такой `setInterval`, а несколько в разных местах кода, то браузеру нужно в те же 20 мс работать со страницей уже несколько раз. А ведь кроме `setInterval` есть ещё другие действия, к примеру, прокрутка страницы, которую тоже надо нарисовать.

Если все действия по перерисовке производить независимо, то будет выполняться много двойной работы.

Гораздо выгоднее с точки зрения производительности – сгруппировать все перерисовки в одну и запускать их централизованно, все вместе.

Для этого в JavaScript-фреймворках, которые поддерживают анимацию, есть единый таймер:

```
setInterval(function() {
  /* отрисовать все анимации */
}, 20);
```

...Все анимации, которые запускает такой фреймворк, добавляются в общий список, и раз в 20 мс единый таймер проверяет его, запускает текущие, удаляет завершившиеся.

Современные браузеры, кроме IE9-, поддерживают стандарт Animation timing, который представляет собой дальнейший шаг в этом направлении. Он позволяет синхронизировать наши анимации со встроенными механизмами обновления страницы. То есть, сгруппированы будут не только наши, но и CSS-анимации и другие браузерные перерисовки.

При этом графический ускоритель будет использован максимально эффективно, и исключена повторная обработка одних и тех же участков страницы. А значит – меньше будет загрузка CPU, да и сама анимация станет более плавной.

Для этого используется функция requestAnimationFrame.

Синтаксис:

```
var requestId = requestAnimationFrame(callback)
```

Такой вызов планирует запуск `callback` в ближайшее время, когда браузер сочтёт возможным осуществить анимацию.

Если запланировать в `callback` какое-то рисование, то оно будет сгруппировано с другими `requestAnimationFrame` и с внутренними перерисовками браузера.

Возвращаемое значение `requestId` служит для отмены запуска:

```
// отменить запланированное выше выполнение callback
cancelAnimationFrame(requestId);
```

Функция `callback` получает один аргумент – время, прошедшее с начала загрузки страницы, результат вызова performance.now() .

Как правило, запуск `callback` происходит очень скоро. Если у процессора большая загрузка или батарея у ноутбука почти разряжена – то пореже.

Если вы запустите этот код, то увидите промежутки между первыми 20 запусками `requestAnimationFrame` . Как правило, это 10-20 мс, но бывает и больше и меньше. Это оптимальная частота анимации с точки зрения браузера.

```
<script>
  var prev = performance.now();
  var times = 0;

  requestAnimationFrame(function measure(time) {
    document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");
    prev = time;

    if (times++ < 10) requestAnimationFrame(measure);
  })
</script>
```

Функция анимации на основе `requestAnimationFrame` :

```
// Рисует функция draw
// Продолжительность анимации duration
function animate(draw, duration) {
  var start = performance.now();

  requestAnimationFrame(function animate(time) {
    // определить, сколько прошло времени с начала анимации
    var timePassed = time - start;

    // возможно небольшое превышение времени, в этом случае зафиксировать конец
    if (timePassed > duration) timePassed = duration;

    // нарисовать состояние анимации в момент timePassed
    draw(timePassed);

    // если время анимации не закончилось - запланировать ещё кадр
    if (timePassed < duration) {
      requestAnimationFrame(animate);
    }

  });
}
```

Использование для поезда:

```
animate(function(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}, 2000);
```

В действии:

Структура анимации

На основе `requestAnimationFrame` можно соорудить и гораздо более мощную, но в то же время простую функцию анимации.

У анимации есть три основных параметра:

`duration`

Общее время, которое должна длиться анимация, в мс. Например, `1000` .

`timing(timeFraction)`

Временная функция, которая, по аналогии с CSS-свойством `transition-timing-function` , будет по текущему времени вычислять состояние анимации.

Она получает на вход непрерывно возрастающее число `timeFraction` – от `0` до `1` , где `0` означает самое начало анимации, а `1` – её конец.

Её результатом должно быть значение завершённости анимации, которому в CSS transitions на кривых Безье соответствует координата `y` .

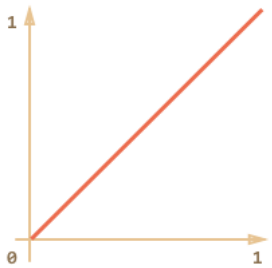
Также по аналогии с `transition-timing-function` должны соблюдаться условия:

- timing(0) = 0
- timing(1) = 1

  ...То есть, анимация начинается в точке `(0,0)` – нулевое время и нулевой прогресс и заканчивается в `(1, 1)` – прошло полное время, и процесс завершён.

  Например, функция-прямая означает равномерное развитие процесса:

```
function linear(timeFraction) {
  return timeFraction;
}
```



  Её график:

  Как видно, её график полностью совпадает с `transition-timing-function: linear` , и эффект абсолютно такой же.

  Есть и другие, более интересные варианты, мы рассмотрим их чуть позже.

`draw(progress)`

Функция, которая получает состояние завершённости анимации и рисует его. Значению `progress=0` соответствует начальная точка анимации, `progress=1` – конечная.

Именно эта функция и осуществляет, собственно, анимацию.

Например, может двигать элемент:

```
function draw(progress) {
  train.style.left = progress + 'px';
}
```

Возможны любые варианты, анимировать можно что угодно и как угодно.

Анимируем ширину элемента `width` от `0` до `100%` , используя нашу функцию.

Кликните для демонстрации:

Код для запуска анимации:

```
animate({
  duration: 1000,
  timing: function(timeFraction) {
    return timeFraction;
  },
  draw: function(progress) {
    elem.style.width = progress * 100 + '%';
  }
});
```

Временные функции

Выше мы видели самую простую, линейную временную функцию.

Рассмотрим примеры анимации движения с использованием различных `timing` .

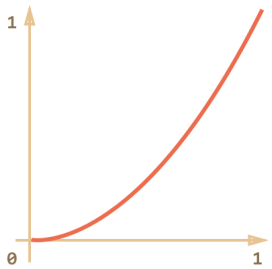
В степени n

Вот еще один простой случай – `progress` в степени `n` . Частные случаи – квадратичная, кубическая функции и т.д.

Для квадратичной функции:

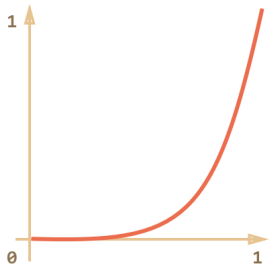
```
function quad(progress) {
  return Math.pow(progress, 2)
}
```

График квадратичной функции:



Пример для квадратичной функции (клик для просмотра):

Увеличение степени влияет на ускорение. Например, график для 5-й степени:



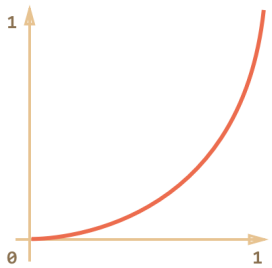
В действии:

Дуга

Функция:

```
function circ(timeFraction) {
  return 1 - Math.sin(Math.acos(timeFraction))
}
```

График:



Back: стреляем из лука

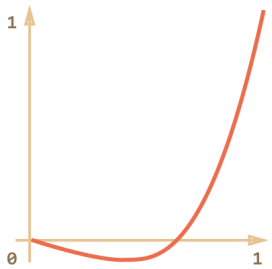
Эта функция работает по принципу лука: сначала мы «натягиваем тетиву», а затем «стреляем».

В отличие от предыдущих функций, эта зависит от дополнительного параметра `x`, который является «коэффициентом упругости». Он определяет расстояние, на которое «оттягивается тетива».

Её код:

```
function back(x, timeFraction) {
  return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x)
}
```

График для **`x = 1.5`**:



Пример для `x = 1.5`:

```
```

Отскок bounce

Представьте, что мы отпускаем мяч, он падает на пол, несколько раз отскакивает и останавливается.

Функция `bounce` делает то же самое, только наоборот: «подпрыгивание» начинается сразу.

Эта функция немного сложнее предыдущих и использует специальные коэффициенты:

```
function bounce(timeFraction) {
  for (var a = 0, b = 1, result; 1; a += b, b /= 2) {
    if (timeFraction >= (7 - 4 * a) / 11) {
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2)
    }
  }
}
```

Код взят из MooTools.FX.Transitions. Конечно же, есть и другие реализации `bounce`.

Пример:

```
```

Упругая анимация

Эта функция зависит от дополнительного параметра `x`, который определяет начальный диапазон.

```
function elastic(x, timeFraction) {
  return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI * x / 3 * timeFraction)
}
```

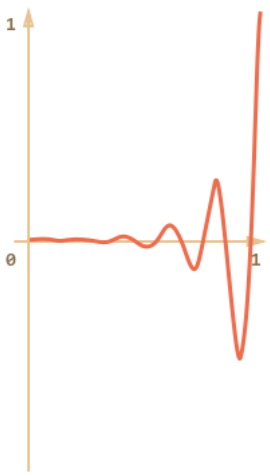


График для **`x=1.5`**:

Пример для `x=1.5`:

```
```

Реверсивные функции ease*

Итак, у нас есть коллекция временных функций.

Их прямое использование называется «easeIn».

Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

easeOut

В режиме «easeOut», значение timing вычисляется по формуле: `timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)`

Например, функция `bounce` в режиме «easeOut»:

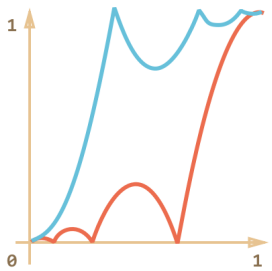
```
// обычный вариант
function bounce(timeFraction) {
  for (var a = 0, b = 1, result; 1; a += b, b /= 2) {
    if (timeFraction >= (7 - 4 * a) / 11) {
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2);
    }
  }
}

// преобразователь в easeOut
function makeEaseOut(timing) {
  return function(timeFraction) {
    return 1 - timing(1 - timeFraction);
  }
}
```

```
var bounceEaseOut = makeEaseOut(bounce);
```

Полный пример – отскок в `bounceEaseOut` теперь не в начале, а в конце (и это куда красивее):

На этом графике видно, как преобразование `easeOut` изменяет поведение функции:



Если есть анимационный эффект, такой как подпрыгивание – он будет показан в конце, а не в начале (или наоборот, в начале, а не в конце).

Красным цветом обозначен обычный вариант, а синим – `easeOut`.

- Обычно анимируемый объект сначала медленно скачет внизу, а затем, в конце, резко достигает верха...
- А после `easeOut` – он сначала прыгает наверх, а затем медленно скачет внизу.

easeInOut

А еще можно сделать так, чтобы показать эффект *и в начале и в конце* анимации. Соответствующее преобразование называется «easeInOut».

Его код выглядит так:

```
if (timeFraction <= 0.5) { // первая половина анимации)
  return timing(2 * timeFraction) / 2;
} else { // вторая половина
  return (2 - timing(2 * (1 - timeFraction))) / 2;
}
```

Код, который трансформирует `timing`:

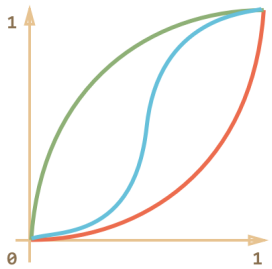
```
function makeEaseInOut(timing) {
  return function(timeFraction) {
    if (timeFraction < .5)
      return timing(2 * timeFraction) / 2;
    else
      return (2 - timing(2 * (1 - timeFraction))) / 2;
  }
}

bounceEaseInOut = makeEaseInOut(bounce);
```

Пример с `bounceEaseInOut`:

Трансформация «easeInOut» объединяет в себе два графика в один: `easeIn` для первой половины анимации и `easeOut` – для второй.

Это отлично видно, если посмотреть графики `easeIn`, `easeOut` и `easeInOut` на примере функции `circ`:



- Красным цветом обозначен обычный вариант функции `circ`.
- Зелёным – `easeOut`.
- Синим – `easeInOut`.

Как видно, график первой половины анимации представляет собой уменьшенный «easeIn», а второй – уменьшенный «easeOut». В результате, анимация начинается и заканчивается одинаковым эффектом.

> Процесс анимации полностью в ваших руках благодаря `timing`. Её можно сделать настолько реалистичной, насколько захочется.
>
> Впрочем, исходя из практики, можно сказать, что варианты `timing`, описанные выше, покрывают 95% потребностей в анимации.

Сложные варианты step

Анимировать можно все, что угодно. Вместо движения, как во всех предыдущих примерах, можно изменять любые CSS свойства... И не только!

Достаточно лишь написать соответствующий `draw`.

Набор текста

Можно, к примеру, анимировать набор текста в «скачущем» режиме:

Итого

Анимация выполняется путём вызовов `requestAnimationFrame`. Для поддержки IE9- желательно подключить полифилл, который будет внутри использовать `setTimeout`. Это будет всё равно лучше, чем независимые `setInterval`.

Реализация анимации – очень простая и вместе с тем гибкая:

```
function animate(options) {

  var start = performance.now();

  requestAnimationFrame(function animate(time) {
    // timeFraction от 0 до 1
    var timeFraction = (time - start) / options.duration;
    if (timeFraction > 1) timeFraction = 1;

    // текущее состояние анимации
    var progress = options.timing(timeFraction)

    options.draw(progress);

    if (timeFraction < 1) {
      requestAnimationFrame(animate);
    }

  });
}
```

Основные параметры:

- `duration` – длительность анимации в мс.
- `timing` – функция, которая определяет состояние анимации каждый кадр. Получает часть времени от 0 до 1, возвращает завершенность анимации от 0 до 1.
- `draw` – функция, которая отрисовывает состояние анимации от 0 до 1.

Эту функцию можно улучшить, например добавить коллбэк `complete` для вызова в конце анимации.

Мы рассмотрели ряд примеров для `timing` и трансформации `easeOut`, `easeInOut`, которые позволяют их разнообразить. В отличие от CSS мы не ограничены кривыми Безье, можно реализовать всё, что угодно.

Это же относится и к функции `draw`.

Такая реализация анимации имеет три основных области применения:

- Нестандартные задачи и требования, не укладывающиеся в рамки CSS.

- Поддержка IE9-.
- Графика, рисование на canvas.

Задачи

Анимируйте мяч

важность: 5

Сделайте так, чтобы мяч подпрыгивал. Кликните по мячу, чтобы увидеть, как это должно выглядеть.

Открыть песочницу для задачи.

К решению

Анимируйте падение мяча с отскоками вправо

важность: 5

Заставьте мяч падать вправо. Кликните, чтобы увидеть в действии.

Напишите код, который будет анимировать мяч. Дистанция вправо составляет `100px` .

В качестве исходного кода возьмите решение предыдущей задачи Анимируйте мяч.

К решению

Оптимизация

Утечки памяти, увеличение скорости выполнения и загрузки скриптов.

Введение

В отличие от ряда других курсов учебника, этот раздел – не является курсом как таковым. Связное и грамотное изложение темы требует времени, которое я пока не могу ему уделить. Но, надеюсь, смогу в будущем.

Пока что раздел содержит лишь некоторые статьи-заметки по теме оптимизации, которые, надеюсь, будут вам полезны.

Как работают сжиматели JavaScript

Перед выкладыванием JavaScript на «боевую» машину – пропускаем его через минификатор (также говорят «сжиматель»), который удаляет пробелы и по-всякому оптимизирует код, уменьшая его размер.

В этой статье мы посмотрим, как работают современные минификаторы, за счёт чего они укорачивают код и какие с ними возможны проблемы.

Современные сжиматели

Рассматриваемые в этой статье алгоритмы и подходы относятся к минификаторам последнего поколения.

Вот их список:

- Google Closure Compiler
- UglifyJS
- Microsoft AJAX Minifier

Самые широко используемые – первые два, поэтому будем рассматривать в первую очередь их.

Наша цель – понять, как они работают, и что интересного с их помощью можно сотворить.

С чего начать?

Для GCC:

1. Убедиться, что стоит Java
2. Скачать и распаковать http://closure-compiler.googlecode.com/files/compiler-latest.zip, нам нужен файл `compiler.jar`.
3. Сжать файл `my.js`: `java -jar compiler.jar --charset UTF-8 --js my.js --js_output_file my.min.js`

Обратите внимание на флаг `--charset` для GCC. Без него русские буквы будут закодированы во что-то типа `\u1234`.

Google Closure Compiler также содержит песочницу для тестирования сжатия и веб-сервис, на который код можно отправлять для сжатия. Но скачать файл обычно гораздо проще, поэтому его редко где используют.

Для UglifyJS:

1. Убедиться, что стоит Node.js
2. Поставить `npm install -g uglify-js`.
3. Сжать файл `my.js`: `uglifyjs my.js -o my.min.js`

Что делает минификатор?

Все современные минификаторы работают следующим образом:

1. Разбирают JavaScript-код в синтаксическое дерево.

   Также поступает любой интерпретатор JavaScript перед тем, как его выполнять. Но затем, вместо исполнения кода...

2. Бегают по этому дереву, анализируют и оптимизируют его.
3. Записывают из синтаксического дерева получившийся код.

Как выглядит дерево?

Посмотреть синтаксическое дерево можно, запустив компилятор со специальным флагом.

Для GCC есть даже способ вывести его:

1. Сначала сгенерируем дерево в формате DOT:

   ```
   java -jar compiler.jar --js my.js --use_only_custom_externs --print_tree >my.dot
   ```

   Здесь флаг `--print_tree` выводит дерево, а `--use_only_custom_externs` убирает лишнюю служебную информацию.

2. Файл в этом формате используется в различных программах для графопостроения.

   Чтобы превратить его в обычную картинку, подойдёт утилита `dot` из пакета Graphviz:

   ```
   // конвертировать в формат png
   dot -Tpng my.dot -o my.png

   // конвертировать в формат svg
   dot -Tsvg my.dot -o my.svg
   ```

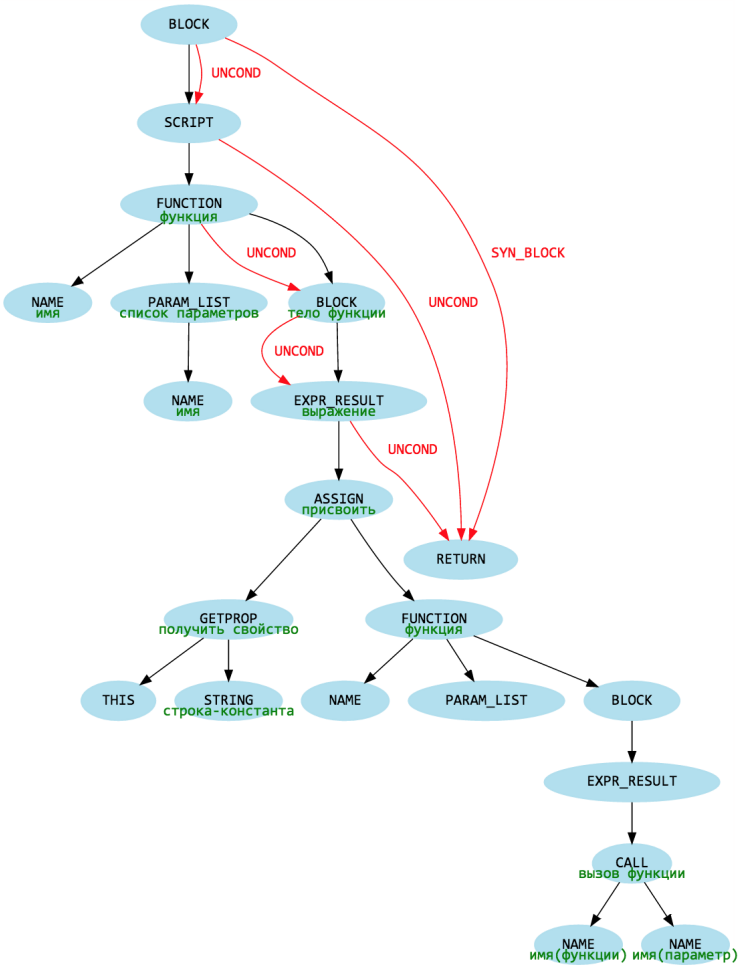
Пример кода `my.js`:

```
function User(name) {

  this.sayHi = function() {
    alert( name );
  };

}
```

Результат, получившееся из `my.js` дерево:



В узлах-эллипсах на иллюстрации выше стоит тип, например `FUNCTION` (функция) или `NAME` (имя переменной). Комментарии к ним на русском языке добавлены мной вручную.

Кроме него к каждому узлу привязаны конкретные данные. Сжиматель умеет ходить по этому дереву и менять его, как пожелает.

Комментарии JSDoc

Обычно когда код превращается в дерево – из него естественным образом исчезают комментарии и пробелы. Они не имеют значения при выполнении, поэтому игнорируются.

Но Google Closure Compiler добавляет в дерево информацию из *комментариев JSDoc*, т.е. комментариев вида `/** ... */`, например:

```
/**
 * Номер минимальной поддерживаемой версии IE
 * @const
 * @type {number}
 */
var minIEVersion = 8;
```

Такие комментарии не создают новых узлов дерева, а добавляются в качестве информации к существующем. В данном случае – к переменной `minIEVersion`.

В них может содержаться информация о типе переменной (`number`) и другая, которая поможет сжимателю лучше оптимизировать код (`const` – константа).

Оптимизации

Сжиматель бегает по дереву, ищет «паттерны» – известные ему структуры, которые он знает, как оптимизировать, и обновляет дерево.

В разных минификаторах реализован разный набор оптимизаций, сами оптимизации применяются в разном порядке, поэтому результаты работы могут отличаться. В примерах ниже даётся результат работы GCC.

Объединение и сжатие констант

До оптимизации:

```
function test(a, b) {
  run(a, 'my' + 'string', 600 * 600 * 5, 1 && 0, b && 0)
}
```

После:

```
function test(a,b){run(a,"mystring",18E5,0,b&&0)};
```

- `'my' + 'string'` → `"mystring"`.
- `600 * 600 * 5` → `18E5` (научная форма числа, для краткости).
- `1 && 0` → `0`.
- `b && 0` → без изменений, т.к. результат зависит от `b`.

Укорачивание локальных переменных

До оптимизации:

```
function sayHi(name, message) {
  alert(name +" сказал: " + message);
}
```

После оптимизации:

```
function sayHi(a,b){alert(a+" сказал: "+b)};
```

- Локальная переменная заведомо доступна только внутри функции, поэтому обычно её переименование безопасно (необычные случаи рассмотрим далее).
- Также переименовываются локальные функции.
- Вложенные функции обрабатываются корректно.

Объединение и удаление локальных переменных

До оптимизации:

```
function test(nodeId) {
  var elem = document.getElementsById(nodeId);
  var parent = elem.parentNode;
  alert( parent );
}
```

После оптимизации GCC:

```
function test(a){a=document.getElementsById(a).parentNode;alert(a)};
```

- Локальные переменные были переименованы.
- Лишние переменные убраны. Для этого сжиматель создаёт вспомогательную внутреннюю структуру данных, в которой хранятся сведения о «пути использования» каждой переменной. Если одна переменная заканчивает свой путь и начинает другая, то вполне можно дать им одно имя.
- Кроме того, операции `elem = getElementsById` и `elem.parentNode` объединены, но это уже другая оптимизация.

Уничтожение недостижимого кода, разворачивание `if`-веток

До оптимизации:

```
function test(node) {
  var parent = node.parentNode;

  if (0) {
    alert( "Привет с параллельной планеты" );
  } else {
    alert( "Останется только один" );
  }

  return;

  alert( 1 );
}
```

После оптимизации:

```
function test(){alert("Останется только один")}
```

- Если переменная присваивается, но не используется, она может быть удалена. В примере выше эта оптимизация была применена к переменной `parent`, а затем и к параметру `node`.
- Заведомо ложная ветка `if(0) { .. }` убрана, заведомо истинная – оставлена.

  То же самое будет с условиями в других конструкциях, например `a = true ? c : d` превратится в `a = c`.
- Код после `return` удалён как недостижимый.

Переписывание синтаксических конструкций

До оптимизации:

```
var i = 0;
while (i++ < 10) {
  alert( i );
}

if (i) {
  alert( i );
}

if (i == '1') {
  alert( 1 );
} else if (i == '2') {
  alert( 2 );
} else {
  alert( i );
}
```

После оптимизации:

```
for(var i=0;10>i++;)alert(i);i&&alert(i);"1"==i?alert(1):"2"==i?alert(2):alert(i);
```

- Конструкция `while` переписана в `for`.
- Конструкция `if (i) ...` переписана в `i&&...`.
- Конструкция `if (cond) ... else ...` была переписана в `cond ? ... : ...`.

Инлайнинг функций

Инлайнинг функции – приём оптимизации, при котором функция заменяется на своё тело.

До оптимизации:

```
function sayHi(message) {

  var elem = createMessage('div', message);
  showElement(elem);

  function createMessage(tagName, message) {
    var el = document.createElement(tagName);
    el.innerHTML = message;
    return el;
  }

  function showElement(elem) {
    document.body.appendChild(elem);
  }
}
```

После оптимизации (переводы строк также будут убраны):

```
function sayHi(b) {
  var a = document.createElement("div");
  a.innerHTML = b;
  document.body.appendChild(a)
};
```

- Вызовы функций `createMessage` и `showElement` заменены на тело функций. В данном случае это возможно, так как функции используются всего по разу.
- Эта оптимизация применяется не всегда. Если бы каждая функция использовалась много раз, то с точки зрения размера выгоднее оставить их «как есть».

Инлайнинг переменных

Переменные заменяются на значение, если оно заведомо известно.

До оптимизации:

```
(function() {
  var isVisible = true;
  var hi = "Привет вам из JavaScript";

  window.sayHi = function() {
    if (isVisible) {
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
      alert( hi );
    }
  }

})();
```

После оптимизации:

```
(function() {
    window.sayHi = function() {
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
      alert( "Привет вам из JavaScript" );
    };
  }
})();
```

- Переменная `isVisible` заменена на `true` , после чего `if` стало возможным убрать.
- Переменная `hi` заменена на строку.

  Казалось бы – зачем менять `hi` на строку? Ведь код стал ощутимо длиннее!

  ...Но всё дело в том, что минификатор знает, что дальше код будет сжиматься при помощи gzip. Во всяком случае, все правильно настроенные сервера так делают.

Алгоритм работы gzip заключается в том, что он ищет повторы в данных и выносит их в специальный «словарь», заменяя на более короткий идентификатор. Архив как раз и состоит из словаря и данных, в которых дубликаты заменены на идентификаторы.

`Если вынести строку обратно в переменную, то получится как раз частный случай такого сжатия -- взяли `"Привет вам из JavaScript"` и заменили`

`Плюс такого подхода станет очевиден, если сжать gzip оба кода -- до и после минификации. Минифицированный gzip-сжатый код в итоге даст меньш`

Разные мелкие оптимизации

Кроме основных оптимизаций, описанных выше, есть ещё много мелких:

- Убираются лишние кавычки у ключей

  ```
  ```js no-beautify
  {"prop" : "val" }   =>  {prop:"val"}
  ```
  ```

- Упрощаются простые вызовы `Array/Object`

  ```
  ```js no-beautify
  a = new Array()   =>  a = []
  o = new Object()  => o = {}
  ```
  ```

  Эта оптимизация предполагает, что `Array` и `Object` не переопределены программистом. Для включения её в UglifyJS нужен флаг `--unsafe` .

- ...И еще некоторые другие мелкие изменения кода...

Подводные камни

Описанные оптимизации, в целом, безопасны, но есть ряд подводных камней.

Конструкция with

Рассмотрим код:

```
function changePosition(style) {
  var position, test;

  with (style) {
    position = 'absolute';
  }
}
```

Куда будет присвоено значение `position = 'absolute'` ?

Это неизвестно до момента выполнения: если свойство `position` есть в `style` – то туда, а если нет – то в локальную переменную.

Можно ли в такой ситуации заменить локальную переменную на более короткую? Очевидно, нет:

```
function changePosition(style) {
  var a, b;

  with (style) {          // а что, если в style нет такого свойства?
    position = 'absolute';// куда будет осуществлена запись? в window.position?
  }
}
```

Такая же опасность для сжатия кроется в использованном `eval` . Ведь `eval` может обращаться к локальным переменным:

```
function f(code) {
  var myVar;

  eval(code); // а что, если будет присвоение eval("myVar = ...") ?

  alert(myVar);
```

Получается, что при наличии `eval` мы не имеем права переименовывать локальные переменные. Причём (!), если функция является вложенной, то и во внешних функциях тоже.

А ведь сжатие переменных – очень важная оптимизация. Как правило, она уменьшает размер сильнее всего.

Что делать? Разные минификаторы поступают по-разному.

- UglifyJS – не будет переименовывать переменные. Так что наличие `with/eval` сильно повлияет на степень сжатие кода.
- GCC – всё равно сожмёт локальные переменные. Это, конечно же, может привести к ошибкам, причём в сжатом коде, отлаживать который не очень-то удобно. Поэтому он выдаст предупреждение о наличии опасной конструкции.

Ни тот ни другой вариант нас, по большому счёту, не устраивают.

Для того, чтобы код сжимался хорошо и работал правильно, не используем **`with`** и **`eval`**.

Либо, если уж очень надо использовать – делаем это с оглядкой на поведение минификатора, чтобы не было проблем.

Условная компиляция IE10-

В IE10- поддерживалось условное выполнение JavaScript.

Синтаксис: `/*@cc_on код */`.

Такой код выполнится в IE10-, например:

```
var isIE /*@cc_on =true@*/ ;

alert( isIE ); // true в IE10-
```

Можно хитро сделать, чтобы комментарий остался, например так:

```
var isIE = new Function('', '/*@cc_on return true@*/')();

alert( isIE ); // true в IE.
```

...Однако, с учётом того, что в современных IE11+ эта компиляция не работает в любом случае, лучше избавиться от неё вообще.

В следующих главах мы посмотрим, какие продвинутые возможности есть в минификаторах, как сделать сжатие более эффективным.

Улучшаем сжатие кода

Здесь мы обсудим разные приёмы, которые используются, чтобы улучшить сжатие кода.

Больше локальных переменных

Например, код jQuery обёрнут в функцию, запускаемую «на месте».

```
(function(window, undefined) {
  // ...
  var jQuery = ...

  window.jQuery = jQuery; // сделать переменную глобальной

})(window);
```

Переменные `window` и `undefined` стали локальными. Это позволит сжимателю заменить их на более короткие.

ООП без прототипов

Приватные переменные будут сжаты и заинлайнены.

Например, этот код хорошо сожмётся:

```
function User(firstName, lastName) {
  var fullName = firstName + ' ' + lastName;

  this.sayHi = function() {
    showMessage(fullName);
  }

  function showMessage(msg) {
    alert( '**' + msg + '**' );
  }
}
```

...А этот – плохо:

```
function User(firstName, lastName) {
  this._firstName = firstName;
  this._lastName = lastName;
}

User.prototype.sayHi = function() {
  this._showMessage(this._fullName);
}

User.prototype._showMessage = function(msg) {
  alert( '**' + msg + '**' );
}
```

Сжимаются только локальные переменные, свойства объектов не сжимаются, поэтому эффект от сжатия для второго кода будет совсем небольшим.

При этом, конечно, нужно иметь в виду общий стиль ООП проекта, достоинства и недостатки такого подхода.

Сжатие под платформу, define

Можно делать разные сборки в зависимости от платформы (мобильная/десктоп) и браузера.

Ведь не секрет, что ряд функций могут быть реализованы по разному, в зависимости от того, поддерживает ли среда выполнения нужную возможность.

Способ 1: локальная переменная

Проще всего это сделать локальной переменной в модуле:

```
(function($) {

  /** @const */
  var platform = 'IE';

  // .....

  if (platform == 'IE') {
    alert( 'IE' );
  } else {
    alert( 'NON-IE' );
  }

})(jQuery);
```

Нужное значение директивы можно вставить при подготовке JavaScript к сжатию.

Сжиматель заинлайнит её и оптимизирует соответствующие IE.

Способ 2: define

UglifyJS и GCC позволяют задать значение глобальной переменной из командной строки.

В GCC эта возможность доступна лишь в «продвинутом режиме» работы оптимизатора, который мы рассмотрим далее (он редко используется).

Удобнее в этом плане устроен UglifyJS. В нём можно указать флаг `-d SYMBOL[=VALUE]`, который заменит все переменные `SYMBOL` на указанное значение `VALUE`. Если `VALUE` не указано, то оно считается равным `true`.

Флаг не работает, если переменная определена явно.

Например, рассмотрим код:

```
// my.js
if (isIE) {
  alert( "Привет от IE" );
} else {
  alert( "Не IE :)" );
}
```

Сжатие вызовом `uglifyjs -d isIE my.js` даст:

```
alert( "Привет от IE" );
```

...Ну а чтобы код работал в обычном окружении, нужно определить в нём значение переменной по умолчанию. Это обычно делается в каком-то другом файле (на весь проект), так как если объявить `var isIE` в этом, то флаг `-d isIE` не сработает.

Но можно и «хакнуть» сжиматель, объявив переменную так:

```
// объявит переменную при отсутствии сжатия
// при сжатии не повредит
window.isIE = window.isIE || getBrowserVersion();
```

Убираем вызовы console.*

Минификатор имеет в своём распоряжении дерево кода и может удалить ненужные вызовы.

Для UglifyJS это делают опции компилятора:

- `drop_debugger` – убирает вызовы `debugger`.
- `drop_console` – убирает вызовы `console.*`.

Можно написать и дополнительную функцию преобразования, которая убирает другие вызовы, например для `log.*`:

```
var uglify = require('uglify-js');
var pro = uglify.uglify;

function ast_squeeze_console(ast) {
  var w = pro.ast_walker(),
    walk = w.walk,
    scope;
  return w.with_walkers({
    "stat": function(stmt) {
      if (stmt[0] === "call" && stmt[1][0] == "dot" && stmt[1][1] instanceof Array && stmt[1][1][0] == 'name' && stmt[1][1][1] == "log") {
        return ["block"];
      }
      return ["stat", walk(stmt)];
    },
    "call": function(expr, args) {
      if (expr[0] == "dot" && expr[1] instanceof Array && expr[1][0] == 'name' && expr[1][1] == "console") {
        return ["atom", "0"];
      }
    }
  }, function() {
    return walk(ast);
  });
};
```

Эту функцию следует вызвать на результате `parse`, и она пройдётся по дереву и удалит все вызовы `log.*`.

Утечки памяти

Утечки памяти происходят, когда браузер по какой-то причине не может освободить память от недостижимых объектов.

Обычно это происходит автоматически (Управление памятью в JavaScript). Кроме того, браузер освобождает память при переходе на другую страницу. Поэтому утечки в реальной жизни проявляют себя в двух ситуациях:

1. Приложение, в котором посетитель все время на одной странице и работает со сложным JavaScript-интерфейсом. В этом случае утечки могут постепенно съедать доступную память.
2. Страница регулярно делает что-то, вызывающее утечку памяти. Посетитель (например, менеджер) оставляет компьютер на ночь включенным, чтобы не закрывать браузер с кучей вкладок. Приходит утром – а браузер съел всю память ~~и рухнул~~ и сильно тормозит.

Утечки бывают из-за ошибок браузера, ошибок в расширениях браузера и, гораздо реже, по причине ошибок в архитектуре JavaScript-кода. Мы разберём несколько наиболее частых и важных примеров.

Коллекция утечек в IE

Утечка DOM ↔ JS в IE8-

IE до версии 8 не умел очищать циклические ссылки, появляющиеся между DOM-объектами и объектами JavaScript. В результате и DOM и JS оставались в памяти навсегда.

В браузере IE8 была проведена серьёзная работа над ошибками, но утечка в IE8- появляется, если круговая ссылка возникает «через объект».

Чтобы было понятнее, о чём речь, посмотрите на следующий код. Он вызывает утечку памяти в IE8-:

```
function leak() {
  // Создаём новый DIV, добавляем к BODY
  var elem = document.createElement('div');
  document.body.appendChild(elem);

  // Записываем в свойство жирный объект
  elem.__expando = {
    bigAss: new Array(1000000).join('lalala')
  };

  // Создаём круговую ссылку. Без этой строки утечки не будет.
  elem.__expando.__elem = elem;

  // Удалить элемент из DOM. Браузер должен очистить память.
  elem.parentElement.removeChild(elem);
}
```

Полный пример (только для IE8-, а также IE9 в режиме совместимости с IE8):

Круговая ссылка и, как следствие, утечка может возникать и неявным образом, через замыкание:

```
function leak() {
  var elem = document.createElement('div');
  document.body.appendChild(elem);

  elem.__expando = {
    bigAss: new Array(1000000).join('lalala'),
    method: function() {} // создаётся круговая ссылка через замыкание
  };

  // Удалить элемент из DOM. Браузер должен очистить память.
```

```
  elem.parentElement.removeChild(elem);
}
```

Полный пример (IE8-, IE9 в режиме совместимости с IE8):

Без привязки метода `method` к элементу здесь утечки не возникнет.

Бывает ли такая ситуация в реальной жизни? Или это – целиком синтетический пример, для заумных программистов?

Да, конечно бывает. Например, при разработке графических компонент – бывает удобно присвоить DOM-элементу ссылку на JavaScript-объект, который представляет собой компонент. Это упрощает делегирование и, в общем-то, логично, что DOM-элемент знает о компоненте на себе. Но в IE8- прямая привязка ведёт к утечке памяти!

Примерно так:

```
function Menu(elem) {
  elem.onclick = function() {};
}

var menu = new Menu(elem); // Menu содержит ссылку на elem
elem.menu = menu; // такая привязка или что-то подобное ведёт к утечке в IE8
```

Полный пример (IE8-, IE9 в режиме совместимости с IE8):

Утечка IE8 при обращении к коллекциям таблицы

Эта утечка происходит только в IE8 в стандартном режиме. В нём при обращении к табличным псевдо-массивам (напр. `rows`) создаются и не очищаются внутренние ссылки, что приводит к утечкам.

Также воспроизводится в новых IE в режиме совместимости с IE8.

Код:

```
var elem = document.createElement('div'); // любой элемент

function leak() {

  elem.innerHTML = '<table><tr><td>1</td></tr></table>';

  elem.firstChild.rows[0]; // просто доступ через rows[] приводит к утечке
  // при том, что мы даже не сохраняем значение в переменную

  elem.removeChild(elem.firstChild); // удалить таблицу (*)
  // alert(elem.childNodes.length) // выдал бы 0, elem очищен, всё честно
}
```

Полный пример (IE8):

Особенности:

- Если убрать отмеченную строку, то утечки не будет.
- Если заменить строку `(*)` на `elem.innerHTML = ''` , то память будет очищена, т.к. этот способ работает по-другому, нежели просто `removeChild` (см. главу Управление памятью в JavaScript).
- Утечка произойдёт не только при доступе к `rows` , но и к другим свойствам, например `elem.firstChild.tBodies[0]` .

Эта утечка проявляется, в частности, при удалении детей элемента следующей функцией:

```
function empty(elem) {
  while (elem.firstChild) elem.removeChild(elem.firstChild);
}
```

Если идёт доступ к табличным коллекциям и регулярное обновление таблиц при помощи DOM-методов – утечка в IE8 будет расти.

Более подробно вы можете почитать об этой утечке в статье Утечки памяти в IE8, или страшная сказка со счастливым концом.

Утечка через XmlHttpRequest в IE8-

Следующий код вызывает утечки памяти в IE8-:

```
function leak() {
  var xhr = new XMLHttpRequest();

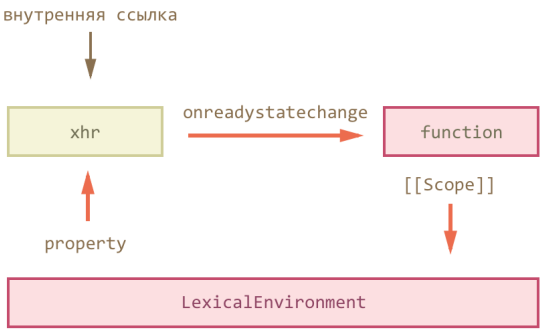
  xhr.open('GET', '/server.do', true);

  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      // ...
    }
  }

  xhr.send(null);
}
```

Как вы думаете, почему? Если вы внимательно читали то, что написано выше, то имеете информацию для ответа на этот вопрос...

Посмотрим, какая структура памяти создается при каждом запуске:



Когда запускается асинхронный запрос `xhr`, браузер создаёт специальную внутреннюю ссылку (internal reference) на этот объект и будет поддерживать её, пока он находится в процессе коммуникации. Именно поэтому объект `xhr` будет жив после окончания работы функции.

Когда запрос завершен, браузер удаляет внутреннюю ссылку, `xhr` становится недостижимым и память очищается... Везде, кроме IE8-.

Полный пример (IE8):

```
function leak() {
  var xhr = new XMLHttpRequest();

  xhr.open('GET', 'something.js?' + Math.random(), true);

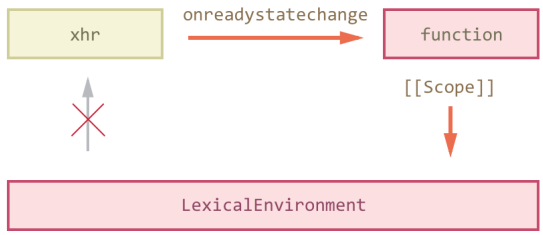
  xhr.onreadystatechange = function() {
    if (xhr.readyState != 4) return;

    if (xhr.status == 200) {
      document.getElementById('test').innerHTML++;
    }

    xhr = null; // по завершении запроса удаляем ссылку из замыкания
  }

  xhr.send(null);
}
```

Чтобы это исправить, нам нужно разорвать круговую ссылку `XMLHttpRequest ⇔ JS`. Например, можно удалить `xhr` из замыкания:



Теперь циклической ссылки нет – и не будет утечки.

Объемы утечек памяти

Объем «утекающей» памяти может быть небольшим. Тогда это почти не ощущается. Но так как замыкания ведут к сохранению переменных внешних функций, то одна функция может тянуть за собой много чего ещё.

Представьте, вы создали функцию, и одна из ее переменных содержит очень большую по объему строку (например, получает с сервера).

```
function f() {
  var data = "Большой объем данных, например, переданных сервером"

  /* делаем что-то хорошее (ну или плохое) с полученными данными */

  function inner() {
    // ...
  }

  return inner;
}
```

Пока функция `inner` остается в памяти, `LexicalEnvironment` с переменной большого объема внутри висит в памяти.

Висит до тех пор, пока функция `inner` жива.

Как правило, JavaScript не знает, какие из переменных функции `inner` будут использованы, поэтому оставляет их все. Исключение – виртуальная машина V8 (Chrome, Opera, Node.JS), она часто (не всегда) видит, что переменная не используется во внутренних функциях, и очистит память.

В других же интерпретаторах, даже если код спроектирован так, что никакой утечки нет, по вполне разумной причине может создаваться множество функций, а память будет расти потому, что функция тянет за собой своё замыкание.

Сэкономить память здесь вполне можно. Мы же знаем, что переменная `data` не используется в `inner`. Поэтому просто обнулим её:

```
function f() {
  var data = "Большое количество данных, например, переданных сервером"

  /* действия с data */

  function inner() {
    // ...
  }

  data = null; // когда data станет не нужна -

  return inner;
}
```

Поиск и устранение утечек памяти

Проверка на утечки

Существует множество шаблонов утечек и ошибок в браузерах, которые могут приводить к утечкам. Для их устранения сперва надо постараться изолировать и воспроизвести утечку.

- Необходимо помнить, что браузер может очистить память не сразу когда объект стал недостижим, а чуть позже. Например, сборщик мусора может ждать, пока не будет достигнут определенный лимит использования памяти, или запускаться время от времени.

Поэтому если вы думаете, что нашли проблему и тестовый код, запущенный в цикле, течёт – подождите примерно минуту, добейтесь, чтобы памяти ело стабильно и много. Тогда будет понятно, что это не особенность сборщика мусора.

- Если речь об IE, то надо смотреть «Виртуальную память» в списке процессов, а не только обычную «Память». Обычная может очищаться за счет того, что перемещается в виртуальную (на диск).
- Для простоты отладки, если есть подозрение на утечку конкретных объектов, в них добавляют большие свойства-маркеры. Например, подойдет фрагмент текста: `new Array(999999).join('leak')`.

Настройка браузера

Утечки могут возникать из-за расширений браузера, взаимодействующих со страницей. Еще более важно, что утечки могут быть следствием конфликта двух браузерных расширений Например, было такое: память текла когда включены расширения Skype и плагин антивируса одновременно.

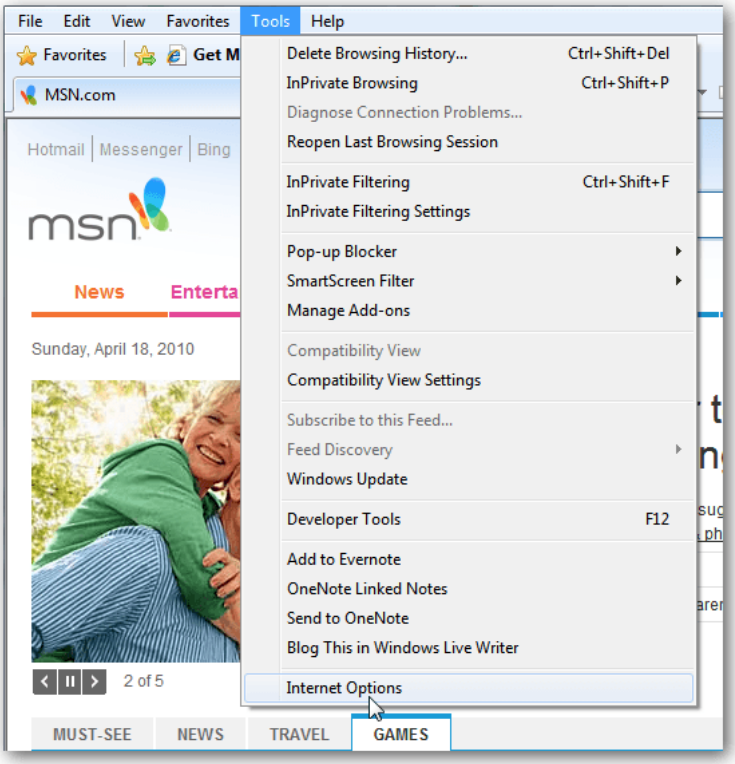
Чтобы понять, в расширениях дело или нет, нужно отключить их:

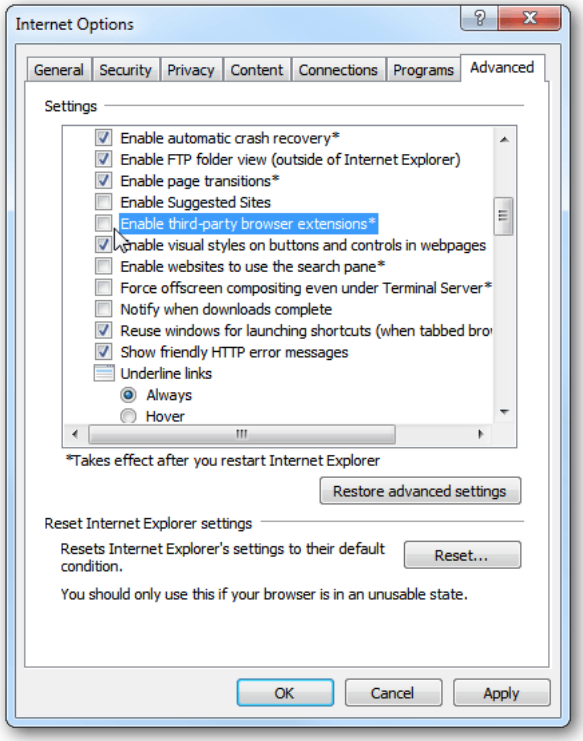
1. Отключить Flash.
2. Отключить антивирусную защиту, проверку ссылок и другие модули, и дополнения.
3. Отключить плагины. Отключить ВСЕ плагины.

- Для IE есть параметр коммандной строки:

  ```
  "C:\Program Files\Internet Explorer\iexplore.exe" -extoff
  ```

  Кроме того необходимо отключить сторонние расширения в свойствах IE.



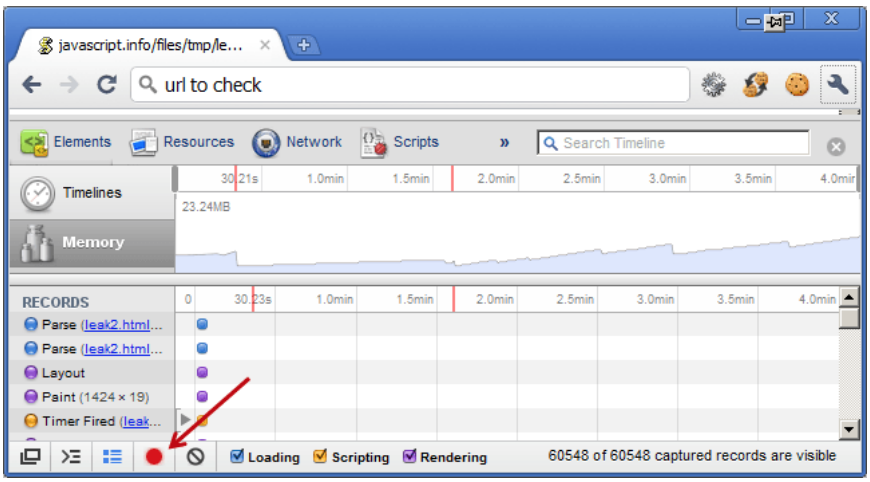


- Firefox необходимо запускать с чистым профилем. Используйте следующую команду для запуска менеджера профилей и создания чистого пустого профиля:

  ```
  firefox --profilemanager
  ```

Инструменты

Пожалуй, единственный браузер с поддержкой отладки памяти – это Chrome. В инструментах разработчика вкладка Timeline – Memory показывает график использования памяти.



Можем посмотреть, сколько памяти используется и на что.

Также в Profiles есть кнопка Take Heap Snapshot, здесь можно сделать и исследовать снимок текущего состояния страницы. Снимки можно сравнивать друг с другом, выяснять количество новых объектов. Можно смотреть, почему объект не очищен и кто на него ссылается.

Замечательная статья на эту тему есть в документации: Chrome Developer Tools: Heap Profiling.

Утечки памяти штука довольно сложная. В борьбе с ними вам определенно понадобится одна вещь: *Удача!*

Утечки памяти при использовании jQuery

В jQuery для хранения обработчиков событий и других вспомогательных данных, связанных с DOM-элементами, используется внутренний объект, который в jQuery 1 доступен через $.data.

В jQuery 2 доступ к нему закрыт через замыкание, он стал локальной переменной внутри jQuery с именем `data_priv`, но в остальном всё работает точно так, как описано, и с теми же последствиями.

$.data

Встроенная функция `$.data` позволяет хранить и привязывать произвольные значения к DOM-узлам.

Например:

```
// присвоить
$(document).data('prop', { anything: "любой объект" })

// прочитать
alert( $(document).data('prop').anything ) // любой объект
```

Реализована она хитрым образом. Данные не хранятся в самом элементе, а во внутреннем объекте jQuery.

jQuery-вызов `elem.data(prop, val)` делает следующее:

1. Элемент получает уникальный идентификатор, если у него такого еще нет:

```
elem[jQuery.expando] = id = ++jQuery.uuid; // средствами jQuery
```

`jQuery.expando` – это случайная строка, сгенерированная jQuery один раз при входе на страницу. Уникальное свойство, чтобы ничего важного не перезаписать. 2. ...А сами данные сохраняются в специальном объекте `jQuery.cache`:

```
```js no-beautify
jQuery.cache[id]['prop'] = { anything: "любой объект" };
```
```

Когда данные считываются из элемента:

1. Уникальный идентификатор элемента извлекается из `id = elem[ jQuery.expando]`.
2. Данные считываются из `jQuery.cache[id]`.

Смысл этого API в том, что DOM-элемент никогда не ссылается на JavaScript объект напрямую. Задействуется идентификатор, а сами данные хранятся в `jQuery.cache`. Утечек в IE не будет.

К тому же все данные известны библиотеке, так что можно клонировать с ними и т.п.

Как побочный эффект – возникает утечка памяти, если элемент удален из DOM без дополнительной очистки.

Примеры утечек в jQuery

Следующая функция `leak` создает jQuery-утечку во всех браузерах:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>

<div id="data"></div>

<script>
  function leak() {

    $('<div/>')
      .html(new Array(1000).join('text'))
      .click(function() {})
      .appendTo('#data');

    document.getElementById('data').innerHTML = ''; // (*)
```

```
  }

  var interval = setInterval(leak, 10)
</script>

Утечка идёт...

<input type="button" onclick="clearInterval(interval)" value="stop" />
```

Утечка происходит потому, что обработчик события в jQuery хранится в данных элемента. В строке `(*)` элемент удален очисткой родительского `innerHTML`, но в `jQuery.cache` данные остались.

Более того, система обработки событий в jQuery устроена так, что вместе с обработчиком в данных хранится и ссылка на элемент, так что в итоге оба – и обработчик и элемент – остаются в памяти вместе со всем замыканием!

Ещё более простой пример утечки:

Этот код также создает утечку:

```
function leak() {
  $('<div/>')
    .click(function() {})
}
```

...То есть, мы создаём элемент, вешаем на него обработчик... И всё.

Такой код ведёт к утечке памяти как раз потому, что элемент `<div>` создан, но нигде не размещен :). После выполнения функции ссылка на него теряется. Но обработчик события `click` уже сохранил данные в `jQuery.cache`, которые застревают там навсегда.

Используем jQuery без утечек

Чтобы избежать утечек, описанных выше, для удаления элементов используйте функции jQuery API, а не чистый JavaScript.

Методы remove(), empty() и html() проверяют дочерние элементы на наличие данных и очищают их. Это несколько замедляет процедуру удаления, но зато освобождается память.

К счастью обнаружить такие утечки легко. Проверьте размер `$.cache`. Если он большой и растет, то изучите кэш, посмотрите, какие записи остаются и почему.

Улучшение производительности jQuery

У способа организации внутренних данных, применённого в jQuery, есть важный побочный эффект.

Функции, удаляющие элементы, также должны удалить и связанные с ними внутренние данные. Для этого нужно для каждого удаляемого элемента проверить – а нет ли чего во внутреннем хранилище? И, если есть – удалить.

Представим, что у нас есть большая таблица `<table>`, и мы хотим обновить её содержимое на новое. Вызов `$('table').html(новые данные)` перед вставкой новых данных аккуратно удалит старые: пробежит по всем ячейкам и проверит внутреннее хранилище.

Если это большая таблица, то обработчики, скорее всего, стоят не на ячейках, а на самом элементе `<table>`, то есть используется делегирование. А, значит, тратить время на проверку всех подэлементов ни к чему.

Но jQuery-то об этом не знает!

Чтобы «грязно» удалить элемент, без чистки, мы можем сделать это через «обычные» DOM-вызовы или воспользоваться методом detach(). Его официальное назначение – в том, чтобы убрать элемент из DOM, но сохранить возможность для вставки (и, соответственно, оставить на нём все данные). А неофициальное – быстро убрать элемент из DOM, без чистки.

Возможен и промежуточный вариант: никто не мешает сделать `elem.detach()` и поместить вызов `elem.remove()` в `setTimeout`. В результате очистка будет происходить асинхронно и незаметно.

Итого

- Утечки памяти при использовании jQuery возможны, если через DOM-методы удалять элементы, к которым привязаны данные или обработчики.
- Чтобы утечки не было, достаточно убедиться, что элемент удаляется при помощи методов jQuery.
- Побочный эффект – при удалении элементов jQuery должна проверить наличие данных для них. Это сильно замедляет процесс удаления большого поддерева DOM.
- Если мы знаем, что обработчиков и данных нет – гораздо быстрее удалять элементы при помощи вызова `detach` или обычного DOM.

Очистка памяти при removeChild/innerHTML

Управление памятью в случае с DOM работает по сути так же, как и с обычными JavaScript-объектами. Пока объект достижим – он остаётся в памяти.

Но есть и особенности, поскольку DOM весь переплетён ссылками.

Пример

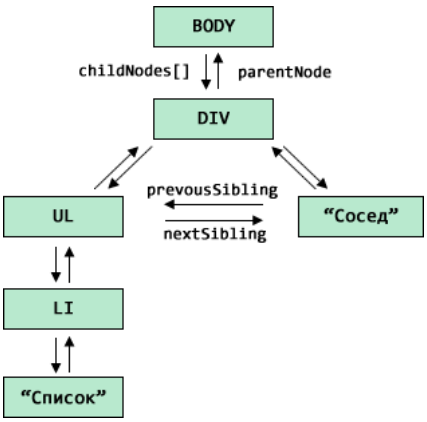
Для примера рассмотрим следующий HTML:

```
<html>

<body>
  <div>
    <ul>
      <li>Список</li>
    </ul>
    Сосед
  </div>
</body>

</html>
```

Его DOM (показаны только основные ссылки):



Удаление removeChild

Операция `removeChild` разрывает все связи удаляемым узлом и его родителем.

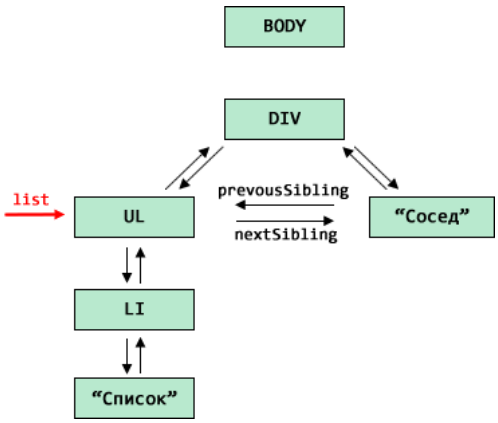
Поэтому, если удалить `DIV` из `BODY`, то всё поддерево под `DIV` станет недостижимым и будет удалено.

А что происходит, если на какой-то элемент внутри удаляемого поддерева есть ссылка?

Например, `UL` сохранён в переменную `list`:

```
var list = document.getElementsByTagName('UL')[0];
document.body.removeChild(document.body.children[0]);
```

В этом случае, так как из этого `UL` можно по ссылкам добраться до любого другого места DOM, то получается, что все объекты по-прежнему достижимы и должны остаться в памяти:



То есть, DOM-объекты при использовании `removeChild` работают по той же логике, что и обычные объекты.

Удаление через innerHTML

А вот удаление через очистку `elem.innerHTML="..."` браузеры интерпретируют по-разному.

По идее, при присвоении `elem.innerHTML=html` из DOM должны удаляться предыдущие узлы и добавляться новые, из указанного `html`. Но стандарт ничего не говорит о том, что делать с узлами после удаления. И тут разные браузеры имеют разное мнение.

Посмотрим, что произойдёт с DOM-структурой при очистке `BODY`, если на какой-либо элемент есть ссылка.

```
var list = document.getElementsByTagName('UL')[0];
document.body.innerHTML = "";
```

Обращаю внимание – связь разрывается только между `DIV` и `BODY`, т.е. на верхнем уровне, а `list` – это произвольный элемент.

Чтобы увидеть, что останется в памяти, а что нет – запустим код:

```
<div>
  <ul>
    <li>Список</li>
  </ul>
  Сосед
</div>

<script>
  var list = document.getElementsByTagName('ul')[0];
  document.body.innerHTML = ''; // удалили DIV

  alert( list.parentNode ); // цела ли ссылка UL -> DIV ?
  alert( list.nextSibling ); // живы ли соседи UL ?
  alert( list.children.length ); // живы ли потомки UL ?
</script>
```

Как ни странно, браузеры ведут себя по-разному:

	parentNode	nextSibling	children.length
Chrome/Safari/Opera	null	null	1
Firefox	узел DOM	узел DOM	1
IE 11-	null	null	0

Иными словами, браузеры ведут себя с различной степенью агрессивности по отношению к элементам.

Firefox

Главный пацифист. Оставляет всё, на что есть ссылки, т.е. элемент, его родителя, соседей и детей, в точности как при `removeChild`.

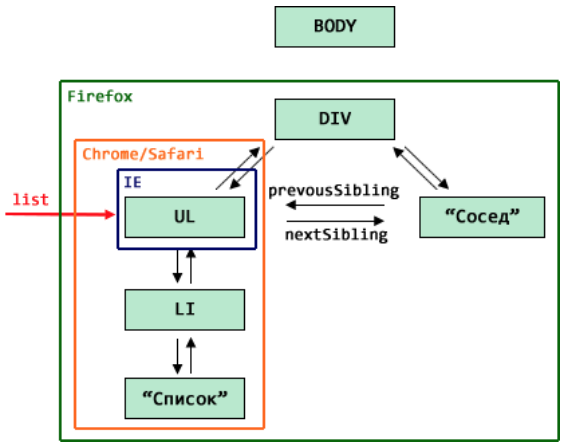
Chrome/Safari/Opera

Считают, что раз мы задали ссылку на `UL`, то нам нужно только это поддерево, а остальные узлы (соседей, родителей) можно удалить.

Internet Explorer

Как ни странно, самый агрессивный. Удаляет вообще всё, кроме узла, на который есть ссылка. Это поведение одинаково для всех версий IE.

На иллюстрации ниже показано, какую часть DOM оставит каждый из браузеров:



Итого

Если на какой-то DOM-узел есть ссылка, то:

- При использовании `removeChild` на родителе (или на этом узле, не важно) все узлы, достижимые из данного, остаются в памяти.

  То есть, фактически, в памяти может остаться большая часть дерева DOM. Это даёт наибольшую свободу в коде, но может привести к большим «утечкам памяти» из-за сохранения данных, которые реально не нужны.

- При удалении через `innerHTML` браузеры ведут себя с различной степенью агрессивности. Кросс-браузерно гарантировано одно: сам узел, на который есть ссылка, останется в памяти.

  Поэтому обращаться к соседям и детям узла, предок которого удалён через присвоение `innerHTML`, нельзя.

GCC: продвинутые оптимизации

Продвинутый режим оптимизации google closure compiler включается опцией `–compilation_level ADVANCED_OPTIMIZATIONS`.

Слово «продвинутый» (advanced) здесь, пожалуй, не совсем подходит. Было бы более правильно назвать его «супер-агрессивный-ломающий-ваш-неподготовленный-код-режим». Кардинальное отличие применяемых оптимизаций от обычных (simple) – в том, что они небезопасны.

Чтобы им пользоваться – надо уметь это делать.

Основной принцип продвинутого режима

- Если в обычном режиме переименовываются только локальные переменные внутри функций, то в «продвинутом» – на более короткие имена заменяется все.
- Если в обычном режиме удаляется недостижимый код после `return`, то в продвинутом – вообще весь код, который не вызывается в явном виде.

Например, если запустить продвинутую оптимизацию на таком коде:

```
// my.js
function test(node) {
  node.innerHTML = "newValue"
}
```

Строка запуска компилятора:

```
java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --js my.js
```

...То результат будет – пустой файл. Google Closure Compiler увидит, что функция `test` не используется, и с чистой совестью вырежет ее.

А в следующем скрипте функция сохранится:

```
function test(n) {
  alert( "this is my test number " + n );
}
test(1);
test(2);
```

После сжатия:

```
function a(b) {
  alert("this is my test number " + b)
}
a(1);
a(2);
```

Здесь в скрипте присутствует явный вызов функции, поэтому она сохранилась.

Конечно, есть способы, чтобы сохранить функции, вызов которых происходит вне скрипта, и мы их обязательно рассмотрим.

Продвинутый режим сжатия не предусматривает сохранения глобальных переменных. Он переименовывает, инлайнит, удаляет вообще все символы, кроме зарезервированных.

Иначе говоря, продвинутый режим (ADVANCED_OPTIMIZATIONS), в отличие от простого (SIMPLE_OPTIMIZATIONS – по умолчанию), вообще не заботится о доступности кода извне и сохранении ссылочной целостности относительно внешних скриптов.

Единственное, что он гарантирует – это внутреннюю ссылочную целостность, и то – при соблюдении ряда условий и практик программирования.

Собственно, за счет такого агрессивного подхода и достигается дополнительный эффект оптимизации и сжатия скриптов.

> То есть, продвинутый режим – это не просто «улучшенный обычный», а принципиально другой, небезопасный и обфусцирующий подход к сжатию.
>
> Этот режим является «фирменной фишкой» Google Closure Compiler, недоступной при использовании других компиляторов.

Для того, чтобы эффективно сжимать Google Closure Compiler в продвинутом режиме, нужно понимать, что и как он делает. Это мы сейчас обсудим.

Сохранение ссылочной целостности

Чтобы использовать сжатый скрипт, мы должны иметь возможность вызывать функции под теми именами, которые им дали.

То есть, перед нами стоит задача *сохранения ссылочной целостности*, которая заключается в том, чтобы обеспечить доступность нужных функций для обращений по исходному имени извне скрипта.

Существует два способа сохранения внешней ссылочной целостности: экстерны и экспорты. Мы в подробностях рассмотрим оба, но перед этим необходимо упомянуть о модулях – другой важнейшей возможности GCC.

Модули

При сжатии GCC можно указать одновременно много JavaScript-файлов. "Эка невидаль, " – скажете вы, и будете правы. Да, пока что ничего особого.

Но в дополнение к этому можно явно указать, какие исходные файлы сжать в какие файлы результата. То есть, разбить итоговую сборку на модули.

Так что страницы могут грузить модули по мере надобности. Например, по умолчанию – главный, а дополнительная функциональность – загружаться лишь там, где она нужна.

Для такой сборки используется флаг компилятора `--module имя:количество файлов`.

Например:

```
java -jar compiler.jar --js base.js --js main.js  --js admin.js --module
first:2 --module second:1:first
```

Эта команда создаст модули: first.js и second.js.

Первый модуль, который назван «first», создан из объединённого и оптимизированного кода первых двух файлов (`base.js` и `main.js`).

Второй модуль, который назван «second», создан из `admin.js` – это следующий аргумент `--js` после включенных в первый модуль.

Второй модуль в нашем случае зависит от первого. Флаг `--module second:1:first` как раз означает, что модуль `second` будет создан из одного файла после вошедших в предыдущий модуль (`first`) и зависит от модуля `first` .

А теперь – самое вкусное.

Ссылочная целостность между всеми получившимися файлами гарантируется.

Если в одном функция `doFoo` заменена на `b` , то и в другом тоже будет использоваться `b` .

Это означает, что проблем между JS-файлами не будет. Они могут свободно вызывать друг друга без экспорта, пока находятся в единой модульной сборке.

Экстерны

Экстерн (extern) – имя, которое числится в специальном списке компилятора. Он должен быть определен вне скрипта, в файле экстернов.

Компилятор никогда не переименовывает экстерны.

Например:

```
document.onkeyup = function(event) {
  alert(event.type)
}
```

После продвинутого сжатия:

```
document.onkeyup = function(a) {
  alert(a.type)
}
```

Как видите, переименованной оказалась только переменная `event` . Такое переименование заведомо безопасно, т.к. `event` – локальная переменная.

Почему компилятор не тронул остального? Попробуем другой вариант:

```
document.blabla = function(event) {
  alert(event.megaProperty)
}
```

После компиляции:

```
document.a = function(a) {
  alert(a.b)
}
```

Теперь компилятор переименовал и `blabla` и `megaProperty` .

Дело в том, что названия, использованные до этого, были во внутреннем списке экстернов компилятора. Этот список охватывает основные объекты браузеров и находится (под именем `externs.zip`) в корне архива `compiler.jar` .

Компилятор переименовывает имя списка экстернов только когда так названа локальная переменная.

Например:

```
window.resetNode = function(node) {
  var innerHTML = "test";
  node.innerHTML = innerHTML;
}
```

На выходе:

```
window.a = function(a) {
  a.innerHTML = "test"
};
```

Как видите, внутренняя переменная `innerHTML` не просто переименована – она заинлайнена (заменена на значение). Так как переменная локальна, то любые действия внутри функции с ней безопасны.

А свойство `innerHTML` не тронуто, как и объект `window` – так как они в списке экстернов и не являются локальными переменными.

Это приводит к следующему побочному эффекту. Иногда свойства, которые следовало бы сжать, не сжимаются. Например:

```
window['User'] = function(name, type, age) {
  this.name = name
  this.type = type
  this.age = age
}
```

После сжатия:

```
window.User = function(a, b, c) {
  this.name = a;
  this.type = b;
  this.a = c
};
```

Как видно, свойство `age` сжалось, а `name` и `type` – нет. Это побочный эффект экстернов: `name` и `type` – в списке объектов браузера, и компилятор просто старается не наломать дров.

Поэтому отметим еще одно полезное правило оптимизации:

Названия своих свойств не должны совпадать с зарезервированными словами (экстернами). Тогда они будут хорошо сжиматься.

Для задания списка экстернов их достаточно перечислить в файле и указать этот файл флагом `–externs <файл экстернов.js>` .

При перечислении объектов в файле экстернов – объявляйте их и перечисляйте свойства. Все эти объявления никуда не идут, они используются только для создания списка, который обрабатывается компилятором.

Например, файл `myexterns.js` :

```
var dojo = {}
dojo._scopeMap;
```

Использование такого файла при сжатии (опция `–externs myexterns.js`) приведет к тому, что все обращения к символам `dojo` и к `dojo._scopeMap` будут не сжаты, а оставлены «как есть».

Экспорт

Экспорт – программный ход, основанный на следующем правиле поведения компилятора.

Компилятор заменяет обращения к свойствам через кавычки на точку, и при этом не трогает название свойства.

Например, `window[„User“]` превратится в `window.User` , но не дальше.

Таким образом можно *«экспортировать»* нужные функции и объекты:

```
function SayWidget(elem) {
  this.elem = elem
  this.init()
}
window['SayWidget'] = SayWidget;
```

На выходе:

```
function a(b) {
  this.a = b;
  this.b()
}
window.SayWidget = a;
```

Обратим внимание – сама функция `SayWidget` была переименована в `a` . Но затем – экспортирована как `window.SayWidget` , и таким образом доступна внешним скриптам.

Добавим пару методов в прототип:

```
function SayWidget(elem) {
  this.elem = elem;
  this.init();
}

SayWidget.prototype = {
  init: function() {
    this.elem.style.display = 'none'
  },

  setSayHandler: function() {
    this.elem.onclick = function() {
      alert("hi")
    };
  }
}

window['SayWidget'] = SayWidget;
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler;
```

После сжатия:

```
function a(b) {
  this.a = b;
  this.b()
}
a.prototype = {b:function() {
  this.a.style.display = "none"
}, c:function() {
  this.a.onclick = function() {
    alert("hi")
  }
```

```
}};
window.SayWidget = a;
a.prototype.setSayHandler = a.prototype.c;
```

Благодаря строке

```
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler
```

метод `setSayHandler` экспортирован и доступен для внешнего вызова.

Сама строка экспорта выглядит довольно глупо. По виду – присваиваем свойство самому себе.

Но логика сжатия GCC работает так, что такая конструкция является экспортом. Справа переименование свойства `setSayHandler` происходит, а слева – нет.

Планируйте жизнь после сжатия

Рассмотрим следующий код:

```
window['Animal'] = function() {
  this.blabla = 1;
  this['blabla'] = 2;
}
```

После сжатия:

```
window.Animal = function() {
  this.a = 1;
  this.blabla = 2
};
```

Как видно, первое обращение к свойству `blabla` сжалось, а второе (как и все аналогичные) – преобразовалось в синтаксис через точку. В результате получили некорректное поведение кода.

Так что, используя продвинутый режим оптимизации, планируйте поведение кода после сжатия.

Если где-то возможно обращение к свойствам через квадратные скобки по полному имени – такое свойство должно быть экспортировано.

goog.exportSymbol и goog.exportProperty

В библиотеке Google Closure Library для экспорта есть специальная функция `goog.exportSymbol`. Вызывается так:

```
goog.exportSymbol('my.SayWidget', SayWidget)
```

Эта функция по сути работает также, как и рассмотренная выше строка с присвоением свойства, но при необходимости создает нужные объекты.

Она аналогична коду:

```
window['my'] = window['my'] || {}
window['my']['SayWidget'] = SayWidget
```

То есть, если путь к объекту не существует – `exportSymbol` создаст нужные пустые объекты.

Функция `goog.exportProperty` экспортирует свойство объекта:

```
goog.exportProperty(SayWidget.prototype, 'setSayHandler', SayWidget.prototype.setSayHandler)
```

Строка выше – то же самое, что и:

```
SayWidget.prototype['setSayHandler'] = SayWidget.prototype.setSayHandler
```

Зачем они нужны, если все можно сделать простым присваиванием?

Основная цель этих функций – во взаимодействии с Google Closure Compiler. Они дают информацию компилятору об экспортах, которую он может использовать.

Например, есть недокументированная внутренняя опция `externExportsPath`, которая генерирует из всех экспортов файл экстернов. Таким образом можно распространять откомпилированный JavaScript-файл как внешнюю библиотеку, с файлом экстернов для удобного внешнего связывания.

Кроме того, экспорт через эти функциями удобен и нагляден.

Если вы используете продвинутый режим оптимизации, то можно взять их из файла base.js Google Closure Library. Можно и подключить этот файл целиком – оптимизатор при продвинутом сжатии вырежет из него почти всё лишнее, так что overhead будет минимальным.

Отличия экспорта от экстерна

Между экспортом и экстерном есть кое-что общее. И то и другое дает возможность доступа к объектам под исходным именем, до переименования.

Но, в остальном, это совершенно разные вещи.

Экстерн	Экспорт
Служит для тотального запрета на переименование всех обращений к свойству. Задумано для сохранения обращений к стандартным объектам браузера, внешним библиотекам.	Служит для открытия доступа к свойству извне под указанным именем. Задумано для открытия внешнего интерфейса к сжатому скрипту.
Работает со свойством, объявленным вне скрипта. Вы не можете объявить новое свойство в скрипте и сделать его экстерном.	Создает ссылку на свойство, объявленное в скрипте.
Если `window` - экстерн, то все обращения к `window` в скрипте останутся как есть.	Если `user` экспортируется, то создается только одна ссылка под полным именем, а все остальные обращения будут сокращены.

Стиль разработки

Посмотрим, как сжиматель поведёт себя на следующем, типичном, объявлении библиотеки:

```
(function(window, undefined) {

  // пространство имен и локальная переменная для него
  var MyFramework = window.MyFramework = {};

  // функция фреймворка, доступная снаружи
  MyFramework.publicOne = function() {
    makeElem();
  };

  // приватная функция фреймворка
  function makeElem() {
    var div = document.createElement('div');
    document.body.appendChild(div);
  }

  // еще какая-то функция
  MyFramework.publicTwo = function() {};

})(window);

// использование
MyFramework.publicOne();
```

Результат компиляции в обычном режиме:

```
// java -jar compiler.jar --js myframework.js --formatting PRETTY_PRINT
(function(a) {
  a = a.MyFramework = {};
  a.publicOne = function() {
    var a = document.createElement("div");
    document.body.appendChild(a)
  };
  a.publicTwo = function() {
  }
})(window);
MyFramework.publicOne();
```

Это – примерно то, что мы ожидали. Неиспользованный метод `publicTwo` остался, локальные свойства переименованы и заинлайнены.

А теперь продвинутый режим:

```
// --compilation_level ADVANCED_OPTIMIZATIONS
window.a = {};
MyFramework.b();
```

Оно не работает! Компилятор попросту не разобрался, что и как вызывается, и превратил рабочий JS-файл в один сплошной баг.

В зависимости от версии GCC у вас может быть и что-то другое.

Всё дело в том, что такой стиль объявления нетипичен для инструментов, которые в самом Google разрабатываются и сжимаются этим минификатором.

Типичный правильный стиль:

```
// пространство имен и локальная переменная для него
var MyFramework = {};

MyFrameWork._makeElem = function() {
  var div = document.createElement('div');
  document.body.appendChild(div);
};

MyFramework.publicOne = function() {
  MyFramework._makeElem();
};

MyFramework.publicTwo = function() {};

// использование
MyFramework.publicOne();
```

Обычное сжатие здесь будет бесполезно, а вот продвинутый режим идеален:

```
// в зависимости от версии GCC результат может отличаться
MyFrameWork.a = function() {
  var a = document.createElement("div");
  document.body.appendChild(a)
};
MyFrameWork.a();
```

Google Closure Compiler не только разобрался в структуре и удалил лишний метод – он заинлайнил функции, чтобы итоговый размер получился минимальным.

Как говорится, преимущества налицо.

Резюме

Продвинутый режим оптимизации сжимает, оптимизирует и, при возможности, удаляет все свойства и методы, за исключением экстернов.

Это является принципиальным отличием, по сравнению с другими упаковщиками.

Отказ от сохранения внешней ссылочной целостности с одной стороны позволяет увеличить уровень сжатия, но требует поддержки со стороны разработчика.

Основная проблема этого сжатия – усложнение разработки. Добавляется дополнительный уровень возможных проблем: сжатие. Конечно, можно отлаживать и сжатый код, для этого придуманы Source Maps, но клиентская разработка и без того достаточно сложна.

Поэтому его используют редко.

Как правило, есть две причины для использования продвинутого режима:

1. Обфускация кода.

   Если в коде после обычного сжатия ещё как-то можно разобраться, то после продвинутого – уже нет. Всё переименовано и заинлайнено. В теории это, конечно, возможно, но «порог входа» в такой код несоизмеримо выше.

   Судя по виду скриптов на сайтах, созданных Google, сам Google жмет свои скрипты именно продвинутым режимом оптимизации. И библиотека Google Closure Library тоже рассчитана на него.

2. Хорошие сжатие виджетов, счётчиков.

   Небольшой код, который отдаётся наружу, может быть сжат в продвинутом режиме. Так как он небольшой – все ошибки можно легко исправить, а продвинутый режим гарантирует наилучшее сжатие.

GCC: статическая проверка типов

Google Closure Compiler, как и любой кошерный компилятор, старается проверить правильность кода и предупредить о возможных ошибках.

Первым делом он, разумеется, проверяет структуру кода и сразу же выдает такие ошибки как пропущенная скобка или лишняя запятая.

Но, кроме этого, он умеет проверять типы переменных, используя как свои собственные знания о встроенных javascript-функциях и преобразованиях типов, так и информацию о типах из JSDoc, указываемую javascript-разработчиком.

Это обеспечивает то, чем так гордятся компилируемые языки – статическую проверку типов, что позволяет избежать лишних ошибок во время выполнения.

Для вывода предупреждений при проверки типов используется флаг `--jscomp_warning checkTypes`.

Задание типа при помощи аннотации

Самый очевидный способ задать тип – это использовать аннотацию. Полный список аннотаций вы найдете в документации.

В следующем примере параметр `id` функции `f1` присваивается переменной `boolVar` другого типа:

```
/** @param {number} id */
function f(id) {
  /** @type {boolean} */
  var boolVar;

  boolVar = id; // (!)
}
```

Компиляция с флагом `--jscomp_warning checkTypes` выдаст предупреждение:

```
f.js:6: WARNING - assignment
found   : number
required: boolean
        boolVar = id; // (!)
        ^
```

Действительно: произошло присвоение значения типа `number` переменной типа `boolean`.

Типы отслеживаются по цепочке вызовов.

Еще пример, на этот раз вызов функции с некорректным параметром:

```
/** @param {number} id */
function f1(id) {
  f2(id); // (!)
}

/** @param {string} id */
function f2(id) {}
```

Такой вызов приведёт к предупреждению со стороны минификатора:

```
f2.js:3: WARNING - actual parameter 1 of f2 does not match formal parameter
found   : number
required: string
  f2(id); // (!)
     ^
```

Действительно, вызов функции `f2` произошел с числовым типом вместо строки.

Отслеживание приведений и типов идёт при помощи графа взаимодействий и выведению (infer) типов, который строит GCC по коду.

Знания о преобразовании типов

Google Closure Compiler знает, как операторы javascript преобразуют типы. Такой код уже не выдаст ошибку:

```
/** @param {number} id */
function f1(id) {
  /** @type {boolean} */
  var boolVar;

  boolVar = !!id
}
```

Действительно – переменная преобразована к типу boolean двойным оператором НЕ. А код `boolVar = „test-“+id` выдаст ошибку, т.к. конкатенация со строкой дает тип `string`.

Знание о типах встроенных функций, объектные типы

Google Closure Compiler содержит описания большинства встроенных объектов и функций javascript вместе с типами параметров и результатов.

Например, объектный тип `Node` соответствует узлу DOM.

Пример некорректного кода:

```
/** @param {Node} node */
function removeNode(node) {
  node.parentNode.removeChild(node)
}
document.onclick = function() {
  removeNode("123")
}
```

Выдаст предупреждение

```
f3.js:7: WARNING - actual parameter 1 of removeNode does not match formal parameter
found   : string
required: (Node|null)
  removeNode("123")
             ^
```

Обратите внимание – в этом примере компилятор выдает `required: Node|null`. Это потому, что указание объектного типа (не элементарного) подразумевает, что в функцию может быть передан `null`.

В следующем примере тип указан жестко, без возможности обнуления:

```
/** @param {!Node} node */
function removeNode(node) {
  node.parentNode.removeChild(node)
}
```

Восклицательный знак означает, что параметр обязателен.

Найти описания встроенных типов и объектов javascript вы можете в файле экстернов: `externs.zip` находится в корне архива `compiler.jar`.

Интеграция с проверками типов из Google Closure Library

В Google Closure Library есть функции проверки типов: `goog.isArray`, `goog.isDef`, `goog.isNumber` и т.п.

Google Closure Compiler знает о них и понимает, что внутри следующего `if` переменная может быть только функцией:

```
var goog = {
  isFunction: function(f) {
    return typeof f == 'function'
  }
}

if (goog.isFunction(func)) {
  func.apply(1, 2)
}
```

Сжатие с проверкой выдаст предупреждение:

```
f.js:6: WARNING - actual parameter 2 of Function.apply does not match formal parameter
found   : number
required: (Object|null|undefined)
  func.apply(1, 2)
                ^      ^
```

То есть, компилятор увидел, что код, использующий `func` находится в `if (goog.isFunction(func))` и сделал соответствующий вывод, что в этой ветке `func` является функцией, а значит вызов `func.apply(1,2)` ошибочен (второй аргумент не может быть числом).

Дело тут именно в интеграции с Google Closure Library. Если поменять `goog` на `g` – предупреждения не будет.

Резюме

Из нескольких примеров, которые мы рассмотрели, должна быть понятна общая логика проверки типов.

Соответствующие различным типам и ограничениям на типы аннотации вы можете найти в Документации Google. В частности, возможно указание нескольких возможных типов, типа `undefined` и т.п.

Также можно указывать количество и тип параметров функции, ключевого слова `this` , объявлять классы, приватные методы и интерфейсы.

Проверка типов javascript, предоставляемая Google Closure Compiler – пожалуй, самая продвинутая из существующих на сегодняшний день.

С ней аннотации, документирующие типы и параметры, становятся не просто украшением, а реальным средством проверки, уменьшающим количество ошибок на production.

Очень подробно проверка типов описана в книге Closure: The Definitive Guide, автора Michael Bolin.

GCC: интеграция с Google Closure Library

Google Closure Compiler содержит ряд специальных возможностей для интеграции с Google Closure Library.

Здесь важны две вещи.

1. Для их использования возможно использовать минимум от Google Closure Library. Например, взять одну или несколько функций из библиотеки.
2. GCC – расширяемый компилятор, можно добавить к нему свои «фазы оптимизации» для интеграции с другими инструментами и фреймворками.

Интеграция с Google Closure Library подключается флагом `–process_closure_primitives` , который по умолчанию установлен в `true` . То есть, она включена по умолчанию.

Этот флаг запускает специальный проход компилятора, описанный классом `ProcessClosurePrimitives` и подключает дополнительную проверку типов `ClosureReverseAbstractInterpreter` .

Мы рассмотрим все действия, которые при этом происходят, а также некоторые опции, которые безопасным образом используют символы Google Closure Library без объявления флага.

Преобразование основных символов

Следующие действия описаны в классе `ProcessClosurePrimitives` .

Замена константы `COMPILED`

В Google Closure Library есть переменная:

```
/**
 * @define {boolean} ...
 */
var COMPILED = false;
```

Проход `ProcessClosurePrimitives` переопределяет ее в `true` и использует это при оптимизациях, удаляя ветки кода, не предназначены для запуска на production.

Такие функции существуют, например, в ядре Google Closure Library. К ним в первую очередь относятся вызовы, предназначенные для сборки и проверки зависимостей. Они содержат код, обрамленный проверкой `COMPILED` , например:

```
goog.require = function(rule) {
  // ...
  if (!COMPILED) {
    // основное тело функции
  }
}
```

Аналогично может поступить и любой скрипт, даже без использования Google Closure Library:

```
/** @define {boolean} */
var COMPILED = false

Framework = {}

Framework.sayCompiled = function() {
  if (!COMPILED) {
    alert("Not compressed")
  } else {
    alert("Compressed")
  }
}
```

Для того, чтобы сработало, нужно сжать в продвинутом режиме:

```
Framework = {};
Framework.sayCompiled = Framework.a = function() {
  alert( "Compressed" );
};
```

Компилятор переопределил `COMPILED` в `true` и произвел соответствующие оптимизации.

Автоподстановка локали

В Google Closure Compiler есть внутренняя опция `locale`

Эта опция переопределяет переменную `goog.LOCALE` на установленную при компиляции.

Для использования опции `locale`, на момент написания статьи, ее нужно задать в Java коде компилятора, т.к. соответствующего флага нет.

Как и `COMPILED`, константу `goog.LOCALE` можно использовать в своем коде без библиотеки Google Closure Library.

Проверка зависимостей

Директивы `goog.provide`, `goog.require`, `goog.addDependency` обрабатываются особым образом.

Все зависимости проверяются, а сами директивы проверки – удаляются из сжатого файла.

Экспорт символов

Вызов `goog.exportSymbol` задаёт экспорт символа.

Если подробнее, то код `goog.exportSymbol(„a“,myVar)` эквивалентен `window['a'] = myVar`.

Автозамена классов CSS

Google Closure Library умеет преобразовывать классы CSS на более короткие по списку, который задаётся при помощи `goog.setCssNameMapping`.

Например, следующая функция задает такой список.

```
goog.setCssNameMapping({
   "goog-menu": "a",
   "goog-menu-disabled": "a-b",
   "CSS_LOGO": "b",
   "hidden": "c"
 });
```

Тогда следующий вызов преобразуется в «a a-b»:

```
goog.getCssName('goog-menu') + ' ' + goog.getCssName('goog-menu', 'disabled')
```

Google Closure Compiler производит соответствующие преобразования в сжатом файле и удаляет вызов `setCssNameMapping` из кода.

Чтобы это сжатие работало, в HTML/CSS классы тоже должны сжиматься. По всей видимости, в приложениях Google это и происходит, но соответствующие инструменты закрыты от публики.

Генерация списка экстернов

При объявлении внутренней опции `externExportsPath`, содержащей путь к файлу, в этот файл будут записаны все экспорты, описанные через `goog.exportSymbol` / `goog.exportProperty`.

В дальнейшем этот файл может быть использован как список экстернов для компиляции.

Эта опция может быть полезна для создания внешних библиотек, распространяемых со списком экстернов.

Для её использования нужна своя обёртка вокруг компилятора на Java. Соответствующий проход компилятора описан в классе `ExternExportsPass`.

Проверка типов

В Google Closure Library есть ряд функций для проверки типов. Например: `goog.isArray`, `goog.isString`, `goog.isNumber`, `goog.isDef` и т.п.

Компилятор использует их для проверки типов, более подробно см. GCC: статическая проверка типов

Эта логика описана в классе `ClosureReverseAbstractInterpreter`. Названия функций, определяющих типы, жестко прописаны в Java-коде, поменять их на свои без модификации исходников нельзя.

Автогенерация экспортов из аннотаций

Для этого в Google Closure Compiler есть внутренняя опция `generateExports`.

Эта недокументированная опция добавляет проход компилятора, описанный классом `GenerateExports`.

Он читает аннотации `@export` и создает из них экспортирующие вызовы `goog.exportSymbol/exportProperty`. Название экспортирующих функций находится в классе соглашений кодирования, каким по умолчанию является `GoogleCodingConvention`.

Например:

```
/** @export */
function Widget() {}
  /** @export */
Widget.prototype.hide = function() {
  this.elem.style.display = 'none'
}
```

После компиляции в продвинутом режиме:

```
function a() {}
goog.d("Widget", a);
a.prototype.a = function() {
  this.b.style.display = "none"
};
goog.c(a.prototype, "hide", a.prototype.a);
```

Свойства благополучно экспортированы. Удобно.

Резюме

Google Closure Compiler содержит дополнительные фичи, облегчающие интеграцию с Google Closure Library. Некоторые из них весьма полезны, но требуют создания своего Java-файла, который ставит внутренние опции.

При обработке символов компилятор не смотрит, подключена ли библиотека, он находит обрабатывает их просто по именам. Поэтому вы можете использовать свою реализацию соответствующих функций.

Google Closure Compiler можно легко расширить, добавив свои опции и проходы оптимизатора, для интеграции с вашими инструментами.

Окна и Фреймы

Открытие окон и методы window

Всплывающее окно («попап» – от англ. Popup window) – один из старейших способов показать пользователю ещё один документ.

В этой статье мы рассмотрим открытие окон и ряд тонких моментов, которые с этим связаны.

Простейший пример:

```
window.open("http://ya.ru");
```

...При запуске откроется новое окно с указанным URL.

Большинство браузеров по умолчанию создают новую вкладку вместо отдельного окна, но чуть далее мы увидим, что можно и «заказать» именно окно.

Блокировщик всплывающих окон

Рекламные попапы очень надоели посетителям, аж со времён 20-го века, поэтому современные браузеры всплывающие окна обычно блокируют. При этом пользователь, конечно, может изменить настройки блокирования для конкретного сайта.

Всплывающее окно блокируется в том случае, если вызов **`window.open`** произошёл не в результате действия посетителя.

Как же браузер понимает – посетитель вызвал открытие окна или нет?

Для этого при работе скрипта он хранит внутренний «флаг», который говорит – инициировал посетитель выполнение или нет. Например, при клике на кнопку весь код, который выполнится в результате, включая вложенные вызовы, будет иметь флаг «инициировано посетителем» и попапы при этом разрешены.

А если код был на странице и выполнился автоматически при её загрузке – у него этого флага не будет. Попапы будут заблокированы.

Полный синтаксис window.open

Полный синтаксис:

```
win = window.open(url, name, params)
```

Функция возвращает ссылку на объект `window` нового окна, либо `null`, если окно было заблокировано браузером.

Параметры:

`url`

URL для загрузки в новое окно.

`name`

Имя нового окна. Может быть использовано в параметре `target` в формах. Если позднее вызвать `window.open()` с тем же именем, то браузеры (кроме IE) заменяют существующее окно на новое.

params

Строка с конфигурацией для нового окна. Состоит из параметров, перечисленных через запятую. Пробелов в ней быть не должно.

Значения параметров `params`.

1. Настройки расположения окна:

left/top (число)

Координаты верхнего левого угла относительно экрана. Ограничение: новое окно не может быть позиционированно за пределами экрана.

width/height (число)

Ширина/высота нового окна. Минимальные значения ограничены, так что невозможно создать невидимое окно с нулевыми размерами.

Если координаты и размеры не указаны, то обычно браузер открывает не окно, а новую вкладку.

2. Свойства окна:

menubar (yes/no)

Скрыть или показать строку меню браузера.

toolbar (yes/no)

Показать или скрыть панель навигации браузера (кнопки назад, вперед, обновить страницу и остальные) в новом окне.

location (yes/no)

Показать/скрыть поле URL-адреса в новом окне. По умолчанию Firefox и IE не позволяют скрывать строку адреса.

status (yes/no)

Показать или скрыть строку состояния. С другой стороны, браузер может в принудительном порядке показать строку состояния.

resizable (yes/no)

Позволяет отключить возможность изменять размеры нового окна. Значение `no` обычно неудобно посетителям.

scrollbars (yes/no)

Разрешает убрать полосы прокрутки для нового окна. Значение `no` обычно неудобно посетителям.

3. Еще есть небольшое количество не кросс-браузерных свойств, которые обычно не используются. Вы можете узнать о них в документации, например MDN: window.open.

> ⚠️ Важно:
>
> Браузер подходит к этим параметрам интеллектуально. Он может проигнорировать их часть или даже все, они скорее являются «пожеланиями», нежели «требованиями».

Важные моменты:

- Если при вызове `open` указан только первый параметр, параметр отсутствует, то используются параметры по умолчанию. Обычно при этом будет открыто не окно, а вкладка, что зачастую более удобно.
- Если указана строка с параметрами, но некоторые `yes/no` параметры отсутствуют, то браузер выставляет их в `no`. Поэтому убедитесь, что все нужные вам параметры выставлены в `yes`.
- Когда не указан `top/left`, то браузер откроет окно с небольшим смещением относительно левого верхнего угла последнего открытого окна.
- Если не указаны `width/height`, новое окно будет такого же размера, как последнее открытое.

Доступ к новому окну

Вызов `window.open` возвращает ссылку на новое окно. Она может быть использована для манипуляции свойствами окна, изменения URL, доступа к его переменным и т.п.

В примере ниже мы заполняем новое окно содержимым целиком из JavaScript:

```
var newWin = window.open("about:blank", "hello", "width=200,height=200");
newWin.document.write("Привет, мир!");
```

А здесь модифицируем содержимое после загрузки:

```
var newWin = window.open('/', 'example', 'width=600,height=400');
alert(newWin.location.href); // (*) about:blank, загрузка ещё не началась
newWin.onload = function() {
```

```
  // создать div в документе нового окна
  var div = newWin.document.createElement('div'),
      body = newWin.document.body;

  div.innerHTML = 'Добро пожаловать!'
  div.style.fontSize = '30px'

  // вставить первым элементом в body нового окна
  body.insertBefore(div, body.firstChild);
}
```

Обратим внимание: сразу после `window.open` новое окно ещё не загружено. Это демонстрирует `alert` в строке `(*)`. Поэтому в примере выше окно модифицируется при `onload`. Можно было и поставить обработчик на `DOMContentLoaded` для `newWin.document`.

Связь между окнами – двухсторонняя.

Родительское окно получает ссылку на новое через `window.open`, а дочернее – ссылку на родителя `window.opener`.

Оно тоже может его модифицировать.

Если запустить пример ниже, то новое окно заменит содержимое текущего на `'Test'`:

```
var newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(
  "<script>window.opener.document.body.innerHTML = 'Test'</script>"
);
```

Same Origin Policy – защита проверкой протокол-сайт-порт

Большинство действий, особенно получение содержимого окна и его переменных, возможны лишь в том случае, если URL нового окна происходит из того же источника (англ. – *«Same Origin»*), т.е. совпадают домен, протокол и порт.

Иначе говоря, если новое окно содержит документ с того же сайта.

Больше информации об этом будет позже, в главе Кросс-доменные ограничения и их обход.

События

Наиболее важные события при работе с окном браузера:

- `onresize` – событие изменения размера окна.
- `onscroll` – событие при прокрутке окна.
- `onload` – полностью загрузилась страница со всеми ресурсами.
- `onfocus/onblur` – получение/потеря фокуса.

Методы и свойства

`window.closed`

Свойство `window.closed` равно `true`, если окно закрыто. Может быть использовано, чтобы проверить, закрыл ли посетитель попап.

`window.close()`

Закрывает попап без предупреждений и уведомлений. Вообще, метод `close()` можно вызвать для любого окна, в том числе, текущего. Но если окно открыто не с помощью `window.open()`, то браузер может проигнорировать вызов `close` или запросить подтверждение.

Перемещение и изменение размеров окна

Существует несколько методов для перемещения/изменения размеров окна.

`win.moveBy(x,y)`

Перемещает окно относительно текущего положения на `x` пикселей вправо и `y` пикселей вниз. Допускаются отрицательные значения.

`win.moveTo(x,y)`

Передвигает окно в заданную координатами `x` и `y` точку экрана монитора.

`win.resizeBy(width,height)`

Изменяет размер окна на заданную величину `width/height` (ширина/высота). Допускаются отрицательные значения.

`win.resizeTo(width,height)`

Изменяет размер окна на заданное значение.

Важно:

Чтобы предотвратить использование этих методов с плохими целями, браузеры часто блокируют их выполнение. Как правило, они работают, если окно `win` открыто вызовом window.open из JavaScript текущей страницы и в нём нет дополнительных вкладок.

Ни свернуть ни развернуть

Заметим, что JavaScript не может ни свернуть ни развернуть ни «максимизировать» (Windows) окно.

Эти функции операционной системы от Frontend-разработчиков скрыты. Вызовы, описанные выше, в случае свёрнутого или максимизированного окна не работают.

Прокрутка окна

Прокрутка окна требуется, пожалуй, чаще всего. Мы уже говорили о ней в главе Размеры и прокрутка страницы:

`win.scrollBy(x,y)`

Прокрутка окна на заданное число пикселей вперед или назад. Допускаются отрицательные значения.

`win.scrollTo(x,y)`

Прокручивает окно к заданным координатам.

`elem.scrollIntoView(top)`

Этот метод прокрутки вызывается на элементе. При этом окно прокручивается так, чтобы элемент был полностью видим. Если параметр `top` равен `true` или не задан, то верх элемента совпадает с верхом окна. Если он равен `false` , то окно прокручивается так, чтобы нижний край элемента совпал с нижним краем окна.

Итого

- Всплывающее окно открывается с помощью вызова `window.open(url, name, params)` .
- Метод `window.open` возвращает ссылку на новое окно или `null` , если окно было заблокировано.
- Современные браузеры блокируют окна, если `window.open` вызвано не в результате действия посетителя.
- Обычно открывается вкладка, но если заданы размеры и позиция – то именно окно.
- Новое окно имеет ссылку на родительское в `window.opener` .
- Окна могут общаться между собой как угодно, если они из одного источника. Иначе действуют жёсткие ограничения безопасности.

Всплывающие окна используются нечасто. Ведь загрузить новую информацию можно динамически, с помощью технологии AJAX, а показать – в элементе `<div>` , расположенным над страницей (`z-index`). Ещё одна альтернатива – тег `<iframe>` .

Но в некоторых случаях всплывающие окна бывают очень даже полезны. Например, отдельное окно сервиса онлайн-консультаций. Посетитель может ходить по сайту в основном окне, а общаться в чате – во вспомогательном.

Если вы хотите использовать всплывающее окно, предупредите посетителя об этом, так же и при использовании `target="_blank"` в ссылках или формах. Иконка открывающегося окошка на ссылке поможет посетителю понять, что происходит и не потерять оба окна из поля зрения.

Общение между окнами и фреймами

Элемент `iframe` является обычным узлом DOM, как и любой другой. Существенное отличие – в том, что с ним связан объект `window` внутреннего окна. Он доступен по ссылке `iframe.contentWindow` .

Таким образом, `iframe.contentWindow.document` будет внутренним документом, `iframe.contentWindow.document.body` – его `<body>` и так далее.

Когда-то...

В старых браузерах использовались другие свойства, такие как `iframe.contentDocument` и даже `iframe.document` , но они давно не нужны.

Переход внутрь ифрейма

В примере ниже JavaScript получает документ внутри ифрейма и модифицирует его:

```
<iframe src="javascript:'тест'" style="height:60px"></iframe>

<script>
  var iframe = document.getElementsByTagName('iframe')[0];

  var iframeDoc = iframe.contentWindow.document;

  if (iframeDoc.readyState == 'complete') {
    iframeDoc.body.style.backgroundColor = 'green';
  }
  iframe.onload = function() {
    var iframeDoc2 = iframe.contentWindow.document;
    iframeDoc2.body.style.backgroundColor = 'orange';
  }
</script>
```

src=„javascript:«текст»“

Атрибут `src` может использовать протокол `javascript`, как указано выше: `src="javascript:код"`. При этом код выполняется и его результат будет содержимым ифрейма. Этот способ описан в стандарте и поддерживается всеми браузерами.

Атрибут `src` является обязательным, и его отсутствие может привести к проблемам, вплоть до игнорирования ифрейма браузером. Чтобы ничего не загружать в ифрейм, можно указать пустую строку: `src="javascript:''"` или специальную страницу: `src="about:blank"`.

В некоторых браузерах (Chrome) пример выше покажет `iframe` зелёным. А в некоторых (Firefox) – оранжевым.

Дело в том, что, когда `iframe` только создан, документ в нём обычно ещё не загружен.

При обычных значениях `iframe src="..."`, которые указывают на HTML-страницу (даже если она уже в кеше), это всегда так. Документ, который в `iframe` на момент срабатывания скрипта `iframeDoc` – временный, он будет заменён на новый очень скоро. И работать надо уже с новым документом `iframeDoc2` – например, по событию `iframe.onload`.

В случае с `javascript`-протоколом, по идее, ифрейм уже загружен, и тогда `onload` у него уже не будет. Но здесь мнения браузеров расходятся, некоторые (Firefox) всё равно «подгрузят» документ позже. Поэтому факт «готовности» документа в скрипте проверяется через `iframeDoc.readyState`.

Ещё раз заметим, что при обычных URL в качестве `src` нужно работать не с начальным документом, а с тем, который появится позже.

Кросс-доменность: ограничение доступа к окну

Элемент `<iframe>` является «двуличным». С одной стороны, это обычный узел DOM, с другой – внутри находится окно, которое может иметь совершенно другой URL, содержать независимый документ из другого источника.

Внешний документ имеет полный доступ к `<iframe>` как к DOM-узлу. А вот к окну – если они с одного источника.

Это приводит к забавным последствиям. Например, чтобы узнать об окончании загрузки `<iframe>`, мы можем повесить обработчик `iframe.onload`. По сути, это то же самое что `iframe.contentWindow.onload`, но его мы можем поставить лишь в случае, если окно с того же источника.

```
<iframe src="https://example.com" style="height:100px"></iframe>

<script>
  var iframe = document.getElementsByTagName('iframe')[0];

  // сработает
  iframe.onload = function() {
    alert( "iframe onload" );
  };

  // не сработает
  iframe.contentWindow.onload = function() {
    alert( "contentWindow onload" );
  };
</script>
```

Если бы в примере выше `<iframe src>` был с текущего сайта, то оба обработчика сработали бы.

Иерархия window.frames

Альтернативный способ доступа к окну ифрейма – это получить его из коллекции `window.frames`.

Есть два способа доступа:

1. `window.frames[0]` – доступ по номеру.
2. `window.frames.iframeName` – доступ по `name` ифрейма.

Обратим внимание: в коллекции хранится именно окно (`contentWindow`), а не DOM-элемент.

Демонстрация всех способов доступа к окну:

```
<iframe src="javascript:''" style="height:80px" name="i"></iframe>

<script>
  var iframeTag = document.body.children[0];

  var iframeWindow = iframeTag.contentWindow; // окно из тега

  alert( frames[0] === iframeWindow ); // true, окно из коллекции frames
  alert( frames.i == iframeWindow ); // true, окно из frames по имени
</script>
```

Внутри ифрейма могут быть свои вложенные ифреймы. Всё это вместе образует иерархию.

Ссылки для навигации по ней:

- `window.frames` – коллекция «детей» (вложенных ифреймов)
- `window.parent` – содержит ссылку на родительское окно, позволяет обратиться к нему из ифрейма.

  Всегда верно:

```
// (из окна со фреймом)
window.frames[0].parent === window; // true
```

- `window.top` – содержит ссылку на самое верхнее окно (вершину иерархии).

  Всегда верно (в предположении, что вложенные фреймы существуют):

```
window.frames[0].frames[0].frames[0].top === window
```

Свойство **`top`** позволяет легко проверить, во фрейме ли находится текущий документ:

```
if (window == top) {
  alert( 'Этот скрипт является окном верхнего уровня в браузере' );
} else {
  alert( 'Этот скрипт исполняется во фрейме!' );
}
```

Песочница sandbox

Атрибут `sandbox` позволяет построить «песочницу» вокруг ифрейма, запретив ему выполнять ряд действий.

Наличие атрибута `sandbox` :

- Заставляет браузер считать ифрейм загруженным с другого источника, так что он и внешнее окно больше не могут обращаться к переменным друг друга.
- Отключает формы и скрипты в ифрейме.
- Запрещает менять `parent.location` из ифрейма.

Пример ниже загружает в `<iframe sandbox>` документ с JavaScript и формой. Ни то ни другое не сработает:

Если у атрибута `sandbox` нет значения, то браузер применяет максимум ограничений.

Атрибут `sandbox` может содержать через пробел список ограничений, которые не нужны:

allow-same-origin

Браузер будет считать документ в ифрейме пришедшим с другого домена и накладывать соответствующие ограничения на работу с ним. Если ифрейм и так с другого домена, то ничего не меняется.

allow-top-navigation

Разрешает ифрейму менять `parent.location` .

allow-forms

Разрешает отправлять формы из `iframe` .

allow-scripts

Разрешает выполнение скриптов из ифрейма. Но скриптам, всё же, будет запрещено открывать попапы.

> **На заметку:**
> Цель атрибута `sandbox` – наложить дополнительные ограничения. Он не может снять уже существующие, в частности, убрать ограничения безопасности, если ифрейм с другого источника.

Кросс-доменные ограничения и их обход

Ограничение «Same Origin» («тот же источник») ограничивает доступ окон и фреймов друг к другу, а также влияет на AJAX-запросы к серверу.

Причина, по которой оно существует – безопасность. Если есть два окна, в одном из которых `vasya-pupkin.com` , а в другом `gmail.com` , то мы бы не хотели, чтобы скрипт из первого мог читать нашу почту.

Сама концепция проста, но есть много важных исключений и особенностей, которые нужно знать для полного понимания этого правила.

Концепция Same Origin

Два URL считаются имеющим один источник («same origin»), если у них одинаковый протокол, домен и порт.

Эти URL имеют один источник:

- `http://site.com`
- `http://site.com /`
- `http://site.com/my/page.html`

А вот эти – все из других источников:

- http://www.site.com (другой домен)
- http://site.org (другой домен)
- https://site.com (другой протокол)
- http://site.com:8080 (другой порт)

Существует ряд исключений, позволяющих-таки окнам с разных доменов обмениваться информацией, но прямой вызов методов друг друга и чтение свойств запрещены.

В действии

Если одно окно попытается обратиться к другому, то браузер проверит, из одного ли они источника. Если нет – доступ будет запрещён.

Например:

```
<iframe src="https://example.com"></iframe>

<script>
  var iframe = document.body.children[0];

  iframe.onload = function() {
    try {
      alert( iframe.contentWindow.document );
    } catch (e) {
      alert( "Ошибка: " + e.message );
    }
  }
</script>
```

Пример выше выведет ошибку.

Исключение: запись в location

Окна могут менять `location` друг друга, даже если они из разных источников.

Причём *читать* свойства `location` нельзя, одно окно не имеет право знать, на каком URL пользователь в другом. А вот *запись* браузеры считают безопасной.

Например, открыв на `javascript.ru` iframe с `http://example.com`, из этого ифрейма нельзя будет прочитать URL, а вот поменять его – запросто:

```
<iframe src="https://example.com"></iframe>

<script>
  var iframe = document.body.children[0];

  iframe.onload = function() {
    try {
      // не сработает (чтение)
      alert( iframe.contentWindow.location.href );
    } catch (e) {
      alert( "Ошибка при чтении: " + e.message );
    }

    // сработает (запись)
    iframe.contentWindow.location.href = 'https://wikipedia.org';

    iframe.onload = null;
  }
</script>
```

Если запустить код выше, то окно сначала загрузит `example.com`, а потом будет перенаправлено на `wikipedia.org`.

Исключение: поддомен 3-го уровня

Ещё одно важное исключение касается доменов третьего уровня.

Если несколько окон имеют общий домен второго уровня, к примеру `john.site.com`, `peter.site.com`, `site.com`, и присваивают в `document.domain` свой общий поддомен 2-го уровня `site.com`, то все ограничения снимаются.

То есть, на всех этих сайтах должен быть код:

```
document.domain = 'site.com';
```

Тогда между ними не будет кросс-доменных ограничений.

Обратим внимание: свойство `document.domain` должно быть присвоено на всех окнах, участвующих в коммуникации. Выглядит абсурдно, но даже на документе с `site.com` нужно вызвать: `document.domain="site.com"`. Иначе не будет работать.

Таким образом разные подсайты в рамках одного общего проекта могут взаимодействовать без ограничений.

Исключения в IE

В браузере Internet Explorer есть два своих, дополнительных исключения из Same Origin Policy.

1. Порт не входит в понятие «источник» (origin).

   Это означает, что окно с `http://site.com` может свободно общаться с `http://site.com:8080`.

   Это иногда используют для общения серверов, использующих один IP-адрес. Но допустимо такое только в IE.

2. Если сайт находится в зоне «Надёжные узлы», то в Internet Explorer ограничения к нему не применяются.

   При этом подразумевается, что для этой зоны в параметрах «Безопасность» включена опция «Доступ к источникам данных за пределами домена».

Итого

Ограничение «одного источника» запрещает окнам и фреймам с разных источников вызывать методы друг друга и читать данные друг из друга.

При этом «из одного источника» означает «совпадают протокол, домен и порт».

У этого подхода ряд существенных исключений:

- Свойства `window.location.*` нельзя читать, но можно менять.
- Домены третьего уровня с общим наддоменом могут поменять `document.domain` на их общий домен второго уровня, и тогда они смогут взаимодействовать без ограничений.
- IE не включает порт в понятие источника. Кроме того, он позволяет снять ограничения для конкретного сайта включением в доверенную зону.

Общение окон с разных доменов: postMessage

Интерфейс `postMessage` позволяет общаться друг с другом окнам и ифреймам с разных доменов.

Он очень удобен, например, для взаимодействия внешних виджетов и сервисов, подключённых через ифрейм с основной страницей.

Отправитель: метод postMessage

Первая часть интерфейса состоит из метода postMessage. Его вызывает окно, которое хочет отправить сообщение, в контексте окна-получателя.

Проще говоря, если мы хотим отправить сообщение в окно `win`, то нужно вызвать `win.postMessage(data, targetOrigin)`.

Аргументы:

data

Данные. По спецификации, это может быть любой объект, который будет *клонирован с сохранением структуры* при передаче.

Но IE поддерживает только строки, поэтому обычно данные JSON-сериализуют.

targetOrigin

Разрешить получение сообщения только окнам с данного источника.

Мы ведь не можем из JavaScript узнать, на каком именно URL находится другое окно. Но иногда хочется быть уверенным, что данные передаются в доверенный документ. Для этого и нужен этот параметр. Проверку осуществляет браузер. При указании `'*'` ограничений нет.

Например:

```
<iframe src="http://target.com" name="target">

<script>
  var win = window.frames.target;
  win.postMessage("сообщение", "http://javascript.ru");
</script>
```

> ⚠ В IE11- можно использовать **`postMessage`** только для ифреймов
>
> В браузере IE, интерфейс `postMessage` работает только с ифреймами. Он не работает между табами и окнами.
>
> Это ошибка в данном конкретном браузере, в других – всё в порядке. Детали по этой и связанным с ней ошибкам: HTML5 Implementation Issues in IE8 and later.

Получатель: событие onmessage

Чтобы получить сообщение, окно должно поставить обработчик на событие `onmessage`.

Свойства объекта события:

`data`

Присланные данные

`origin`

Источник, из которого пришло сообщение, например `http://javascript.ru`.

`source`

Ссылка на окно, с которого пришло сообщение. Можно тут же ответить.

Назначать обработчик нужно обязательно через методы `addEventListener/attachEvent`, например:

```
function listener(event) {
  if (event.origin != 'http://javascript.ru') {
    // что-то прислали с неизвестного домена - проигнорируем..
    return;
  }

  alert( "получено: " + event.data );
}

if (window.addEventListener) {
  window.addEventListener("message", listener);
} else {
  // IE8
  window.attachEvent("onmessage", listener);
}
```

Задержка отсутствуют

Задержки между отправкой и получением нет, совсем.

Если для `setTimeout` стандарт предусматривает минимальную задержку 4 мс, то для `postMessage` она равна 0 мс. Поэтому `postMessage` можно, в том числе, использовать как мгновенную альтернативу `setTimeout`.

Итого

Интерфейс `postMessage` позволяет общаться окнам и ифреймам с разных доменов (в IE8 – только ифреймы), при этом обеспечивая проверки безопасности.

1. Отправитель вызывает `targetWin.postMessage(data, targetOrigin)`.
2. Если `targetOrigin` не `'*'`, то браузер проверяет, совпадает ли источник с `targetWin`.
3. Если совпадает, то на `targetWin` генерируется событие `onmessage`, в котором передаются:

- `origin` – источник, с которого пришло сообщение.
- `source` – ссылка на окно-отправитель.
- `data` – данные. Везде, кроме IE, допустимы объекты, которые клонируются, а в IE – только строка.

4. Обработчик на `onmessage` необходимо вешать при помощи специализированных методов `addEventListener/attachEvent`.

Привлечение внимания к окну

Проверить, находится ли окно в фокусе, а также перевести внимание посетителя на него – сложно.

В первую очередь, это потому, что JavaScript не интегрирован с оконным менеджером ОС. Кроме того, браузер охраняет права посетителя: если он хочет скрыть окно, то JavaScript не может его остановить.

Но кое-что сделать, конечно, можно. Об этом и поговорим.

Метод window.focus

Метод `window.focus` позволяет сфокусироваться на окне. Он работает по-разному в разных ОС и браузерах.

Проверьте, например:

```
setInterval(function() { window.focus() }, 1000);
```

Что будет, если запустить этот код, и затем переключиться в другое окно или вкладку?

Можно подумать, что окно будет оказываться в фокусе раз в секунду. Но это не так.

Произойдет одно из трех:

1. Вообще никакого эффекта. Самый распространённый случай, если в окне много вкладок.
2. Окно развернется (при необходимости) и выйдет на передний план. Обычно это происходит, когда метод `window.focus()` вызывается для попапа, а активно сейчас – главное окно. То есть, в этом случае вызов сработает.
3. Заголовок окна начнет мигать. Чтобы увидеть это в действии – откройте данную страницу в IE, запустите код и переключитесь на другое окно. Браузер попытается привлечь Ваше внимание миганием/мерцанием заголовка окна.

Мерцание заголовка

В дополнение к `window.focus()` используют мерцание заголовка окна, как показано в примере ниже:

```
<script>
  var win = open('/', 'test', 'width=300,height=300')

  function getAttention(win) {
    if (win.closed) {
      alert( "Окно закрыто, привлечь внимание к нему нельзя" );
      return;
    }

    win.focus();
    var i = 0;
    var show = ['************', win.document.title];

    var focusTimer = setInterval(function() {
      if (win.closed) {
        clearInterval(focusTimer);
        return;
      }
      win.document.title = show[i++ % 2];
    }, 1000);

    win.document.onmousemove = function() {
      clearInterval(focusTimer);
      win.document.title = show[1];
      win.document.onmousemove = null;
    }
  }
</script>

<input type="button" onclick="getAttention(win)" value="getAttention(win)">
```

Запустите код и сверните всплывающее окно. А затем – нажмите кнопку с надписью «getAttention(win)». Браузер будет привлекать ваше внимание, как умеет ;)

Обратите внимание: в коде есть проверка на `win.closed`. Попытка манипулирования закрытым окном вызовет исключение.

Как только посетитель сфокусировался на окне индикация прекращается. Для определения момента фокусировки в примере выше используется событие `document.onmousemove`.

Можно было использовать событие `window.onfocus`, но, оказывается, оно ненадежно.

Событие window.onfocus

Вот переписанный вариант функции `getAttention(win)`, с использованием события `onfocus`:

```
<script>
  var win = open('/', 'test', 'width=300,height=300')

  function getAttention(win) {
    if (win.closed) {
      alert( "Окно закрыто, привлечь внимание к нему нельзя" );
      return;
    }

    var i = 0;
    var show = ['************', win.document.title];

    function stop() {
      clearInterval(focusTimer);
      win.document.title = show[1];
    }

    win.onfocus = function() {
      stop();
      win.onfocus = null;
    }

    var focusTimer = setInterval(function() {
      if (win.closed) {
        clearInterval(focusTimer);
        return;
      }

      win.document.title = show[i++ % 2];
    }, 1000);

    win.focus();
  }
</script>

<input type="button" onclick="getAttention(win)" value="getAttention(win)">
```

Далее мы посмотрим случаи, когда он не срабатывает, и почему нам всё же нужно `document.onmousemove`.

Когда событие onfocus не работает?

Возможно такое, что посетитель переключается на окно, а `window.onfocus` не происходит.

Это потому, что переключение между окнами и фокусировка – это разные вещи. Например, если курсор находится в поле для ввода URL браузера, то считается, что окно не в фокусе, хотя посетитель переключился на это окно.

Попробуйте проделать следующее:

1. Запустите пример с `getAttention` в Chrome или IE (кстати, в них нельзя отключить адресную панель).
2. Поместите курсор в панель адреса всплывающего окна.
3. Перейдите обратно к главному окну и нажмите кнопку `getAttention(win)`

Вы увидите, что несмотря на то, что вы переключились на окно, и оно сейчас на переднем плане, событие `onfocus` не срабатывает.

Есть и другие случаи, когда переключение между окнами не вызывает `window.onfocus`. Скажем, если окно сфокусировать щелчком в поле ввода формы, то в IE события `window.onfocus` (а также `window.onfocusin`) – не сработают!

Можно попробовать поймать момент фокусировки и по-другому, повесив дополнительные обработчики событий на `document`. В главе Фокусировка: focus/blur описана техника делегирования для `focus/blur`.

Но этот способ получает фокус только если посетитель сфокусируется где-то в документе: щелкнет или сделает еще какое-то действие в документе, а не просто посмотрит на него и проведет над ним мышкой.

Впрочем, никто не мешает использовать сочетание всех описанных методов.

Итого

Фокусировка и привлечение внимания к окну:

- Метод `focus` для `window` не надёжен. Окнами и вкладками браузера можно уверенно управлять только на уровне ОС.

  Поэтому для привлечения внимания посетителя к окну стоит также использовать мерцающий заголовок окна.

Обнаружение переключения на окно:

- У `window` есть событие `onfocus`, но оно также ненадёжно.

  Поэтому для определения переключения на окно – используйте его вместе с делегируемым `focus` на документе, а также `document.onmousemove`.

Атака Clickjacking и защита от неё

Атака «кликджекинг» (англ. Clickjacking) позволяет хакеру выполнить клик на сайте-жертве *от имени посетителя*.

В русском языке встречается дословный перевод термина clickjacking: «угон клика». Так же применительно к clickjacking-атаке можно встретить термины «перекрытие iframe» и «подмена пользовательского интерфейса».

Кликджекингу подверглись в своё время Twitter, Facebook , PayPal, YouTube и многие другие сайты. Сейчас, конечно, они уже защищены.

Идея атаки

В целом идея очень проста.

Вот как выглядел «угон клика» пользователя, который зарегистрирован на Facebook:

1. На вредоносной странице пользователю подсовывается безобидная ссылка (скажем, что-то скачать, «разбогатеть сейчас», посмотреть ролик или просто перейти по ссылке на интересный ресурс).
2. Поверх этой заманчивой ссылки помещен прозрачный iframe со страницей facebook.com, так что кнопка «Like» находится чётко над ней.
3. Кликая на ссылку, посетитель на самом деле нажимает на эту кнопку.

Демо

Вот пример вредоносной страницы (для наглядности `iframe` – полупрозрачный):

```
<style>
iframe { /* iframe с сайта-жертвы */
  width: 400px;
  height: 100px;
  position: absolute;
  top:0; left:-20px;
  opacity: 0.5; /* в реальности opacity:0 */
  z-index: 1;
}
</style>

<div>Нажмите, чтобы разбогатеть сейчас:</div>

<!-- URL в реальности - с другого домена (атакуемого сайта) -->
<iframe src="facebook.html"></iframe>

<button>Жми тут!</button>

<div>..И всё получится (хе-хе, у меня, злого хакера, получится)!</div>
```

В действии:

Так как `<iframe src="facebook.html">` полупрозрачный, то в примере выше легко видеть, как он перекрывает кнопку. При клике на «Жми тут» на самом деле происходит клик на `<iframe>` (на «Like»).

В итоге, если посетитель авторизован на facebook (а в большинстве случаев так и есть), то facebook.com получает щелчок от имени посетителя.

На Twitter это была бы кнопка «Follow».

Тот же самый пример, но ближе к реальности, с `opacity:0` для `<iframe>`. Вообще незаметно, что на самом деле посетитель кликает на `<iframe>`:

Итак, все, что нужно для проведения атаки – это правильно расположить iframe на вредоносной странице, так чтобы кнопка с Facebook оказалась над «Жми тут!». В большинстве случаев это возможно и делается обычным CSS-позиционированием.

> **С клавиатурой так не сделаешь**
>
> Атака называется «Clickjacking», то есть «угон клика», так как события клавиатуры «угнать» гораздо труднее. Посетителя можно заставить сфокусироваться на `<input>` прозрачного `<iframe>` с сайтом-жертвой, но этот `<input>` невидим, а значит текст в нём также будет невидимым. Посетитель начнёт печатать, но, не увидев текст, прекратит свои действия.

Плохая защита

Самый старый метод защиты – это код JavaScript, не позволяющий отобразить веб-страницу внутри фрейма («framebusting», также его называют «framekilling» и «framebreaking»).

Примерно такой:

```
if (top != window) {
  top.location = window.location;
}
```

То есть, если окно обнаруживает, что оно загружено во фрейме, то оно автоматически делает себя верхним.

Увы, в настоящий момент это уже не является сколько-нибудь надежной защитой. Есть несколько способов обхода framebusting. Давайте рассмотрим некоторые из них.

Блокировка top-навигации.

Можно заблокировать переход, инициированный сменой `top.location`, в событии onbeforeunload.

Обработчик этого события ставится на внешней (хакерской) странице и, при попытке `iframe` поменять `top.location`, спросит посетителя, хочет он покинуть данную страницу. В большинстве браузеров хакер может спросить посетителя, используя своё сообщение.

```
window.onbeforeunload = function() {
  window.onbeforeunload = null;
  return "Хотите уйти с этой страницы, не узнав все её тайны (хе-хе)?";
}
```

Так что, скорее всего, посетитель ответит на такой странный вопрос отрицательно (он же не знает про ифрейм, видит только страницу, причины для ухода нет). А значит, ожидаемая смена `top.location` не произойдёт!

Пример в действии:

Атрибут sandbox

Современные браузеры поддерживают атрибут sandbox

Он позволяет разрешить во фрейме скрипты `allow-scripts` и формы `allow-forms`, но запретить top-навигацию (не указать `allow-top-navigation`).

«Защищённый» `<iframe>` хакер может подключить, к примеру, так:

```
<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>
```

Есть и другие приёмы для обхода этой простейшей защиты.

Firefox и старый IE могут активировать designMode на исходной странице, это также предотвращает framebusting, у IE есть нестандартный атрибут security для ифреймов, который можно использовать с той же целью.

Как мы видим, эта защита не только не выдерживает реальной атаки, но и может скомпрометировать сайт (программист-то думает, что защитил его).

Заголовок X-Frame-Options

Все современные браузеры поддерживают заголовок `X-Frame-Options`.

Он разрешает или запрещает отображение страницы, если она открыта во фрейме.

Браузеры игнорируют заголовок, если он определен в META теге. Таким образом, `<meta http-equiv="X-Frame-Options"...>` будет проигнорирован.

У заголовка может быть три значения:

SAMEORIGIN

Рендеринг документа, при открытии во фрейме, производится только в том случае, когда верхний (top) документ – с того же домена.

DENY

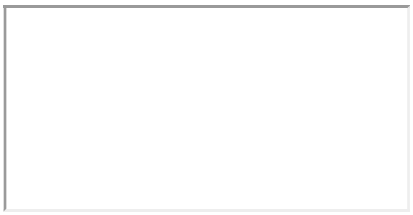
Рендеринг документа внутри фрейма запрещён.

ALLOW-FROM domain

Разрешает рендеринг, если внешний документ с данного домена (не поддерживается в Safari, Firefox).

К примеру, Twitter использует `X-Frame-Options: SAMEORIGIN`. Результат:

```
<iframe src="https://twitter.com"></iframe>
```



В зависимости от браузера, `iframe` выше либо пустой, либо в нём находится сообщение о невозможности отобразить его (IE).

Показ с отключённым функционалом

Заголовок `X-Frame-Options` имеет неприятный побочный эффект. Иногда поисковики, анонимайзеры или другие сайты хотели бы отобразить страницу в `iframe`, по вполне «легальным» причинам, но не могут.

Хорошо бы показывать их посетителям не пустой `iframe`, а нечто, что может быть более интересно.

Например, можно изначально «накрывать» документ `div` с `height:100%;width:100%`, который будет перехватывать все клики. И поставить на нём ссылку, ведующую на страницу в новом окне.

```
<style>
  #iframe-protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
    z-index: 99999999;
  }
</style>

<div id="iframe-protector">
  <a href="/" target="_blank">Перейти на сайт</a>
</div>

<script>
  if (top.document.domain == document.domain) {
    убрать iframe - protector
  }
</script>
```

Если страница – не во фрейме или домен совпадает, то посетитель не увидит его.

Заключение

Атаку «Clickjacking» легко осуществить, если на сайте есть действие, активируемое с помощью одного клика.

Злоумышленник может осуществить атаку целенаправленно на посетителей ресурса – опубликовав ссылку на форуме, или «счастливой рассылкой». Существует масса вариантов.

С первого взгляда, она «неглубокая»: всё, что можно сделать – это один клик. С другой стороны, если хакер знает, что после клика появляется какой-то другой управляющий элемент, то он, хитрыми сообщениями, может заставить посетителя кликнуть и по нему. А это уже не один, а два клика.

Атака особенно опасна, поскольку, проектируя интерфейс сайта, обычно никто и не задумывается о том, что клик от имени юзера может сделать хакер. Точки уязвимости могут быть в совершенно непредсказуемых местах.

- Рекомендуется использовать `X-Frame-Options` на страницах, заведомо не предназначеных для запуска во фрейме и на важнейших страницах (финансовые транзакции).
- Используйте перекрывающий `<div>`, если это допустимо вашим проектом и вы хотите разрешить безопасный показ документа во фреймах с любых доменов.

CSS для JavaScript-разработчика

О чём пойдёт речь

Неужели мы сейчас будем учить CSS? Ничего подобного. Предполагается, что вы *уже* знаете CSS, во всяком случае понимаете его на таком уровне, который позволяет делать Web-страницы.

Особенность квалификации JavaScript-разработчика заключается в том, что он не обязан выбирать цвета, рисовать иконки, «делать красиво». Он также не обязан верстать макет в HTML, разве что если является по совместительству специалистом-верстальщиком.

Вот что он должен уметь абсолютно точно – так это и разработать такую структуру HTML/CSS для элементов управления, которая не сломается, и с которой ему же потом удобно будет взаимодействовать.

Это требует отличного знания CSS в области позиционирования элементов, включая тонкости работы `display`, `margin`, `border`, `outline`, `position`, `float`, `border-box` и остальных свойств, а также подходы к построению структуры компонент (CSS layouts).

Многое можно сделать при помощи JavaScript. И зачастую, не зная CSS, так и делают. Но мы на это ловиться не будем.

> Если что-то можно сделать через CSS – лучше делать это через CSS.

Причина проста – обычно, даже если CSS на вид сложнее – поддерживать и развивать его проще, чем JS. Поэтому овчинка стоит выделки.

Кроме того, есть ещё одно наблюдение.

> Знание JavaScript не может заменить знание CSS.

Жить становится приятнее и проще, если есть хорошее знание и CSS и JavaScript.

Чек-лист

Ниже находится «чек-лист». Если хоть одно свойство незнакомо – это стоп-сигнал для дальнейшего чтения этого раздела.

- Блочная модель, включая:
  - `margin`, `padding`, `border`, `overflow`
  - а также `height/width` и `min-height/min-width`.
- Текст:
  - `font`
  - `line-height`.
- Различные курсоры `cursor`.
- Позиционирование:
  - `position`, `float`, `clear`, `display`, `visibility`
  - Центрирование при помощи CSS
  - Перекрытие `z-index` и прозрачность `opacity`
- Селекторы:
  - Приоритет селекторов
  - Селекторы `#id`, `.class`, `a > b`
- Сброс браузерных стилей, reset.css

Почитать

Книжек много, но хороших – как всегда, мало.

С уверенностью могу рекомендовать следующие:

- Большая книга CSS3. *Дэвид Макфарланд.*
- CSS. Каскадные таблицы стилей. Подробное руководство. *Эрик Мейер*

Дальнейшие статьи раздела не являются *учебником* CSS, поэтому пожалуйста, изучите эту технологию до них.

Это очерки для лучшей систематизации и дополнения уже существующих знаний.

Единицы измерения: "px", "em", "rem" и другие

В этом очерке я постараюсь не только рассказать о различных единицах измерения, но и построить общую картину – что и когда выбирать.

Пиксели: px

Пиксель `px` – это самая базовая, абсолютная и окончательная единица измерения.

Количество пикселей задаётся в настройках разрешения экрана, один `px` – это как раз один такой пиксель на экране. Все значения браузер в итоге пересчитает в пиксели.

Пиксели могут быть дробными, например размер можно задать в `16.5px`. Это совершенно нормально, браузер сам использует дробные пиксели для внутренних вычислений. К примеру, есть элемент шириной в `100px`, его нужно разделить на три части – волей-неволей появляются `33.333...px`. При окончательном отображении дробные пиксели, конечно же, округляются и становятся целыми.

Для мобильных устройств, у которых много пикселей на экране, но сам экран маленький, чтобы обеспечить читаемость, браузер автоматически применяет масштабирование.

Достоинства

- Главное достоинство пикселя – чёткость и понятность

Недостатки

- Другие единицы измерения – в некотором смысле «мощнее», они являются относительными и позволяют устанавливать соотношения между различными размерами

Давно на свалке: `mm`, `cm`, `pt`, `pc`

Существуют также «производные» от пикселя единицы измерения: `mm`, `cm`, `pt` и `pc`, но они давно отправились на свалку истории.

Вот, если интересно, их значения:

- `1mm` (мм) = `3.8px`
- `1cm` (см) = `38px`
- `1pt` (типографский пункт) = `4/3 px`
- `1pc` (типографская пика) = `16px`

Так как браузер пересчитывает эти значения в пиксели, то смысла в их употреблении нет.

Почему в сантиметре `cm` содержится ровно 38 пикселей?

В реальной жизни сантиметр – это эталон длины, одна сотая метра. А пиксель может быть разным, в зависимости от экрана.

Но в формулах выше под пикселем понимается «сферический пиксель в вакууме», точка на «стандартизованном экране», характеристики которого описаны в спецификации.

Поэтому ни о каком соответствии `cm` реальному сантиметру здесь нет и речи. Это полностью синтетическая и производная единица измерения, которая не нужна.

Относительно шрифта: em

`1em` – текущий размер шрифта.

Можно брать любые пропорции от текущего шрифта: `2em`, `0.5em` и т.п.

Размеры в `em` – *относительные*, они определяются по текущему контексту.

Например, давайте сравним `px` с `em` на таком примере:

```
<div style="font-size:24px">
  Страусы
  <div style="font-size:24px">Живут также в Африке</div>
</div>
```

  Страусы
  Живут также в Африке

`24` пикселей – и в Африке `24` пикселей, поэтому размер шрифта в `<div>` одинаков.

А вот аналогичный пример с `em` вместо `px`:

```
<div style="font-size:1.5em">
  Страусы
  <div style="font-size:1.5em">Живут также в Африке</div>
</div>
```

  Страусы
  Живут также в Африке

Так как значение в `em` высчитывается относительно *текущего шрифта*, то вложенная строка в `1.5` раза больше, чем первая.

Выходит, размеры, заданные в `em`, будут уменьшаться или увеличиваться вместе со шрифтом. С учётом того, что размер шрифта обычно определяется в родителе, и может быть изменён ровно в одном месте, это бывает очень удобно.

Что такое размер шрифта?

Что такое «размер шрифта»? Это вовсе не «размер самой большой буквы в нём», как можно было бы подумать.

Размер шрифта – это некоторая «условная единица», которая встроена в шрифт.

Она обычно чуть больше, чем расстояние от верха самой большой буквы до низа самой маленькой. То есть, предполагается, что в эту высоту помещается любая буква или их сочетание. Но при этом «хвосты» букв, таких как `p`, `g` могут заходить за это значение, то есть вылезать снизу. Поэтому обычно высоту строки делают чуть больше, чем размер шрифта.

Единицы ex и ch

В спецификации указаны также единицы ex и ch, которые означают размер символа `"x"` и размер символа `"0"`.

Эти размеры присутствуют в шрифте всегда, даже если по коду этих символов в шрифте находятся другие значения, а не именно буква `"x"` и ноль `"0"`. В этом случае они носят более условный характер.

Эти единицы используются чрезвычайно редко, так как «размер шрифта» `em` обычно вполне подходит.

Проценты %

Проценты `%`, как и `em` – относительные единицы.

Когда мы говорим «процент», то возникает вопрос – «Процент от чего?»

Как правило, процент будет от значения свойства родителя с тем же названием, но не всегда.

Это очень важная особенность процентов, про которую, увы, часто забывают.

Отличный источник информации по этой теме – стандарт, Visual formatting model details.

Вот пример с `%`, он выглядит в точности так же, как с `em`:

```
<div style="font-size:150%">
  Страусы
  <div style="font-size:150%">Живут также в Африке</div>
</div>
```

Страусы
Живут также в Африке

В примере выше процент берётся от размера шрифта родителя.

А вот примеры-исключения, в которых `%` берётся не так:

`margin-left`

При установке свойства `margin-left` в `%`, процент берётся от *ширины* родительского блока, а вовсе не от его `margin-left`.

`line-height`

При установке свойства `line-height` в `%`, процент берётся от текущего *размера шрифта*, а вовсе не от `line-height` родителя. Детали по `line-height` и размеру шрифта вы также можете найти в статье Свойства "font-size" и "line-height".

`width/height`

Для `width/height` обычно процент от ширины/высоты родителя, но при `position:fixed`, процент берётся от ширины/высоты *окна* (а не родителя и не документа). Кроме того, иногда `%` требует соблюдения дополнительных условий, за примером – обратитесь к главе Особенности свойства "height" в %.

Единица rem: смесь px и em

Итак, мы рассмотрели:

- `px` – абсолютные, чёткие, понятные, не зависящие ни от чего.
- `em` – относительно размера шрифта.
- `%` – относительно такого же свойства родителя (а может и не родителя, а может и не такого же – см. примеры выше).

Может быть, пора уже остановиться, может этого достаточно?

Э-э, нет! Не все вещи делаются удобно.

Вернёмся к теме шрифтов. Бывают задачи, когда мы хотим сделать на странице большие кнопки «Шрифт больше» и «Шрифт меньше». При нажатии на них будет срабатывать JavaScript, который будет увеличивать или уменьшать шрифт.

Вообще-то это можно сделать без JavaScript, в браузере обычно есть горячие клавиши для масштабирования вроде `Ctrl++`, но они работают слишком тупо – берут и увеличивают всю страницу, вместе с изображениями и другими элементами, которые масштабировать как раз не надо. А если надо увеличить только шрифт, потому что посетитель хочет комфортнее читать?

Какую единицу использовать для задания шрифтов? Наверно не `px`, ведь значения в `px` абсолютны, если менять, то во всех стилевых правилах. Вполне возможна ситуация, когда мы в одном правиле размер поменяли, а другое забыли.

Следующие кандидаты – `em` и `%`.

Разницы между ними здесь нет, так как при задании `font-size` в процентах, эти проценты берутся от `font-size` родителя, то есть ведут себя так же, как и `em`.

Вроде бы, использовать можно, однако есть проблема.

Попробуем использовать этот подход для `<li>`.

Протестируем на таком списке:

```
<ul>
<li>Собака
  <ul>
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>
```

- Собака
  - бывает
    - кусачей
      - только
        - от жизни
          - собачей

Пока это обычный вложенный список.

Теперь уменьшим размер шрифта до `0.8em`, вот что получится:

```
<style>
  li {
    font-size: 0.8em;
  }
</style>

<ul>
<li>Собака
  <ul>
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>
```

- Собака
  - бывает
    - кусачей
      - только
        - от жизни
          - собачей

Проблема очевидна. Хотели, как лучше, а получилось... Мелковато. Каждый вложенный `<li>` получил размер шрифта `0.8` от родителя, в итоге уменьшившись до нечитаемого состояния. Это не совсем то, чего мы бы здесь хотели.

Можно уменьшить размер шрифта только на одном «корневом элементе»... Или воспользоваться единицей `rem`, которая, можно сказать, специально придумана для таких случаев!

Единица **`rem`** задаёт размер относительно размера шрифта элемента **`<html>`**.

Как правило, браузеры ставят этому элементу некоторый «разумный» (reasonable) размер по-умолчанию, который мы, конечно, можем переопределить и использовать `rem` для задания шрифтов внутри относительно него:

```
<style>
  html {
    font-size: 14px;
  }
  li {
    font-size: 0.8rem;
  }
</style>
```

```
<div><button id="up">Кликните, чтобы увеличить размер шрифта</button></div>

<img src="https://js.cx/clipart/angry_dog.png">

<ul>
<li>Собака
  <ul>
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>

<script>
var html = document.documentElement;
up.onclick = function() {
  // при помощи JS увеличить размер шрифта html на 2px
  html.style.fontSize = parseInt(getComputedStyle(html, '').fontSize) + 2 + 'px';
};
</script>
```

Кликните, чтобы увеличить размер шрифта

- Собака
  - бывает
    - кусачей
      - только
        - от жизни
          - собачей

Получилось удобное масштабирование для шрифтов, не влияющее на другие элементы.

Элементы, размер которых задан в `rem`, не зависят друг от друга и от контекста – и этим похожи на `px`, а с другой стороны они все заданы относительно размера шрифта `<html>`.

Единица `rem` не поддерживается в IE8-.

Относительно экрана: vw, vh, vmin, vmax

Во всех современных браузерах, исключая IE8-, поддерживаются новые единицы из черновика стандарта CSS Values and Units 3:

- `vw` – 1% ширины окна
- `vh` – 1% высоты окна
- `vmin` – наименьшее из (`vw`, `vh`), в IE9 обозначается `vm`
- `vmax` – наибольшее из (`vw`, `vh`)

Эти значения были созданы, в первую очередь, для поддержки мобильных устройств.

Их основное преимущество – в том, что любые размеры, которые в них заданы, автоматически масштабируются при изменении размеров окна.

Итого

Мы рассмотрели единицы измерения:

- `px` – абсолютные пиксели, к которым привязаны и потому не нужны `mm`, `cm`, `pt` и `pc`. Используется для максимально конкретного и точного задания размеров.
- `em` – задаёт размер относительно шрифта родителя, можно относительно конкретных символов: `"x"` (`ex`) и `"0"` (`ch`), используется там, где нужно упростить масштабирование компоненты.

- `rem` – задаёт размер относительно шрифта `<html>`, используется для удобства глобального масштабирования: элементы которые планируется масштабировать, задаются в `rem`, а JS меняет шрифт у `<html>`.
- `%` – относительно такого же свойства родителя (как правило, но не всегда), используется для ширин, высот и так далее, без него никуда, но надо знать, относительно чего он считает проценты.
- `vw`, `vh`, `vmin`, `vmax` – относительно размера экрана.

Все значения свойства "display"

Свойство `display` имеет много разных значений. Обычно, используются только три из них: `none`, `inline` и `block`, потому что когда-то браузеры другие не поддерживали.

Но после ухода IE7-, стало возможным использовать и другие значения тоже. Рассмотрим здесь весь список.

Значение none

Самое простое значение. Элемент не показывается, вообще. Как будто его и нет.

```
<div style="border:1px solid black">
Невидимый div (
  <div style="display: none">Я - невидим!</div>
) Стоит внутри скобок
</div>
```

Невидимый div ( ) Стоит внутри скобок

Значение block

- Блочные элементы располагаются один над другим, вертикально (если нет особых свойств позиционирования, например `float`).
- Блок стремится расшириться на всю доступную ширину. Можно указать ширину и высоту явно.

Это значение `display` многие элементы имеют по умолчанию: `<div>`, заголовок `<h1>`, параграф `<p>`.

```
<div style="border:1px solid black">
  <div style="border:1px solid blue; width: 50%">Первый</div>
  <div style="border:1px solid red">Второй</div>
</div>
```

Первый

Второй

Блоки прилегают друг к другу вплотную, если у них нет `margin`.

Значение inline

- Элементы располагаются на той же строке, последовательно.
- Ширина и высота элемента определяются по содержимому. Поменять их нельзя.

Например, инлайновые элементы по умолчанию: `<span>`, `<a>`.

```
<span style="border:1px solid black">
  <span style="border:1px solid blue; width:50%">Ширина</span>
  <a style="border:1px solid red">Игнорируется</a>
</span>
```

Ширина Игнорируется

Если вы присмотритесь внимательно к примеру выше, то увидите, что между внутренними `<span>` и `<a>` есть пробел. Это потому, что он есть в HTML.

Если расположить элементы вплотную – его не будет:

```
<span style="border:1px solid black">
  <span style="border:1px solid blue; width:50%">Без</span><a style="border:1px solid red">Пробела</a>
</span>
```

БезПробела

Содержимое инлайн-элемента может переноситься на другую строку.

При этом каждая строка в смысле отображения является отдельным прямоугольником («line box»). Так что инлайн-элемент состоит из объединения прямоугольников, но в целом, в отличие от блока, прямоугольником не является.

Это проявляется, например, при назначении фона.

Например, три прямоугольника подряд:

```
<div style="width:400px">
...<span style="background: lightgreen">
  Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
  Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
</span>...
</div>
```

... Ля
Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
Ля Ля Ля Ля Ля ...

Если инлайн-элемент граничит с блоком, то между ними обязательно будет перенос строки:

```
<div style="border:1px solid black">
  <span style="border:1px solid red">Инлайн</span>
  <div style="border:1px solid blue; width:50%">Блок</div>
  <span style="border:1px solid red">Инлайн</span>
</div>
```

Инлайн
Блок
Инлайн

Значение inline-block

Это значение – означает элемент, который продолжает находиться в строке (`inline`), но при этом может иметь важные свойства блока.

Как и инлайн-элемент:

- Располагается в строке.
- Размер устанавливается по содержимому.

Во всём остальном – это блок, то есть:

- Элемент всегда прямоугольный.
- Работают свойства `width/height` .

Это значение `display` используют, чтобы отобразить в одну строку блочные элементы, в том числе разных размеров.

Например:

```
<style>
li {
  display: inline-block;
  list-style: none;
  border: 1px solid red;
}
</style>

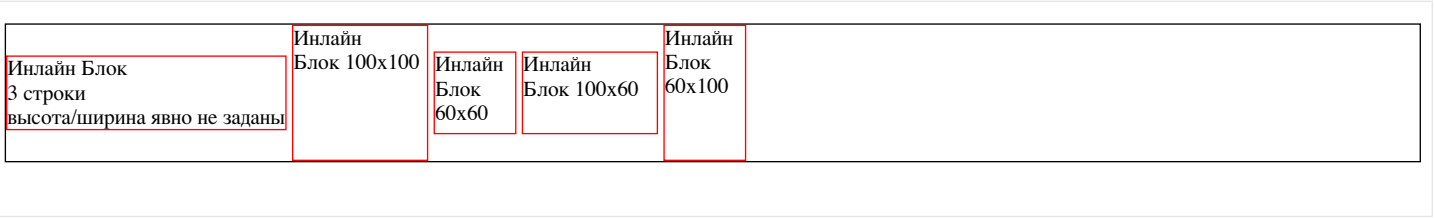
<ul style="border:1px solid black; padding:0">
  <li>Инлайн Блок<br>3 строки<br>высота/ширина явно не заданы</li>
  <li style="width:100px;height:100px">Инлайн<br>Блок 100x100</li>
  <li style="width:60px;height:60px">Инлайн<br>Блок 60x60</li>
  <li style="width:100px;height:60px">Инлайн<br>Блок 100x60</li>
  <li style="width:60px;height:100px">Инлайн<br>Блок 60x100</li>
</ul>
```

Инлайн Блок
3 строки
высота/ширина явно не заданы
Инлайн
Блок 100x100
Инлайн
Блок
60x60
Инлайн
Блок 100x60
Инлайн
Блок
60x100

Свойство `vertical-align` позволяет выровнять такие элементы внутри внешнего блока:

```
<style>
li {
  display: inline-block;
  list-style: none;
  border:1px solid red;
  vertical-align: middle;
}
</style>

<ul style="border:1px solid black; padding:0">
  <li>Инлайн Блок<br>3 строки<br>высота/ширина явно не заданы</li>
  <li style="width:100px;height:100px">Инлайн<br>Блок 100x100</li>
  <li style="width:60px;height:60px">Инлайн<br>Блок 60x60</li>
  <li style="width:100px;height:60px">Инлайн<br>Блок 100x60</li>
  <li style="width:60px;height:100px">Инлайн<br>Блок 60x100</li>
</ul>
```



Как и в случае с инлайн-элементами, пробелы между блоками появляются из-за пробелов в HTML. Если элементы списка идут вплотную, например, генерируются в JavaScript – их не будет.

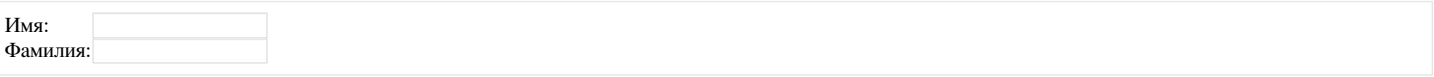
Значения table-*

Современные браузеры (IE8+) позволяют описывать таблицу любыми элементами, если поставить им соответствующие значения `display`.

Для таблицы целиком `table`, для строки – `table-row`, для ячейки – `table-cell` и т.д.

Пример использования:

```
<form style="display: table">
  <div style="display: table-row">
    <label style="display: table-cell">Имя:</label>
    <input style="display: table-cell">
  </div>
  <div style="display: table-row">
    <label style="display: table-cell">Фамилия:</label>
    <input style="display: table-cell">
  </div>
</form>
```



Важно то, что это действительно полноценная таблица. Используются табличные алгоритмы вычисления ширины и высоты элемента, описанные в стандарте.

Это хорошо для семантической вёрстки и позволяет избавиться от лишних тегов.

С точки зрения современного CSS, обычные `<table>`, `<tr>`, `<td>` и т.д. – это просто элементы с предопределёнными значениями `display`:

```
/*+ no-beautify */
table    { display: table }
tr       { display: table-row }
thead    { display: table-header-group }
tbody    { display: table-row-group }
tfoot    { display: table-footer-group }
col      { display: table-column }
colgroup { display: table-column-group }
td, th   { display: table-cell }
caption  { display: table-caption }
```

Очень подробно об алгоритмах вычисления размеров и отображении таблиц рассказывает стандарт CSS 2.1 – Tables.

Вертикальное центрирование с table-cell

Внутри ячеек свойство vertical-align выравнивает содержимое по вертикали.

Это можно использовать для центрирования:

```
<style>
  div { border:1px solid black }
</style>

<div style="height:100px; display: table-cell; vertical-align: middle">
  <div>Элемент<br>С неизвестной<br>Высотой</div>
</div>
```

CSS не требует, чтобы вокруг `table-cell` была структура таблицы: `table-row` и т.п. Может быть просто такой одинокий `DIV`, это допустимо.

При этом он ведёт себя как ячейка `TD`, то есть подстраивается под размер содержимого и умеет вертикально центрировать его при помощи `vertical-align`.

Значения list-item, run-in и flex

У свойства `display` есть и другие значения. Они используются реже, поэтому посмотрим на них кратко:

`list-item`

Этот display по умолчанию используется для элементов списка. Он добавляет к блоку содержимым ещё и блок с номером(значком) списка, который стилизуется стандартными списочными свойствами:

```
<div style="display: list-item; list-style:inside square">Пункт 1</div>
```

- Пункт 1

`run-in`

Если после `run-in` идёт `block`, то `run-in` становится его первым инлайн-элементом, то есть отображается в начале `block`.

Если ваш браузер поддерживает это значение,** то в примере ниже `h3`, благодаря `display:run-in`, окажется визуально внутри `div`:

```
<h3 style="display: run-in; border:2px solid red">Про пчёл.</h3>
<div style="border:2px solid black">Пчёлы - отличные создания, они делают мёд.</div>
```

Про пчёл.

Пчёлы - отличные создания, они делают мёд.

Если же вы видите две строки, то ваш браузер НЕ поддерживает `run-in`.

Вот, для примера, правильный вариант отображения `run-in`, оформленный другим кодом:

```
<div style="border:2px solid black">
  <h3 style="display: inline; border:2px solid red">Про пчёл.</h3>Пчёлы - отличные создания, они делают мёд.
</div>
```

Про пчёл. Пчёлы - отличные создания, они делают мёд.

Если этот вариант отличается от того, что вы видите выше – ваш браузер не поддерживает `run-in`. На момент написания этой статьи только IE поддерживал `display:run-in`.

`flex-box`

Flexbox позволяет удобно управлять дочерними и родительскими элементами на странице, располагая их в необходимом порядке. Официальная спецификация находится здесь: CSS Flexible Box Layout Module

Свойство "float"

Свойство `float` в CSS занимает особенное место. До его появления расположить два блока один слева от другого можно было лишь при помощи таблиц. Но в его работе есть ряд особенностей. Поэтому его иногда не любят, но при их понимании `float` станет вашим верным другом и помощником.

Далее мы рассмотрим, как работает `float`, разберём решения сопутствующих проблем, а также ряд полезных рецептов.

Как работает float

Синтаксис:

```
float: left | right | none | inherit;
```

При применении этого свойства происходит следующее:

1. Элемент позиционируется как обычно, а затем *вынимается из ~~документа~~ потока* и сдвигается влево (для `left`) или вправо (для `right`) до того как коснётся либо границы родителя, либо другого элемента с `float`.
2. Если пространства по горизонтали не хватает для того, чтобы вместить элемент, то он сдвигается вниз до тех пор, пока не начнёт помещаться.
3. Другие непозиционированные блочные элементы без `float` ведут себя так, как будто элемента с `float` нет, так как он убран из потока.
4. Строки (inline-элементы), напротив, «знают» о `float` и обтекают элемент по сторонам.

Ещё детали:

1. Элемент при наличии `float` получает `display:block`.

   То есть, указав элементу, у которого `display:inline` свойство `float: left/right`, мы автоматически сделаем его блочным. В частности, для него будут работать `width/height`.

   Исключением являются некоторые редкие `display` наподобие `inline-table` и `run-in` (см. Relationships between „display", „position", and „float")

2. Ширина `float`-блока определяется по содержимому. («CSS 2.1, 10.3.5»).

3. Вертикальные отступы `margin` элементов с `float` не сливаются с отступами соседей, в отличие от обычных блочных элементов.

Это пока только теория. Далее мы рассмотрим происходящее на примере.

Текст с картинками

Одно из первых применений `float`, для которого это свойство когда-то было придумано – это вёрстка текста с картинками, отжатыми влево или вправо.

Например, вот страница о Винни-Пухе с картинками, которым поставлено свойство `float`:

Её HTML-код выглядит примерно так:

```
<img src="1.jpg" style="float:right">
<p>Текст...</p>
<p>Текст...</p>

<img src="2.jpg" style="float:left">
<p>Текст...</p>

<img src="3.jpg" style="float:right">
<p>Текст...</p>
```

Каждая картинка, у которой есть `float`, обрабатывается в точности по алгоритму, указанному выше.

Посмотрим, например, как выглядело бы начало текста без float:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

1. Элемент `IMG` вынимается из ~~документа~~ потока. Иначе говоря, последующие блоки начинают вести себя так, как будто его нет, и заполняют освободившееся место (изображение для наглядности полупрозрачно):

[iframe src=«winnie-nofloat-1» height=250 border=1 link edit] 2. Элемент `IMG` сдвигается максимально вправо(при `float:right`):

[iframe src=«winnie-nofloat-2» height=250 border=1 link edit] 3. Строки, в отличие от блочных элементов, «чувствуют» `float` и уступают ему место, обтекая картинку слева:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

При `float:left` – всё то же самое, только `IMG` смещается влево (или не смещается, если он и так у левого края), а строки – обтекают справа

Строки и инлайн-элементы смещаются, чтобы уступить место **`float`**. Обычные блоки – ведут себя так, как будто элемента нет.

Чтобы это увидеть, добавим параграфам фон и рамку, а также сделаем изображение немного прозрачным:

Винни-Пух

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

Как видно из рисунка, параграфы проходят «за» `float`. При этом строки в них о `float'ах` знают и обтекают их, поэтому соответствующая часть параграфа пуста.

Блок с float

Свойство `float` можно поставить любому элементу, не обязательно картинке. При этом элемент станет блочным.

Посмотрим, как это работает, на конкретной задаче – сделать рамку с названием вокруг картинки с Винни.

HTML будет такой:

```
<h2>Винни-Пух</h2>

<div class="left-picture">
  <img src="winnie-mult.jpg" width="200" height="150">
  <div>Кадр из советского мультфильма</div>
</div>

<p>Текст...</p>
```

...То есть, `div.left-picture` включает в себя картинку и заголовок к ней. Добавим стиль с `float`:

```
/*+ no-beautify */
.left-picture {
  float: left;

  /* рамочка и отступ для красоты (не обязательно) */
  margin: 0 10px 5px 0;
  text-align: center;
  border: 1px solid black;
}
```

Результат:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Заметим, что блок `div.left-picture` «обернул» картинку и текст под ней, а не растянулся на всю ширину. Это следствие того, что ширина блока с `float` определяется по содержимому.

Очистка под float

Разберём еще одну особенность использования свойства `float`.

Для этого выведем персонажей из мультфильма «Винни-Пух». Цель:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

Реализуем её, шаг за шагом.

Шаг 1. Добавляем информацию

Попробуем просто добавить Сову после Винни-Пуха:

```
<h2>Винни-Пух</h2>
<div class="left">Картинка</div>
<p>..Текст о Винни..</p>

<h2>Сова</h2>
<div class="left">Картинка</div>
<p>..Текст о Сове..</p>
```

Результат такого кода будет странным, но предсказуемым:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

Произошло следующее:

- Заголовок `<h2>Сова</h2>` не заметил `float` (он же блочный элемент) и расположился сразу после предыдущего параграфа `<p>..Текст о Винни..</p>`.
- После него идёт `float`-элемент – картинка «Сова». Он был сдвинут влево. Согласно алгоритму, он двигается до левой границы или до касания с другим `float`-элементом, что и произошло (картинка «Винни-Пух»).
- Так как у совы `float:left`, то последующий текст обтекает её справа.

Шаг 2. Свойство clear

Мы, конечно же, хотели бы расположить заголовок «Сова» и остальную информацию ниже Винни-Пуха.

Для решения возникшей проблемы придумано свойство `clear`.

Синтаксис:

```
clear: left | right | both;
```

Применение этого свойства сдвигает элемент вниз до тех пор, пока не закончатся `float`'ы слева/справа/с обеих сторон.

Применим его к заголовку `H2`:

```
h2 {
  clear: left;
}
```

Результат получившегося кода будет ближе к цели, но всё еще не идеален:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

Элементы теперь в нужном порядке. Но куда пропал отступ `margin-top` у заголовка «Сова»?

Теперь заголовок «Сова» прилегает снизу почти вплотную к картинке, с учётом её `margin-bottom`, но без своего большого отступа `margin-top`.

Таково поведение свойства `clear`. Оно сдвинуло элемент `h2` вниз ровно настолько, чтобы элементов `float` не было *сбоку от его верхней границы*.

Если посмотреть на элемент заголовка внимательно в инструментах разработчика, то можно заметить отступ `margin-top` у заголовка по-прежнему есть, но он располагается «за» элементом `float` и не учитывается при работе в `clear`.

Чтобы исправить ситуацию, можно добавить перед заголовком пустой промежуточный элемент без отступов, с единственным свойством `clear:both`. Тогда уже под ним отступ заголовка будет работать нормально:

```
<h2>Винни-Пух</h2>
<div class="left">Картинка</div>
<p>Текст</p>

<div style="clear:both"></div>

<h2>Сова</h2>
<div class="left">Картинка</div>
<p>Текст</p>
```

Результат получившегося кода:

Винни-Пух


Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова


Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

- Свойство `clear` гарантировало, что `<div style="clear:both">` будет под картинкой с `float`.
- Заголовок `<h2>Сова</h2>` идёт после этого `<div>`. Так что его отступ учитывается.

Заполнение блока-родителя

Итак, мы научились располагать другие элементы *под* `float`. Теперь рассмотрим следующую особенность.

Из-за того, что блок с **`float`** удалён из потока, родитель не выделяет под него места.

Например, выделим для информации о Винни-Пухе красивый элемент-контейнер `<div class="hero">`:

```
<div class="hero">

  <h2>Винни-Пух</h2>

  <div class="left">Картинка</div>

  <p>Текст.</p>
</div>
```

Стиль контейнера:

```
.hero {
  background: #D2B48C;
  border: 1px solid red;
}
```

Результат получившегося кода:

Винни-Пух


Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Элемент с `float` оказался выпавшим за границу родителя `.hero`.

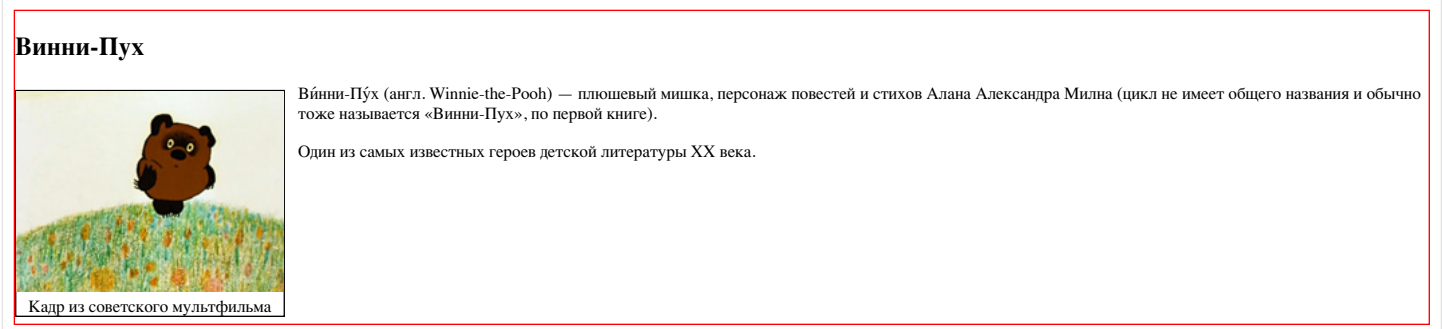
Чтобы этого не происходило, используют одну из следующих техник.

Поставить родителю float

Элемент с `float` обязан расшириться, чтобы вместить вложенные `float`.

Поэтому, если это допустимо, то установка `float` контейнеру всё исправит:

```
/*+ no-beautify */
.hero {
  background: #D2B48C;
  border: 1px solid red;
  float: left;
}
```



Разумеется, не всегда можно поставить родителю `float`, так что смотрим дальше.

Добавить в родителя элемент с clear

Добавим элемент `div style="clear:both"` в самый конец контейнера `.hero`.

Он с одной стороны будет «нормальным» элементом, в потоке, и контейнер будет обязан выделить под него пространство, с другой – он знает о `float` и сместится вниз.

Соответственно, и контейнер вырастет в размере:

```
<div class="hero">

  <h2>Винни-Пух</h2>

  <div class="left">Картинка</div>

  <p>Текст.</p>

  <div style="clear:both"></div>
</div>
```

Результат – правильное отображение, как и в примере выше. Открыть код.

Единственный недостаток этого метода – лишний HTML-элемент в разметке.

Универсальный класс clearfix

Чтобы не добавлять в HTML-код лишний элемент, можно задать его через `:after`.

```
/*+ no-beautify */
.clearfix:after {
  content: "."; /* добавить содержимое: "." */
  display: block;  /* сделать блоком, т.к. inline не может иметь clear */
  clear: both;  /* с обеих сторон clear */
  visibility: hidden; /* сделать невидимым, зачем нам точка внизу? */
  height: 0;  /* сделать высоту 0, чтобы не занимал место */
}
```

Добавив этот класс к родителю, получим тот же результат, что и выше. Открыть код.

overflow:auto/hidden

Если добавить родителю `overflow: hidden` или `overflow: auto`, то всё станет хорошо.

```
/*+ no-beautify */
.hero {
  overflow: auto;
}
```

Этот метод работает во всех браузерах, полный код в песочнице.

Несмотря на внешнюю странность, этот способ не является «хаком». Такое поведение прописано в спецификации CSS.

Однако, установка `overflow` может привести к появлению полосы прокрутки, способ с псевдоэлементом `:after` более безопасен.

float вместо display:inline-block

При помощи `float` можно размещать блочные элементы в строке, похоже на `display: inline-block`:

Стиль здесь:

```
.gallery li {
  float: left;
  width: 130px;
  list-style: none;
}
```

Элементы `float:left` двигаются влево, а если это невозможно, то вниз, автоматически адаптируясь под ширину контейнера, получается эффект, аналогичный `display: inline-block`, но с особенностями `float`.

Вёрстка в несколько колонок

Свойство `float` позволяет делать несколько вертикальных колонок.

float:left + float:right

Например, для вёрстки в две колонки можно сделать два `<div>`. Первому указать `float:left` (левая колонка), а второму – `float:right` (правая колонка).

Чтобы они не ссорились, каждой колонке нужно дополнительно указать ширину:

```
<div>Шапка</div>
<div class="column-left">Левая колонка</div>
<div class="column-right">Правая колонка</div>
<div class="footer">Низ</div>
```

Стили:

```
.column-left {
  float: left;
  width: 30%;
}

.column-right {
  float: left;
  width: 70%;
}

.footer {
  clear: both;
}
```

Результат (добавлены краски):

В эту структуру легко добавить больше колонок с разной шириной. Правой колонке можно было бы указать и `float:right`.

float + margin

Ещё вариант – сделать `float` для левой колонки, а правую оставить в потоке, но с отбивкой через `margin`:

```
.column-left {
  float: left;
  width: 30%;
}

.column-right {
  margin-left: 30%;
}

.footer {
  clear: both;
}
```

Результат (добавлены краски):

В примере выше – показана небольшая проблема. Колонки не растягиваются до одинаковой высоты. Конечно, это не имеет значения, если фон одинаковый, но что, если он разный?

В современных браузерах (кроме IE10-) эту же задачу лучше решает flexbox.

Для старых есть различные обходы и трюки, которые позволяют обойти проблему в ряде ситуаций, но они выходят за рамки нашего обсуждения. Если интересно – посмотрите, например, Faux Columns.

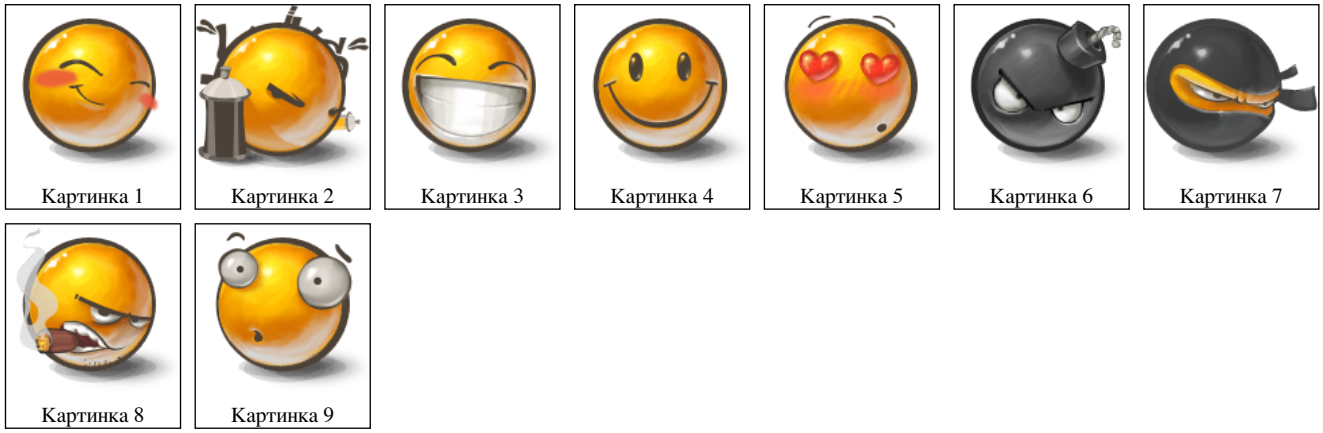
Задачи

Разница inline-block и float

важность: 5

Галерея изображений состоит из картинок в рамках с подписями (возможно, с другой дополнительной информацией).

Пример галереи:



Технически вывод такой галереи можно реализовать при помощи списка UL/LI, где:

1. каждый LI имеет `display:inline-block`
2. каждый LI имеет `float:left`

Какие различия между этими подходами? Какой вариант выбрали бы вы?

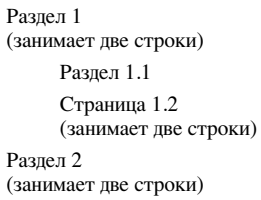
К решению

Дерево с многострочными узлами

важность: 3

Сделайте дерево при помощи семантической вёрстки и CSS-спрайта с иконками (есть готовый).

Выглядеть должно так (не кликабельно):



- Поддержка многострочных названий узлов
- Над иконкой курсор становится указателем.

Исходный документ содержит список UL/LI и ссылку на картинку.

P.S. Достаточно сделать HTML/CSS-структуру, действия добавим позже.

Открыть песочницу для задачи.

К решению

Постраничная навигация (CSS)

важность: 5

Оформите навигацию, центрированную внутри `DIV'a` :

Требования:

- Левая стрелка – слева, правая – справа, список страниц – по центру.
- Список страниц центрирован вертикально.
- Текст сверху и снизу ни на что не наползает.
- Курсор при наведении на стрелку или элемент списка становится стрелкой `pointer`.

P.S. Без использования таблиц.

Открыть песочницу для задачи.

К решению

Добавить рамку, сохранив ширину

важность: 4

Есть две колонки `30%/70%`:

```
<style>
  .left {
    float:left;
    width:30%;
    background: #aef;
  }

  .right {
    float:right;
    width:70%;
    background: tan;
  }
</style>

<div class="left">
 Левая<br>Колонка
</div>
<div class="right">
 Правая<br>Колонка<br>...
</div>
```

Левая
Колонка

Правая
Колонка
...

Добавьте к правой колонке рамку `border-left` и отступ `padding-left`.

Двухколоночная вёрстка при этом не должна сломаться!

Желательно не трогать свойство `width` ни слева ни справа и не создавать дополнительных элементов.

К решению

Свойство "position"

Свойство `position` позволяет сдвигать элемент со своего обычного места. Цель этой главы – не только напомнить, как оно работает, но и разобрать ряд частых заблуждений и граблей.

position: static

Статическое позиционирование производится по умолчанию, в том случае, если свойство `position` не указано.

Его можно также явно указать через CSS-свойство:

```
position: static;
```

Такая запись встречается редко и используется для переопределения других значений `position`.

Здесь и далее, для примеров мы будем использовать следующий документ:

```
<div style="background: #fee; width: 500px">
    Без позиционирования ("position: static").

    <h2 style="background: #aef; margin: 0">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Без позиционирования ("position: static").

Заголовок

А тут - всякий разный текст...
... В две строки!

В этом документе сейчас все элементы отпозиционированы статически, то есть никак.

> Элемент с `position: static` еще называют *не позиционированым*.

position: relative

Относительное позиционирование сдвигает элемент относительно его обычного положения.

Для того, чтобы применить относительное позиционирование, необходимо указать элементу CSS-свойство `position: relative` и координаты `left/right/top/bottom` .

Этот стиль сдвинет элемент на 10 пикселей относительно обычной позиции по вертикали:

```
position: relative;
top: 10px;
```

```
<style>
  h2 {
    position: relative;
    top: 10px;
  }
</style>
```

```
<div style="background: #fee; width: 500px">
    Заголовок сдвинут на 10px вниз.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Заголовок сдвинут на 10px вниз.

Заголовок

А тут - всякий разный текст...
... В две строки!

Координаты

Для сдвига можно использовать координаты:

- `top` – сдвиг от «обычной» верхней границы
- `bottom` – сдвиг от нижней границы
- `left` – сдвиг слева
- `right` – сдвиг справа

Не будут работать одновременно указанные `top` и `bottom` , `left` и `right` . Нужно использовать только одну границу из каждой пары.

Возможны отрицательные координаты и координаты, использующие другие единицы измерения. Например, `left: 10%` сдвинет элемент на 10% его ширины вправо, а `left: -10%` – влево. При этом часть элемента может оказаться за границей окна:

```
<style>
  h2 {
    position: relative;
    left: -10%;
  }
</style>
```

```
<div style="background: #fee; width: 500px">
    Заголовок сдвинут на 10% влево.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Заголовок сдвинут на 10% влево.

ОЛОВОК

А тут - всякий разный текст...
... В две строки!

Свойства **left/top** не будут работать для **position:static** . Если их все же поставить, браузер их проигнорирует. Эти свойства предназначены для работы только с позиционированными элементами.

position: absolute

Синтаксис:

```
position: absolute;
```

Абсолютное позиционирование делает две вещи:

1. Элемент исчезает с того места, где он должен быть и позиционируется заново. Остальные элементы, располагаются так, как будто этого элемента никогда не было.
2. Координаты **top/bottom/left/right** для нового местоположения отсчитываются от ближайшего позиционированного родителя, т.е. родителя с позиционированием, отличным от `static` . Если такого родителя нет – то относительно документа.

Кроме того:

- Ширина элемента с **position: absolute** устанавливается по содержимому. Детали алгоритма вычисления ширины описаны в стандарте .
- Элемент получает **display:block** , который перекрывает почти все возможные `display` (см. Relationships between „display“, „position“, and „float“).

Например, отпозиционируем заголовок в правом-верхнем углу документа:

```
<style>
  h2 {
    position: absolute;
    right: 0;
    top: 0;
  }
</style>
```

```
<div style="background: #fee; width: 500px">
    Заголовок в правом-верхнем углу документа.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Заголовок в правом-верхнем углу документа.

Заголовок

А тут - всякий разный текст...
... В две строки!

Важное отличие от `relative` : так как элемент удаляется со своего обычного места, то элементы под ним сдвигаются, занимая освободившееся пространство. Это видно в примере выше: строки идут одна за другой.

Так как при `position:absolute` размер блока устанавливается по содержимому, то широкий `Заголовок` «съёжился» до прямоугольника в углу.

Иногда бывает нужно поменять элементу `position` на `absolute` , но так, чтобы элементы вокруг не сдвигались. Как правило это делают, меняя соседей – добавляют `margin/padding` или вставляют в документ пустой элемент с такими же размерами.

Одновременное указание `left/right` , `top/bottom`

В абсолютно позиционированном элементе можно одновременно задавать противоположные границы.

Браузер растянет такой элемент до границ.

```
<style>
div {
  position: absolute;
  left: 10px; right: 10px; top: 10px; bottom: 10px;
}
</style>
<div style="background:#aef;text-align:center">10px от границ</div>
```

10px от границ

Внешним блоком является окно

Как растянуть абсолютно позиционированный блок на всю ширину документа?

Первое, что может прийти в голову:

```
/*+ no-beautify */
div {
  position: absolute;
  left: 0; top: 0; /* в левый-верхний угол */
  width: 100%; height: 100%; /* .. и растянуть */
}
```

Но это будет работать лишь до тех пор, пока у страницы не появится скроллинг!

Прокрутите вниз ифрейм:

Прокрутите меня...

Вы увидите, что голубой фон оканчивается задолго до конца документа.

Дело в том, что в CSS `100%` относится к ширине внешнего блока («containing block»). А какой внешний блок имеется в виду здесь, ведь элемент изъят со своего обычного места?

В данном случае им является так называемый ("«initial containing block»"), которым является окно, *а не документ*.

То есть, координаты и ширины вычисляются относительно окна, а не документа.

Может быть, получится так?

```
/*+ no-beautify */
div {
  position: absolute;
  left: 0; top: 0; /* в левый-верхний угол, и растянуть..  */
  right: 0; bottom: 0; /* ..указанием противоположных границ */
}
```

С виду логично, но нет, не получится!

Координаты `top/right/left/bottom` вычисляются относительно *окна*. Значение `bottom: 0` – нижняя граница окна, а не документа, блок растянется до неё. То есть, будет то же самое, что и в предыдущем примере.

position: absolute в позиционированном родителе

Если у элемента есть позиционированный предок, то `position: absolute` работает относительно него, а не относительно документа.

То есть, достаточно поставить родительскому `div` позицию `relative` , даже без координат – и заголовок будет в его правом-верхнем углу, вот так:

```
<style>
  h2 {
    position: absolute;
    right: 0;
    top: 0;
  }
</style>
```

```
<div style="background: #fee; width: 500px; position: relative">
    Заголовок в правом-верхнем углу DIV'а.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
         ... В две строки!</div>
</div>
```

Заголовок в правом-верхнем углу DIV'а.
А тут - всякий разный текст...
... В две строки!

Заголовок

Нужно пользоваться таким позиционированием с осторожностью, т.к оно может перекрыть текст. Этим оно отличается от `float` .

Сравните:

- Используем `position` для размещения элемента управления:

  ```
  <button style="position: absolute; right: 10px; opacity: 0.8">
    Кнопка
  </button>
  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
  ```

  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 Кнопка

  Часть текста перекрывается. Кнопка более не участвует в потоке.

- Используем `float` для размещения элемента управления:

  ```
  <button style="float: right; margin-right: 10px; opacity: 0.8;">
    Кнопка
  </button>
  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
  ```

  1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 Кнопка

  Браузер освобождает место справа, текст перенесён. Кнопка продолжает находиться в потоке, просто сдвинута.

position: fixed

Это подвид абсолютного позиционирования.

Синтаксис:

```
position: fixed;
```

Позиционирует объект точно так же, как `absolute` , но относительно `window` .

Разница в нескольких словах:

Когда страницу прокручивают, фиксированный элемент остается на своем месте и не прокручивается вместе со страницей.

В следующем примере, при прокрутке документа, ссылка `#top` всегда остается на своем месте.

```
<style>
  #top {
    position: fixed;
    right: 10px;
    top: 10px;
    background: #fee;
  }
</style>

<a href="#" id="top">Наверх (остается при прокрутке)</a>

Фиксированное позиционирование.

<p>Текст страницы.. Прокрути меня...</p>
<p>Много строк..</p><p>Много строк..</p>
<p>Много строк..</p><p>Много строк..</p>
```

```
<p>Много строк..</p><p>Много строк..</p>
<p>Много строк..</p><p>Много строк..</p>
```

Фиксированное позиционирование.

Наверх (остается при прокрутке)

Текст страницы.. Прокрути меня...

Много строк..

Много строк..

Много строк..

Много строк..

Итого

Виды позиционирования и их особенности.

`static`

Иначе называется «без позиционирования». В явном виде задаётся только если надо переопределить другое правило CSS.

`relative`

Сдвигает элемент относительно текущего места.

- Противоположные границы `left/right` (`top/bottom`) одновременно указать нельзя.
- Окружающие элементы ведут себя так, как будто элемент не сдвигался.

`absolute`

Визуально переносит элемент на новое место.

Новое место вычисляется по координатам `left/top/right/bottom` относительно ближайшего позиционированного родителя. Если такого родителя нет, то им считается окно.

- Ширина элемента по умолчанию устанавливается по содержимому.
- Можно указать противоположные границы `left/right` (`top/bottom`). Элемент растянется.
- Окружающие элементы заполняют освободившееся место.

`fixed`

Подвид абсолютного позиционирования, при котором элемент привязывается к координатам окна, а не документа.

При прокрутке он остаётся на том же месте.

Почитать

CSS-позиционирование по-настоящему глубоко в спецификации Visual Formatting Model, 9.3 и ниже.

Еще есть хорошее руководство CSS Positioning in 10 steps, которое охватывает основные типы позиционирования.

Задачи

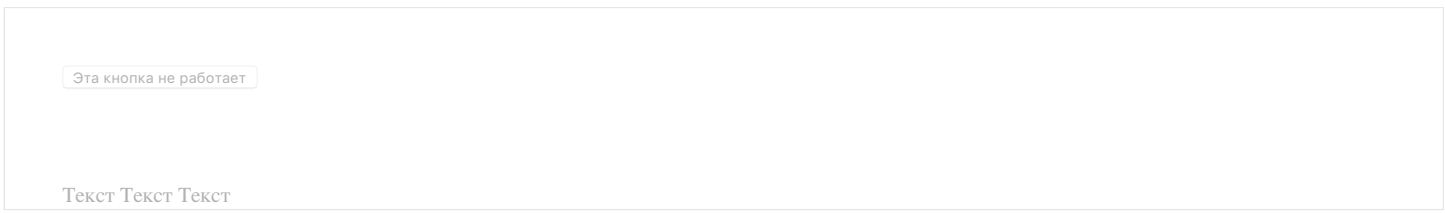
Модальное окно

важность: 5

Создайте при помощи HTML/CSS «модальное окно», то есть `DIV`, который полностью перекрывает документ и находится над ним.

При этом все элементы управления на документе перестают работать, т.к. клики попадают в `DIV`.

В примере ниже `DIV'у` дополнительно поставлен цвет фона и прозрачность, чтобы было видно перекрытие:



Браузеры: все основные, IE8+. Должно работать при прокрутке окна (проверьте).

Открыть песочницу для задачи.

К решению

Центрирование горизонтальное и вертикальное

В CSS есть всего несколько техник центрирования элементов. Если их знать, то большинство задач решаются просто.

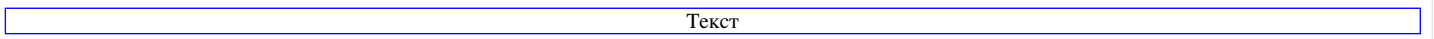
Горизонтальное

text-align

Для центрирования инлайновых элементов – достаточно поставить родителю `text-align: center` .:

```
<style>
  .outer {
    text-align: center;
    border: 1px solid blue;
  }
</style>

<div class="outer">Текст</div>
```



Для центрирования блока это уже не подойдёт, свойство просто не подействует. Например:

```
<style>
  .outer {
    text-align: center;
    border: 1px solid blue;
  }
  .inner {
    width: 100px;
    border: 1px solid red;
  }
</style>

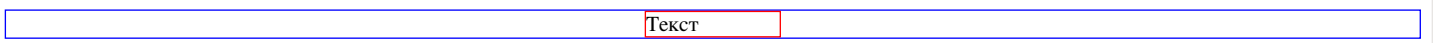
<div class="outer">
  <div class="inner">Текст</div>
</div>
```

margin: auto

Блок по горизонтали центрируется `margin: auto` :

```
<style>
  .outer {
    border: 1px solid blue;
  }
  .inner {
    width: 100px;
    border: 1px solid red;
    margin: auto;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```



В отличие от `width/height` , значение `auto` для `margin` само не появляется. Обычно `margin` равно конкретной величине для элемента, например `0` для `DIV` . Нужно поставить его явно.

Значение `margin-left:auto/margin-right:auto` заставляет браузер выделять под `margin` всё доступное сбоку пространство. А если и то и другое `auto` , то слева и справа будет одинаковый отступ, таким образом элемент окажется в середине. Детали вычислений описаны в разделе спецификации Calculating widths and margins .

Вертикальное

Для горизонтального центрирования всё просто. Вертикальное же изначальное не было предусмотрено в спецификации CSS и по сей день вызывает ряд проблем.

Есть три основных решения.

position:absolute + margin

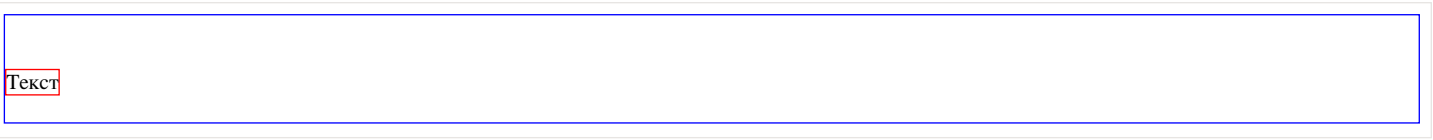
Центрируемый элемент позиционируем абсолютно и опускаем до середины по вертикали при помощи `top:50%` :

```
<style>
  .outer {
    position: relative;
    height: 5em;
    border: 1px solid blue;
  }

  .inner {
```

```
    position: absolute;
    top: 50%;
    border: 1px solid red;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```



Это, конечно, не совсем центр. По центру находится верхняя граница. Нужно ещё приподнять элемент на половину своей высоты.

Высота центрируемого элемента должна быть известна. Родитель может иметь любую высоту.

Если мы знаем, что это ровно одна строка, то её высота равна `line-height`.

Приподнимем элемент на пол-высоты при помощи **`margin-top`**:

```
<style>
  .outer {
    position: relative;
    height: 5em;
    border: 1px solid blue;
  }

  .inner {
    position: absolute;
    top: 50%;
    margin-top: -0.625em;
    border: 1px solid red;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```

Почему‑0.625em?

При стандартных настройках браузера высота строки `line-height: 1.25`, если поделить на два `1.25em / 2 = 0.625em`.

Конечно, высота может быть и другой, главное чтобы мы её знали заранее.

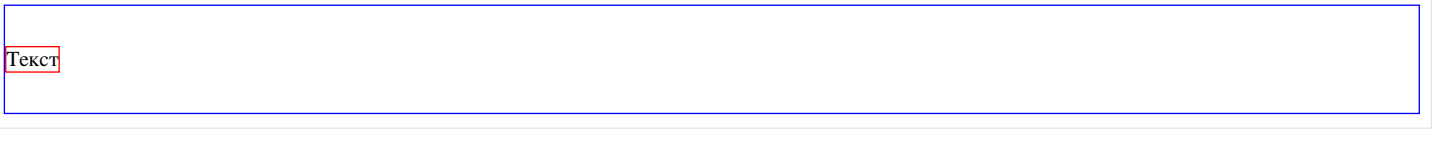
Можно аналогично центрировать и по горизонтали, если известен горизонтальный размер, при помощи `left:50%` и отрицательного `margin-left`.

Одна строка: line-height

Вертикально отцентрировать одну строку в элементе с известной высотой `height` можно, указав эту высоту в свойстве `line-height`:

```
<style>
  .outer {
    height: 5em;
    line-height: 5em;
    border: 1px solid blue;
  }
</style>

<div class="outer">
  <span style="border:1px solid red">Текст</span>
</div>
```



Это работает, но лишь до тех пор, пока строка одна, а если содержимое вдруг переносится на другую строку, то начинает выглядеть довольно уродливо.

Таблица с vertical-align

У свойства vertical-align, которое управляет вертикальным расположением элемента, есть два режима работы.

В таблицах свойство **`vertical-align`** указывает расположение *содержимого* ячейки.

Его возможные значения:

`baseline`

Значение по умолчанию.

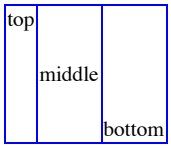
`middle`, `top`, `bottom`

Располагать содержимое посередине, вверху, внизу ячейки.

Например, ниже есть таблица со всеми 3-мя значениями:

```
<style>
  table {  border-collapse: collapse; }
  td {
    border: 1px solid blue;
    height: 100px;
  }
</style>

<table>
<tr>
  <td style="vertical-align: top">top</td>
  <td style="vertical-align: middle">middle</td>
  <td style="vertical-align: bottom">bottom</td>
</tr>
</table>
```

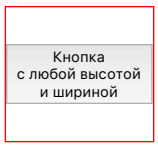


Обратим внимание, что в ячейке с `vertical-align: middle` содержимое находится по центру. Таким образом, можно обернуть нужный элемент в таблицу размера `width:100%;height:100%` с одной ячейкой, у которой указать `vertical-align:middle`, и он будет отцентрирован.

Но мы рассмотрим более красивый способ, который поддерживается во всех современных браузерах, и в IE8+. В них не обязательно делать таблицу, так как доступно значение `display:table-cell`. Для элемента с таким `display` используются те же алгоритмы вычисления ширины и центрирования, что и в `TD`. И, в том числе, работает `vertical-align`:

Пример центрирования:

```
<div style="display: table-cell; vertical-align: middle;  height: 100px; border: 1px solid red">
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
```



Этот способ замечателен тем, что он не требует знания высоты элементов.

Однако у него есть особенность. Вместе с `vertical-align` родительский блок получает табличный алгоритм вычисления ширины и начинает подстраиваться под содержимое. Это не всегда желательно.

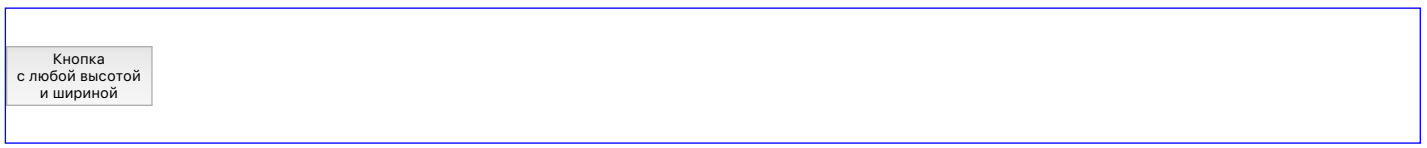
Чтобы его растянуть, нужно указать `width` явно, например: `300px`:

```
<div style="display: table-cell; vertical-align: middle; height: 100px; width: 300px; border: 1px solid red">
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
```

Можно и в процентах, но в примере выше они не сработают, потому что структура таблицы «сломана» – ячейка есть, а собственно таблицы-то нет.

Это можно починить, завернув «псевдоячейку» в элемент с `display:table`, которому и поставим ширину:

```
<div style="display: table; width: 100%">
<div style="display: table-cell; vertical-align: middle; height: 100px; border: 1px solid blue">
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
</div>
```



Если дополнительно нужно горизонтальное центрирование – оно обеспечивается другими средствами, например `margin: 0 auto` для блочных элементов или `text-align:center` на родителе – для других.

Центрирование в строке с vertical-align

Для инлайновых элементов (`display:inline/inline-block`), включая картинки, свойство `vertical-align` центрирует *сам инлайн-элемент в окружающем его тексте*.

В этом случае набор значений несколько другой:

Картинка размером в `30px`, значения `vertical-align`:

`baseline`(по умолчанию) 
`middle`(по середине) 
`sub`вровень с `<sub>` 
`super`вровень с `<sup>` 
`text-top`(верхняя граница вровень с текстом) 
`text-bottom`(нижняя граница вровень с текстом) 

Это можно использовать и для центрирования, если высота родителя известна, а центрируемого элемента – нет.

Допустим, высота внешнего элемента `120px` . Укажем её в свойстве `line-height` :

```
<style>
  .outer {
    line-height: 120px;
  }
  .inner {
    display: inline-block; /* центрировать..*/
    vertical-align: middle;  /* ..по вертикали */
    line-height: 1.25; /* переопределить высоту строки на обычную */
    border: 1px solid red;
  }
</style>
<div class="outer" style="height: 120px;border: 1px solid blue">
  <span class="inner">Центрирован<br>вертикально</span>
</div>
```

Центрирован
вертикально

Работает во всех браузерах и IE8+.

Свойство `line-height` наследуется, поэтому надо знать «правильную» высоту строки и переопределять её для `inner` .

Центрирование с vertical-align без таблиц

Если центрирование должно работать для любой высоты родителя и центрируемого элемента, то обычно используют таблицы или `display:table-cell` с `vertical-align` .

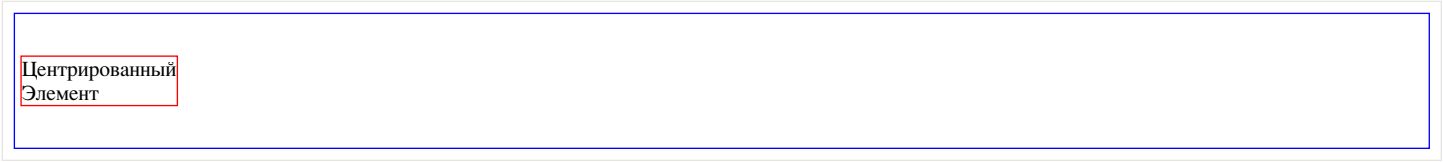
Если центрируются не-блочные элементы, например `inline` или `inline-block` , то `vertical-align` может решить задачу без всяких таблиц. Правда, понадобится вспомогательный элемент (можно через `:before`).

Пример:

```
<style>
.before {
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
}
</style>

<div class="outer" style="height:100px;border:1px solid blue">
  <span class="before"></span>
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```



- Перед центрируемым элементом помещается вспомогательный инлайн-блок `before` , занимающий всю возможную высоту.
- Центрируемый блок выровнен по его середине.

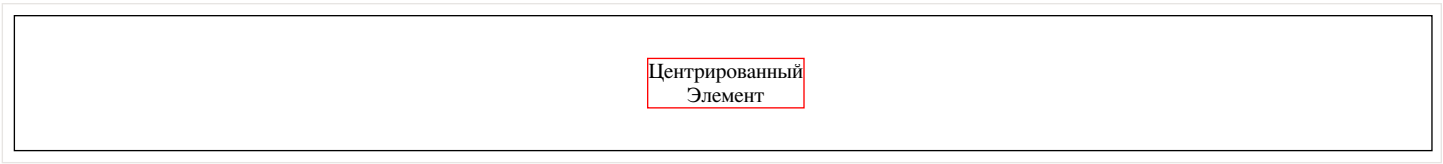
Для всех современных браузеров и IE8 можно добавить вспомогательный элемент через `:before` :

```
<style>
.outer:before {
  content: '';
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
}

/* добавим горизонтальное центрирование */
.outer {
  text-align: center;
}
</style>

<div class="outer" style="height:100px; width: 100%; border:1px solid black">
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```



В пример выше добавлено также горизонтальное центрирование `text-align: center` . Но вы можете видеть, что на самом деле внутренний элемент не центрирован горизонтально, он немного сдвинут вправо.

Это происходит потому, что центрируется *весь текст*, а перед `inner` находится пробел, который занимает место.

Варианта два:

1. Убрать лишний пробел между `div` и началом `inner` , будет `<div class="outer"><span class="inner">...` .
2. Оставить пробел, но сделать отрицательный `margin-left` у `inner` , равный размеру пробела, чтобы `inner` сместился левее.

Второе решение:

```
<style>
.outer:before {
  content: '';
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
  margin-left: -0.35em;
}

.outer {
  text-align: center;
}
</style>

<div class="outer" style="height:100px; width: 100%; border:1px solid black">
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```

Итого

Обобщим решения, которые обсуждались в этой статье.

Для горизонтального центрирования:

- `text-align: center` – центрирует инлайн-элементы в блоке.
- `margin: 0 auto` – центрирует блок внутри родителя. У блока должна быть указана ширина.

Для вертикального центрирования одного блока внутри другого:

Если размер центрируемого элемента известен, а родителя – нет

Родителю `position:relative`, потомку `position:absolute; top:50%` и `margin-top:-<половина-высоты-потомка>`. Аналогично можно отцентрировать и по горизонтали.

Если нужно отцентрировать одну строку в блоке, высота которого известна

Поставить блоку `line-height: <высота>`. Нужны конкретные единицы высоты (`px`, `em` ...). Значение `line-height:100%` не будет работать, т.к. проценты берутся не от высоты блока, а от текущей `line-height`.

Высота родителя известна, а центрируемого элемента – нет.

Поставить `line-height` родителю во всю его высоту, а потомку поставить `display:inline-block`.

Высота обоих элементов неизвестна.

Два варианта:

1. Сделать элемент-родитель ячейкой таблицы при помощи `display:table-cell` (IE8) или реальной таблицы, и поставить ему `vertical-align:middle`. Отлично работает, но мы имеем дело с таблицей вместо обычного блока.
2. Решение с вспомогательным элементом `outer:before` и инлайн-блоками. Вполне универсально и не создаёт таблицу.

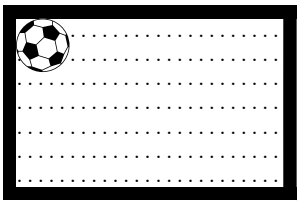
Задачи

Поместите мяч в центр поля (CSS)

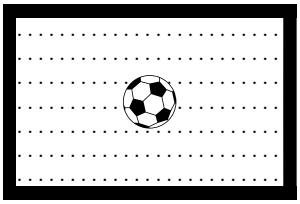
важность: 5

Поместите мяч в центр поля при помощи CSS.

Исходный код:



Используйте CSS, чтобы поместить мяч в центр:



- CSS для центрирования может использовать размеры мяча.
- CSS для центрирования не должен опираться на конкретный размер поля.

Открыть песочницу для задачи.

К решению

Форма + модальное окно

важность: 5

Создайте при помощи HTML/CSS форму для логина в модальном окне.

Текст Текст Текст

Эта кнопка не работает

Текст Текст Текст

Добро пожаловать!
Логин
Пароль
Войти

Требования:

- Кнопки окна вне формы не работают (даже на левый край нажать нельзя).
- Полупрозрачный голубой «экран» отстоит от границ на `20px` .
- Форма центрирована вертикально и горизонтально, её размеры фиксированы.
- Посетитель может менять размер окна браузера, геометрия должна сохраняться.
- Не ломается при прокрутке.

Браузеры: все основные, IE8+.

Открыть песочницу для задачи.

К решению

vertical-align + table-cell + position = ?

важность: 5

В коде ниже используется вертикальное центрирование при помощи `table-cell + vertical-align` .

Почему оно не работает? Нажмите на просмотр, чтобы увидеть (стрелка должна быть в центре по вертикали).

```
<style>
  .arrow {
    position: absolute;
    height: 60px;
    border: 1px solid black;
    font-size: 28px;

    display: table-cell;
    vertical-align: middle;
  }
</style>

<div class="arrow">«</div>
```

Как починить центрирование при помощи CSS? Свойства `position/height` менять нельзя.

К решению

Свойства "font-size" и "line-height"

Здесь мы рассмотрим, как соотносятся размеры шрифта и строки, и как их правильно задавать.

font-size и line-height

- `font-size` – *размер шрифта*, в частности, определяющий высоту букв.
- `line-height` – *высота строки*.

Для наглядности посмотрим пример HTML, в котором шрифт и размер строки одинаковы:

```
<style>
body {
    font-size: 30px;
    font-family: monospace;
    line-height: 30px;
  }
</style>

<div style="outline: 1px dotted red">Ёрш р</div>
<div style="outline: 1px dotted red">Ёрш Ё</div>
```

Размер шрифта `font-size` – это абстрактное значение, которое привязано к шрифту, и даётся в типографских целях.

Обычно оно равно расстоянию от самой верхней границы букв до самой нижней, исключая «нижние хвосты» букв, таких как `p`, `g`. Как видно из примера выше, при размере строки, равном `font-size`, строка не будет размером точно «под букву».

В зависимости от шрифта, «хвосты» букв при этом могут вылезать, правые буквы `Ё` и `p` в примере выше пересекаются как раз поэтому.

В некоторых особо размашистых шрифтах «хвосты букв» могут быть размером с саму букву, а то и больше. Но это, всё же исключение.

Обычно размер строки делают чуть больше, чем шрифт.

По умолчанию в браузерах используется специальное значение **`line-height:normal`**.

Оно означает, что браузер может принимать решение о размере строки самостоятельно. Как правило, оно будет в диапазоне `1.1 - 1.25`, но стандарт не гарантирует этого, он говорит лишь, что оно должно быть «разумным» (дословно – англ. reasonable).

Множитель для line-height

Значение `line-height` можно указать при помощи `px` или `em`, но гораздо лучше – задать его числом.

Значение-число интерпретируется как множитель относительно размера шрифта. Например, значение с множителем `line-height: 2` при `font-size: 16px` будет аналогично `line-height: 32px (=16px*2)`.

Однако, между множителем и точным значением есть одна существенная разница.

- Значение, заданное множителем, наследуется и применяется в каждом элементе относительно его размера шрифта.

  То есть, при `line-height: 2` означает, что высота строки будет равна удвоенному размеру шрифта, не важно какой шрифт.

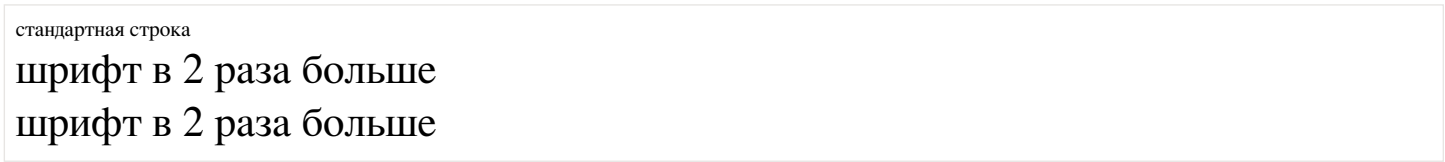
- Значение, заданное в единицах измерения, запоминается и наследуется «как есть».

  Это означает, что `line-height: 32px` будет всегда жёстко задавать высоту строки, даже если шрифт во вложенных элементах станет больше или меньше текущего.

Давайте посмотрим, как это выглядит, на примерах:

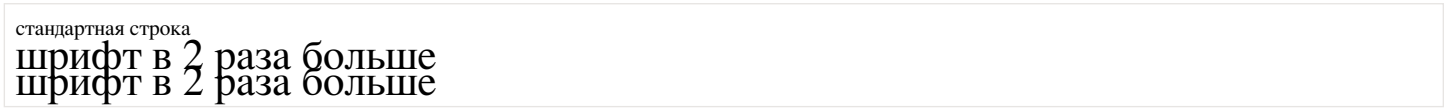
Множитель, `line-height:1.25`

```
<div style="line-height: 1.25">
  стандартная строка
  <div style="font-size:2em">
    шрифт в 2 раза больше<br>
    шрифт в 2 раза больше
  </div>
</div>
```



Конкретное значение, `line-height:1.25em`

```
<div style="line-height: 1.25em">
  стандартная строка
  <div style="font-size:2em">
    шрифт в 2 раза больше<br>
    шрифт в 2 раза больше
  </div>
</div>
```



Какой вариант выглядит лучше? Наверно, первый. В нём размер строки более-менее соответствует шрифту, поскольку задан через множитель.

В обычных ситуациях рекомендуется использовать именно множитель, за исключением особых случаев, когда вы действительно знаете что делаете.

Синтаксис font: size/height family

Установить `font-size` и `line-height` можно одновременно.

Соответствующий синтаксис выглядит он так:

```
font: 20px/1.5 Arial,sans-serif;
```

При этом нужно обязательно указать сам шрифт, например `Arial,sans-serif`. Укороченный вариант `font: 20px/1.5` работать не будет.

Дополнительно можно задать и свойства `font-style`, `font-weight`:

```
font: italic bold 20px/1.5 Arial,sans-serif;
```

Итого

`line-height`

Размер строки, обычно он больше размера шрифта. При установке множителем рассчитывается каждый раз относительно текущего шрифта, при установке в единицах измерения – фиксируется.

`font-size`

Размер шрифта. Если сделать блок такой же высоты, как шрифт, то хвосты букв будут вылезать из-под него.

`font: 125%/1.5 FontFamily`

Даёт возможность одновременно задать размер, высоту строки и, собственно, сам шрифт.

Свойство white-space

Свойство `white-space` позволяет сохранять пробелы и переносы строк.

У него есть два известных значения:

- `white-space: normal` – обычное поведение
- `white-space: pre` – текст ведёт себя, как будто оформлен в тег `pre`.

Но браузеры поддерживают и другие, которые также бывают очень полезны.

pre / nowrap

Встречаем первую «сладкую парочку» – `pre` и `nowrap`.

Оба этих значения меняют стандартное поведение HTML при работе с текстом:

`pre`:

- Сохраняет пробелы.
- Переносит текст при явном разрыве строки.

`nowrap`

- Не сохраняет пробелы.
- Игнорирует явные разрывы строки (не переносит текст).

Оба этих значения поддерживаются кросс-браузерно.

Их основной недостаток – текст может вылезти из контейнера.

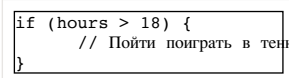
Для примера, рассмотрим следующий фрагмент кода:

```
if (hours > 18) {
      // Пойти поиграть в теннис
}
```

`white-space: pre`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```

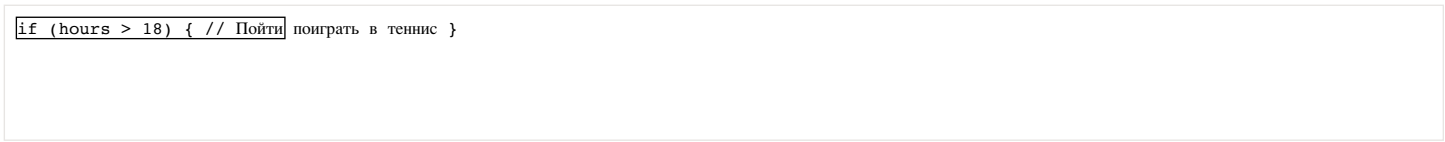


Здесь пробелы и переводы строк сохранены. В HTML этому значению `white-space` соответствует тег `PRE`.

`white-space: nowrap`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:nowrap">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```



Здесь переводы строки проигнорированы, а подряд идущие пробелы, если присмотреться – сжаты в один (например, перед комментарием `//`).

Допустим, мы хотим разрешить посетителям публиковать код на сайте, с сохранением разметки. Но тег `PRE` и описанные выше значения `white-space` для этого не подойдут!

Злой посетитель Василий Пупкин может написать такой текст, который вылезет из контейнера и поломает вёрстку страницы.

Можно скрыть вылезшее значение при помощи `overflow-x: hidden` или сделать так, чтобы была горизонтальная прокрутка, но, к счастью, есть другие значения `white-space`, специально для таких случаев.

pre-wrap/pre-line

`pre-wrap`

То же самое, что `pre`, но переводит строку, если текст вылезает из контейнера.

`pre-line`

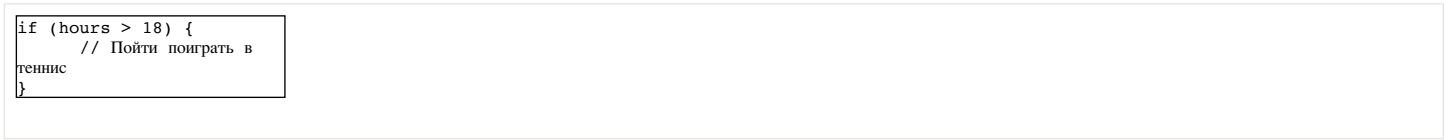
То же самое, что `pre`, но переводит строку, если текст вылезает из контейнера и не сохраняет пробелы.

Оба поведения отлично прослеживаются на примерах:

`white-space: pre-wrap`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre-wrap">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```

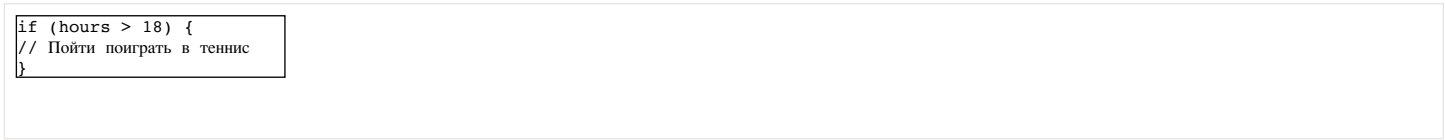


Отличный выбор для безопасной вставки кода посетителями.

`white-space: pre-line`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre-line">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```



Сохранены переводы строк, ничего не вылезает, но пробелы интерпретированы в режиме обычного HTML.

Свойство "outline"

Свойство `outline` задаёт дополнительную рамку вокруг элемента, *за пределами его CSS-блока*. Поддерживается во всех браузерах, IE8+.

Для примера, рассмотрим его вместе с обычной рамкой `border`:

```
<div style="border:3px solid blue; outline: 3px solid red">
 Элемент
</div>
```

- В отличие от **border**, рамка **outline** не участвует в блочной модели CSS. Она не занимает места и не меняет размер элемента. Поэтому его используют, когда хотят добавить рамку без изменения других CSS-параметров.
- Также, в отличие от border, рамку outline можно задать только со всех сторон: свойств outline-top, outline-left не существует.

Так как outline находится за границами элемента – **outline**-рамки соседей могут перекрывать друг друга:

```
<div style="outline: 3px solid green">
 Элемент
</div>
<div style="outline: 3px solid red">
 Элемент
</div>
```

В примере выше верхняя рамка нижнего элемента находится на территории верхнего и наоборот.

Все браузеры, кроме IE9-, также поддерживают свойство outline-offset, задающее отступ outline от внешней границы элемента:

```
<div style="border:3px solid blue; outline: 3px solid red; outline-offset:5px">
 Везде, кроме IE9-, между рамками будет расстояние 5px
</div>
```

Ещё раз заметим, что основная особенность outline – в том, что при наличии outline-offset или без него – он не занимает места в блоке.

Поэтому его часто используют для стилей :hover и других аналогичных, когда нужно выделить элемент, но чтобы ничего при этом не прыгало.

Свойство "box-sizing"

Свойство box-sizing может принимать одно из двух значений – border-box или content-box. В зависимости от выбранного значения браузер по-разному трактует значение свойств width/height.

Значения box-sizing

content-box

Это значение по умолчанию. В этом случае свойства width/height обозначают то, что находится *внутри padding*.

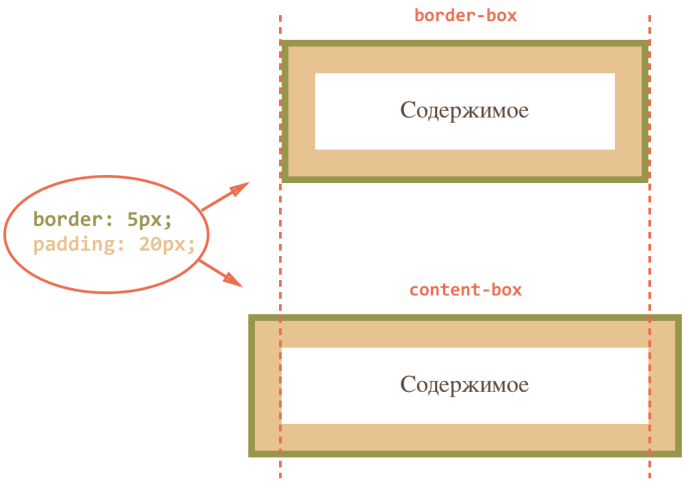
border-box

Значения width/height задают высоту/ширину *всего элемента*.

Для большей наглядности посмотрим на картинку этого div в зависимости от box-sizing:

```
/*+ no-beautify */
div {
  width: 200px;
  height: 100px;
  box-sizing: border-box (вверху) | content-box (внизу);

  padding: 20px;
  border:5px solid brown;
}
```



В верхнем случае браузер нарисовал весь элемент размером в `width` x `height`, в нижнем – интерпретировал `width/height` как размеры внутренней области.

Исторически сложилось так, что по умолчанию принят `content-box`, а `border-box` некоторые браузеры используют если не указан `DOCTYPE`, в режиме совместимости.

Но есть как минимум один случай, когда явное указание `border-box` может быть полезно: растягивание элемента до ширины родителя.

Пример: подстроить ширину к родителю

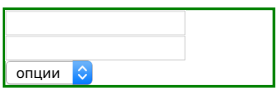
Задача: подогнать элемент по ширине внешнего элемента, чтобы он заполнял всё его пространство. Без привязки к конкретному размеру элемента в пикселях.

Например, мы хотим, чтобы элементы формы ниже были одинакового размера:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input,
  form select {
    display: block;
    padding-left: 5px;
    /* padding для красоты */
  }
</style>

<form>
  <input>
  <input>
  <select>
    <option>опции</option>
  </select>
</form>
```



Как сделать, чтобы элементы растянулись чётко по ширине `FORM`? Попробуйте добиться этого самостоятельно, перед тем как читать дальше.

Попытка width:100%

Первое, что приходит в голову – поставить всем `INPUT'ам` ширину `width: 100%`.

Попробуем:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input, form select {
    display: block;
    padding-left: 5px;
    width: 100%;
  }
</style>

<form>
  <input>
  <input>
  <select><option>опции</option></select>
</form>
```

Как видно, не получается. Элементы вылезают за пределы родителя.

Причина – ширина элемента 100% по умолчанию относится к внутренней области, не включающей **padding** и **border**. То есть, внутренняя область растягивается до `100%` родителя, и к ней снаружи прибавляются `padding/border`, которые и вылезают.

Есть два решения этой проблемы.

Решение: дополнительный элемент

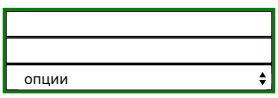
Можно убрать `padding/border` у элементов `INPUT/SELECT` и завернуть каждый из них в дополнительный `DIV`, который будет обеспечивать дизайн:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }
  /* убрать padding/border */

  form input,
  form select {
    padding: 0;
    border: 0;
    width: 100%;
  }
  /* внешний div даст дизайн */

  form div {
    padding-left: 5px;
    border: 1px solid black;
  }
</style>

<form>
  <div>
    <input>
  </div>
  <div>
    <input>
  </div>
  <div>
    <select>
      <option>опции</option>
    </select>
  </div>
</form>
```



В принципе, это работает. Но нужны дополнительные элементы. А если мы делаем дерево или большую редактируемую таблицу, да и вообще – любой интерфейс, где элементов и так много, то лишние нам точно не нужны.

Кроме того, такое решение заставляет пожертвовать встроенным в браузер дизайном элементов `INPUT/SELECT`.

Решение: box-sizing

Существует другой способ, гораздо более естественный, чем предыдущий.

При помощи **`box-sizing: border-box`** мы можем сказать браузеру, что ширина, которую мы ставим, относится к элементу полностью, включая **border** и **padding**:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input, form select {
    display: block;
    padding-left: 5px;
    -moz-box-sizing: border-box; /* в Firefox нужен префикс */
    box-sizing: border-box;
    width: 100%;
  }
</style>

<form>
  <input>
  <input>
  <select><option>опции</option></select>
</form>
```

Мы сохранили «родную» рамку вокруг `INPUT/SELECT` и не добавили лишних элементов. Всё замечательно.

Свойство `box-sizing` поддерживается в IE начиная с версии 8.

Свойство "margin"

Свойство `margin` задаёт отступы вокруг элемента. У него есть несколько особенностей, которые мы здесь рассмотрим.

Объединение отступов

Вертикальные отступы поглощают друг друга, горизонтальные – нет.

Например, вот документ с вертикальными и горизонтальными отступами:

```
<body style="background: #aef">
  <p style="margin:20px; background:white">

    <span style="margin:20px; background:orange">Горизонтальный 20px</span>
    ← 40px →
    <span style="margin:20px; background:orange">20px Отступ </span>

  </p>
  <p style="margin:15px; background:white">Вертикальный 20px</p>
</body>
```

Горизонтальный 20px ← 40px → 20px Отступ

Вертикальный 20px

Расстояние по горизонтали между элементами `SPAN` равно `40px` , так как горизонтальные отступы по `20px` сложились.

А вот по вертикали расстояние от `SPAN` до `P` равно `20px` : из двух вертикальных отступов выбирается больший `max(20px, 15px) = 20px` и применяется.

Отрицательные margin-top/left

Отрицательные значения `margin-top/margin-left` смещают элемент со своего обычного места.

В CSS есть другой способ добиться похожего эффекта – а именно, `position:relative` . Но между ними есть одно принципиальное различие.

При сдвиге через **`margin`** соседние элементы занимают освободившееся пространство, в отличие от **`position: relative`** , при котором элемент визуально сдвигается, но место, где он был, остается «занятым».

То есть, элемент продолжает полноценно участвовать в потоке.

Пример: вынос заголовка

Например, есть документ с информационными блоками:

```
<style>
  div {
    border: 1px solid blue;
    margin: 2em;
    font: .8em/1.25 Arial, sans-serif;
  }

  h2 {
    background: #aef;
    margin: 0 0 0.8em 0;
  }
</style>

<div>
  <h2>Общие положения</h2>

  <p>Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающ
</div>

<div>
  <h2>Общие обязанности водителей</h2>

  <p>Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:</p>
  <ul>
    <li>водительское удостоверение на право управления транспортным средством соответствующей категории;</li>
    <li>...и так далее...</li>
  </ul>
</div>
```

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

Использование отрицательного **margin-top** позволяет вынести заголовки над блоком.

```
/* вверх чуть больше, чем на высоту строки (1.25em) */

h2 {
  margin-top: -1.3em;
}
```

Результат:

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

А вот, что бы было при использовании `position`:

```
h2 {
  position: relative;
  top: -1.3em;
}
```

Результат:

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

При использовании **position**, в отличие от **margin**, на месте заголовков, внутри блоков, осталось пустое пространство.

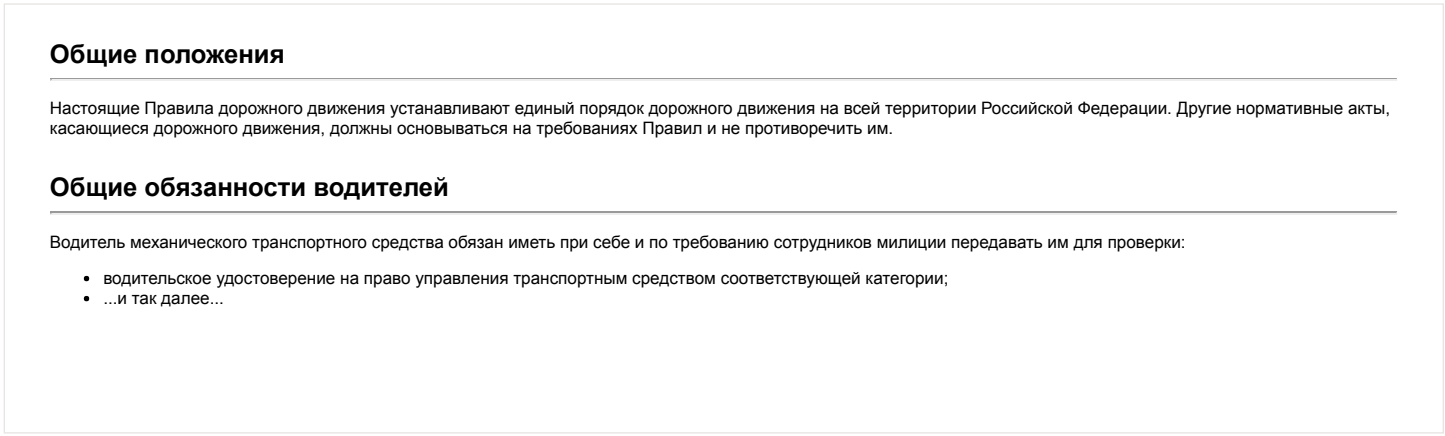
Пример: вынос отчерка

Организуем информацию чуть по-другому. Пусть после каждого заголовка будет отчерк:

```
<div>
  <h2>Заголовок</h2>
  <hr>

  <p>Текст Текст Текст.</p>
</div>
```

Пример документа с такими отчерками:

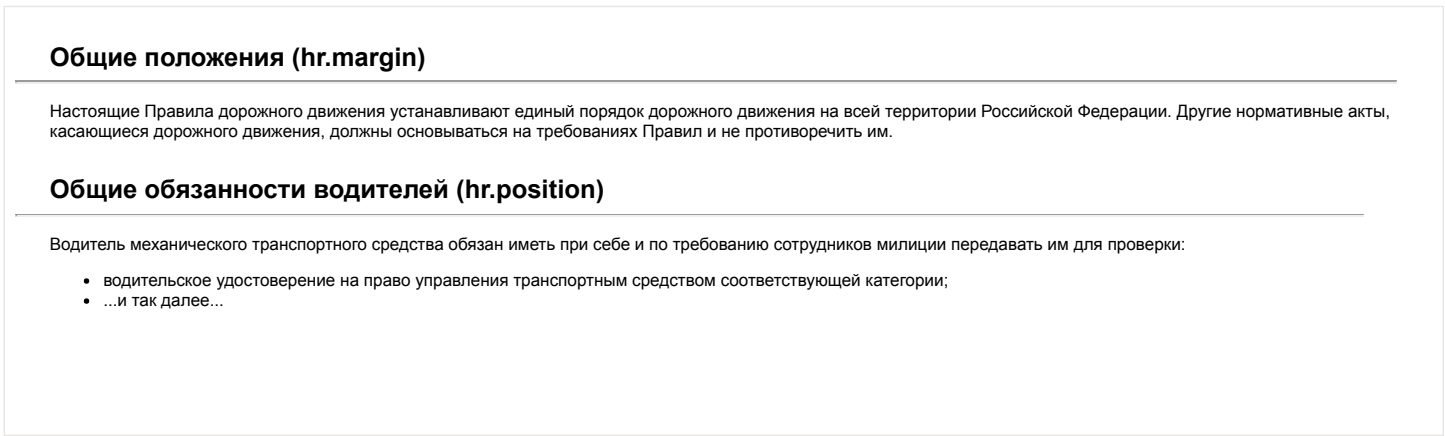


Для красоты мы хотим, чтобы отчерк `HR` начинался левее, чем основной текст. Отрицательный `margin-left` нам поможет:

```
/*+ no-beautify */
hr.margin { margin-left: -2em; }

/* для сравнения */
hr.position { position: relative; left: -2em; }
```

Результат:



Обратите внимание на разницу между методами сдвига!

- `hr.margin` сначала сдвинулся, а потом нарисовался до конца блока.
- `hr.position` сначала нарисовался, а потом сдвинулся – в результате справа осталось пустое пространство.

Уже отсюда видно, что отрицательные `margin` – исключительно полезное средство позиционирования!

Отрицательные margin-right/bottom

Отрицательные `margin-right/bottom` ведут себя по-другому, чем `margin-left/top` . Они не сдвигают элемент, а «укорачивают» его.

То есть, хотя сам размер блока не уменьшается, но следующий элемент будет думать, что он меньше на указанное в `margin-right/bottom` значение.

Например, в примере ниже вторая строка налезает на первую:

```
<div style="border: 1px solid blue; margin-bottom: -0.5em">
  Первый
</div>

<div style="border: 1px solid red">
```

```
  Второй div думает, что высота первого на 0.5em меньше
</div>
```

Первый
Второй div думает, что высота первого на 0.5em меньше

Это используют, в частности для красивых вносок, с приданием иллюзии глубины.

Например:

У DIV'a с холодильниками стоит margin-bottom: -1em

Наши холодильники - самые лучшие холодильники в мире! Наши холодильники - самые лучшие холодильники в мире! Наши холодильники - самые лучшие холодильники в мире! Наши холодильники - самые лучшие холодильники в мире!

Так считают: 5 человек Оставьте свой отзыв!

Итого

- Отрицательные `margin-left/top` сдвигают элемент влево-вверх. Остальные элементы это учитывают, в отличие от сдвига через `position`.
- Отрицательные `margin-right/bottom` заставляют другие элементы думать, что блок меньше по размеру справа-внизу, чем он на самом деле.

Отличная статья на тему отрицательных `margin`: The Definitive Guide to Using Negative Margins

Задачи

Нерабочие margin?

важность: 3

В примере ниже находится блок `.block` фиксированной высоты, а в нём – прямоугольник `.spacer`.

При помощи `margin-left: 20%` и `margin-right: 20%`, прямоугольник центрирован в родителе по горизонтали. Это работает.

Далее делается попытка при помощи свойств `height: 80%`, `margin-top: 10%` и `margin-bottom: 10%` расположить прямоугольник в центре по вертикали, чтобы сам элемент занимал `80%` высоты родителя, а сверху и снизу был одинаковый отступ.

Однако, как видите, это не получается. Почему? Как поправить?

```
<style>
  .block {
    height: 150px;

    border: 1px solid #CCC;
    background: #eee;
  }

  .spacer {
    margin-left: 20%;
    margin-right: 20%;

    height: 80%;
    margin-top: 10%;
    margin-bottom: 10%;

    border: 1px solid black;
    background: #FFF;
  }
</style>

<div class="block">
  <div class="spacer"></div>
</div>
```

К решению

Расположить текст внутри INPUT

важность: 5

Создайте `<input type="password">` с цветной подсказкой внутри (должен правильно выглядеть, не обязан работать):

Добро пожаловать
Скажи пароль, друг
.. и заходи

В дальнейшем мы сможем при помощи JavaScript сделать, чтобы текст при клике пропадал. Получится красивая подсказка.

P.S. Обратите внимание: `type="password"` ! То есть, просто `value` использовать нельзя, будут звёздочки. Кроме того, подсказка, которую вы реализуете, может быть как угодно стилизована.

P.P.S. Вокруг `INPUT` с подсказкой не должно быть лишних отступов, блоки до и после продолжают идти в обычном потоке.

Открыть песочницу для задачи.

К решению

Лишнее место под IMG

Иногда под `IMG` «вдруг» появляется лишнее место. Посмотрим на эти грабли внимательнее, почему такое бывает и как этого избежать.

Демонстрация проблемы

Например:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

Посмотрите внимательно! Вы видите расстояние между рамками снизу? Это потому, что браузер резервирует дополнительное место под инлайновым элементом, чтобы туда выносить «хвосты» букв.

Вот так выглядит та же картинка с выступающим вниз символом рядом:

В примере картинка обёрнута в красный `TD`. Таблица подстраивается под размер содержимого, так что проблема явно видна. Но она касается не только таблицы. Аналогичная ситуация возникнет, если вокруг `IMG` будет другой элемент с явно указанным размером, «облегающий» картинку по высоте. Браузер постарается зарезервировать место под `IMG`, хотя оно там не нужно.

Решение: сделать элемент блочным

Самый лучший способ избежать этого – поставить `display: block` таким картинкам:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  img {
    display: block
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

Под блочным элементом ничего не резервируется. Проблема исчезла.

Решение: задать vertical-align

А что, если мы, по каким-то причинам, *не хотим* делать элемент блочным?

Существует ещё один способ избежать проблемы – использовать свойство vertical-align.

Если установить **`vertical-align`** в **`top`**, то инлайн-элемент будет отпозиционирован по верхней границе текущей строки.

При этом браузер не будет оставлять место под изображением, так как запланирует продолжение строки сверху, а не снизу:

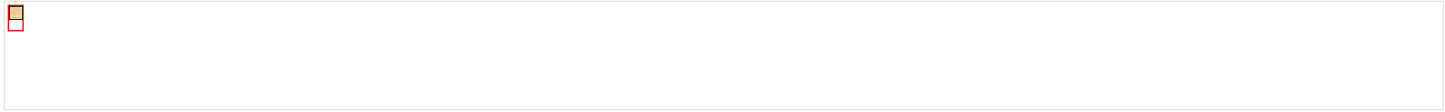
```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  img {
    vertical-align: top
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

А вот, как браузер отобразит соседние символы в этом случае: `p<img src="square.png">p`

С другой стороны, сама строка никуда не делась, изображение по-прежнему является её частью, а браузер планирует разместить другое текстовое содержимое рядом, хоть и сверху. Поэтому если изображение маленькое, то произойдёт дополнение пустым местом до высоты строки:

Например, для `<img src="square.png" style="height:10px">`:



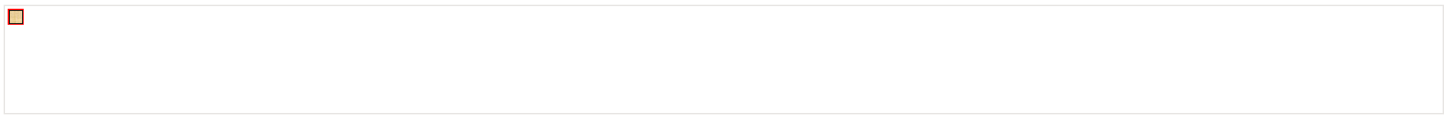
Таким образом, требуется уменьшить еще и `line-height` контейнера. Окончательное решение для маленьких изображений с `vertical-align`:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  td {
    line-height: 1px
  }

  img {
    vertical-align: top
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png" style="height:10px">
    </td>
  </tr>
</table>
```

Результат:



Итого

- Пробелы под картинками появляются, чтобы оставить место под «хвосты» букв в строке. Строка «подразумевается», т.к. `display:inline`.
- Можно избежать пробела, если изменить `display`, например, на `block`.
- Альтернатива: `vertical-align:top` (или `bottom`), но для маленьких изображений может понадобиться уменьшить `line-height`, чтобы контейнер не оставлял место под строку.

Свойство "overflow"

Свойство `overflow` управляет тем, как ведёт себя содержимое блочного элемента, если его размер превышает допустимую длину/ширину.

Обычно блок увеличивается в размерах при добавлении в него элементов, заключая в себе всех потомков.

Но что, если высота/ширина указаны явно? Тогда блок не может увеличиться, и содержимое «переполняет» блок. Его отображение в этом случае задаётся свойством `overflow`.

Возможные значения

- `visible` (по умолчанию)
- `hidden`
- `scroll`
- `auto`

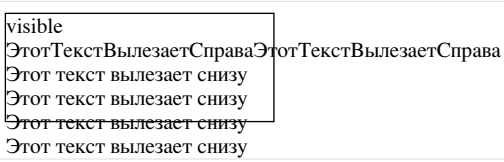
visible

Если не ставить `overflow` явно или поставить `visible`, то содержимое отображается за границами блока.

Например:

```
<style>
  .overflow {
    /* overflow не задан */
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  visible ЭтотТекстВылезаетСправаЭтотТекстВылезаетСправа
  Этот текст вылезает снизу Этот текст вылезает снизу
  Этот текст вылезает снизу Этот текст вылезает снизу
</div>
```



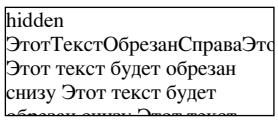
Как правило, такое переполнение указывает на ошибку в вёрстке. Если содержимое может быть больше контейнера – используют другие значения.

hidden

Переполняющее содержимое не отображается.

```
<style>
  .overflow {
    overflow: hidden;
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  hidden ЭтотТекстОбрезанСправаЭтотТекстОбрезанСправа
  Этот текст будет обрезан снизу Этот текст будет обрезан снизу
  Этот текст будет обрезан снизу Этот текст будет обрезан снизу
</div>
```



Вылезающее за границу содержимое становится недоступно.

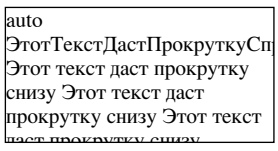
Это свойство иногда используют от лени, когда какой-то элемент дизайна немного вылезает за границу, и вместо исправления вёрстки его просто скрывают. Как правило, долго оно не живёт, вёрстку всё равно приходится исправлять.

auto

При переполнении отображается полоса прокрутки.

```
<style>
  .overflow {
    overflow: auto;
    width: 200px;
    height: 100px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  auto ЭтотТекстДастПрокруткуСправаЭтотТекстДастПрокруткуСправа
  Этот текст даст прокрутку снизу Этот текст даст прокрутку снизу
  Этот текст даст прокрутку снизу
</div>
```

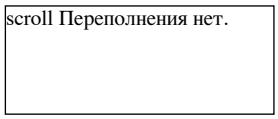


scroll

Полоса прокрутки отображается всегда.

```
<style>
  .overflow {
    overflow: scroll;
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  scroll
  Переполнения нет.
</div>
```



То же самое, что `auto`, но полоса прокрутки видна всегда, даже если переполнения нет.

overflow-x, overflow-y

Можно указать поведение блока при переполнении по ширине в `overflow-x` и высоте – в `overflow-y`:

```
<style>
  .overflow {
    overflow-x: auto;
    overflow-y: hidden;
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  ПоШиринеПолосаПрокруткиAutoПоШиринеПолосаПрокруткиAuto
  Этот текст вылезает снизу Этот текст вылезает снизу
  Этот текст вылезает снизу Этот текст вылезает снизу
</div>
```

ПоШиринеПолосаПрокрутки
Этот текст вылезает снизу
Этот текст вылезает снизу
Этот текст вылезает снизу

Итого

Свойства `overflow-x`/`overflow-y` (или оба одновременно: `overflow`) задают поведение контейнера при переполнении:

`visible`

По умолчанию, содержимое вылезает за границы блока.

`hidden`

Переполняющее содержимое невидимо.

`auto`

Полоса прокрутки при переполнении.

`scroll`

Полоса прокрутки всегда.

Кроме того, значение `overflow: auto | hidden` изменяет поведение контейнера, содержащего `float`'ы. Так как элемент с `float` находится вне потока, то обычно контейнер не выделяет под него место. Но если стоит такой `overflow`, то место выделяется, т.е. контейнер растягивается. Более подробно этот вопрос рассмотрен в статье Свойство "float".

Особенности свойства "height" в %

Обычно свойство `height`, указанное в процентах, означает высоту относительно внешнего блока.

Однако, всё не так просто. Интересно, что для произвольного блочного элемента `height` в процентах работать не будет!

Чтобы лучше понимать ситуацию, рассмотрим пример.

Пример

Наша цель – получить вёрстку такого вида:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

При этом блок с левой стрелкой должен быть отдельным элементом внутри контейнера.

Это удобно для интеграции с JavaScript, чтобы отлавливать на нём клики мыши.

То есть, HTML-код требуется такой:

```
<div class="container">
  <div class="toggler">
    <!-- стрелка влево при помощи CSS, ширина фиксирована -->
  </div>
```

```
  <div class="content">
    ...Текст...
  </div>
</div>
```

Как это реализовать? Подумайте перед тем, как читать дальше...

Придумали?.. Если да – продолжаем.

Есть разные варианты, но, возможно, вы решили сдвинуть `.toggler` влево, при помощи `float:left` (тем более что он фиксированной ширины) и увеличить до `height: 100%`, чтобы он занял всё пространство по вертикали.

Вы ещё не видите подвох? Смотрим внимательно, что будет происходить с **`height: 100%`** ...

Демо height:100% + float:left

CSS:

```
.container {
  border: 1px solid black;
}

.content {
  /* margin-left нужен, так как слева от содержимого будет стрелка */
  margin-left: 35px;
}

.toggler {
  /* Зададим размеры блока со стрелкой */
  height: 100%;
  width: 30px;
  float: left;

  background: #EEE url("arrow_left.png") center center no-repeat;
  border-right: #AAA 1px solid;
  cursor: pointer;
}
```

А теперь – посмотрим этот вариант в действии:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

Как видно, блок со стрелкой вообще исчез! Куда же он подевался?

Ответ нам даст спецификация CSS 2.1 пункт 10.5.

«Если высота внешнего блока вычисляется по содержимому, то высота в % не работает, и заменяется на **`height:auto`**. Кроме случая, когда у элемента стоит **`position:absolute`**.»

В нашем случае высота `.container` как раз определяется по содержимому, поэтому для `.toggler` проценты не действуют, а размер вычисляется как при `height:auto`.

Какая же она – эта автоматическая высота? Вспоминаем, что обычно размеры `float` определяются по содержимому (10.3.5). А содержимого-то в `.toggler` нет, так что высота нулевая. Поэтому этот блок и не виден.

Если бы мы точно знали высоту внешнего блока и добавили её в CSS – это решило бы проблему.

Например:

```
/*+ no-beautify */
.container {
  height: 200px; /* теперь height в % внутри будет работать */
}
```

Результат:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

...Но в данном случае этот путь неприемлем! Ведь мы не знаем, сколько будет текста и какой будет итоговая высота.

Поэтому решим задачу по-другому.

Правильно: height:100% + position:absolute

Проценты будут работать, если поставить `.toggler` свойство `position: absolute` и спозиционировать его в левом-верхнем углу `.container` (у которого сделать `position:relative`):

```
.container {
  position: relative;
  border: 1px solid black;
}

.content {
  margin-left: 35px;
}

.toggler {
  position: absolute;
  left: 0;
  top: 0;

  height: 100%;
  width: 30px;
  cursor: pointer;

  border-right: #AAA 1px solid;
  background: #EEE url("arrow_left.png") center center no-repeat;
}
```

Результат:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е — 1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

Проблема с height: 100%, проявляющаяся, когда у родительского элемента не установлен height, но указан min-height

Вам необходимо установить height: 1px для родителя, чтобы дочерний элемент смог занять всю высоту указанную в min-height.

```
.parent {
    min-height: 300px;
    height: 1px; /* Требуется, чтобы дочерний блок взял высоту 100% */
}

.child {
    height: 100%;
}
```

Итого

- Свойство `height`, указанное в %, работает только если для внешнего блока указана высота.
- Стандарт CSS 2.1 предоставляет обход этой проблемы, отдельно указывая, что проценты работают при `position:absolute`. На практике это часто выручает.
- Если у родительского элемента не установлено height, а указано min-height, то, чтобы дочерний блок занял 100% высоты, нужно родителю поставить свойство height: 1px;

Знаете ли вы селекторы?

CSS3-селекторы – фундаментально полезная вещь.

Даже если вы почему-то (старый IE?) не пользуетесь ими в CSS, есть много фреймворков для их кросс-браузерного использования CSS3 из JavaScript.

Поэтому эти селекторы необходимо знать.

Основные виды селекторов

Основных видов селекторов всего несколько:

- `*` – любые элементы.
- `div` – элементы с таким тегом.
- `#id` – элемент с данным `id`.
- `.class` – элементы с таким классом.
- `[name="value"]` – селекторы на атрибут (см. далее).

- `:visited` – «псевдоклассы», остальные разные условия на элемент (см. далее).

Селекторы можно комбинировать, записывая последовательно, без пробела:

- `.c1.c2` – элементы одновременно с двумя классами `c1` и `c2`
- `a#id.c1.c2:visited` – элемент `a` с данным `id`, классами `c1` и `c2`, и псевдоклассом `visited`

Отношения

В CSS3 предусмотрено четыре вида отношений между элементами.

Самые известные вы наверняка знаете:

- `div p` – элементы `p`, являющиеся потомками `div`.
- `div > p` – только непосредственные потомки

Есть и два более редких:

- `div ~ p` – правые соседи: все `p` на том же уровне вложенности, которые идут после `div`.
- `div + p` – первый правый сосед: `p` на том же уровне вложенности, который идёт сразу после `div` (если есть).

Посмотрим их на примере HTML:

```
<h3>Балтославянские языки</h3>

<ol id="languages">
  ...Вложенный OL/LI список языков...
</ol>
```

CSS-селекторы:

```
/*+ no-beautify */
#languages li {
  color: brown;   /* потомки #languages, подходящие под селектор  LI */
}

#languages > li {
  color: black;   /* первый уровень детей #languages подходящих под LI */
}

#e-slavic { font-style: italic; }

#e-slavic ~ li {  /* правые соседи #e-slavic с селектором LI */
  color: red;
}

#latvian {
  font-style: italic;
}

#latvian * {      /* потомки #latvian, подходяще под * (т.е. любые) */
  font-style: normal;
}

#latvian + li {  /* первый правый сосед #latvian с селектором LI */
 color: green;
}
```

Результат:

Балтославянские языки

1. Славянские языки
    1. Славянские микроязыки
    2. Праславянский язык
    3. *Восточнославянские языки* `#e-slavic`
    4. Западнославянские языки `#e-slavic ~ li`
    5. Южнославянские языки `#e-slavic ~ li`
    6. ... `#e-slavic ~ li`
2. Балтийские языки
    1. Литовский язык
    2. *Латышский язык* `#latvian`
        1. Латгальский язык `#latvian *`
    3. Прусский язык `#latvian + li`
    4. ... (следующий элемент уже не `#latvian + li`)

Фильтр по месту среди соседей

При выборе элемента можно указать его место среди соседей.

Список псевдоклассов для этого:

- `:first-child` – первый потомок своего родителя.
- `:last-child` – последний потомок своего родителя.
- `:only-child` – единственный потомок своего родителя, соседних элементов нет.
- `:nth-child(a)` – потомок номер `a` своего родителя, например `:nth-child(2)` – второй потомок. Нумерация начинается с `1`.
- `:nth-child(an+b)` – расширение предыдущего селектора через указание номера потомка формулой, где `a,b` – константы, а под `n` подразумевается любое целое число.

  Этот псевдокласс будет фильтровать все элементы, которые попадают под формулу при каком-либо `n`. Например: - `:nth-child(2n)` даст элементы номер `2`, `4`, `6` ..., то есть чётные.

  - `:nth-child(2n+1)` даст элементы номер `1`, `3` ..., то есть нечётные.
  - `:nth-child(3n+2)` даст элементы номер `2`, `5`, `8` и так далее.

Пример использования для выделения в списке:

- Древнерусский язык
- Древненовгородский диалект `li:nth-child(2n)`
- Западнорусский письменный язык `li:nth-child(3)`
- Украинский язык `li:nth-child(2n)`
- Белорусский язык
- Другие языки `li:nth-child(2n)`

```
/*+ hide="CSS к примеру выше"  no-beautify */
li:nth-child(2n) { /* чётные */
  background: #eee;
}

li:nth-child(3) {  /* 3-ий потомок */
  color: red;
}
```

- `:nth-last-child(a)`, `:nth-last-child(an+b)` – то же самое, но отсчёт начинается с конца, например `:nth-last-child(2)` – второй элемент с конца.

Фильтр по месту среди соседей с тем же тегом

Есть аналогичные псевдоклассы, которые учитывают не всех соседей, а только с тем же тегом:

- `:first-of-type`
- `:last-of-type`
- `:only-of-type`
- `:nth-of-type`
- `:nth-last-of-type`

Они имеют в точности тот же смысл, что и обычные `:first-child`, `:last-child` и так далее, но во время подсчёта игнорируют элементы с другими тегами, чем тот, к которому применяется фильтр.

Пример использования для раскраски списка `DT` «через один» и предпоследнего `DD`:

Первый dt
Описание dd
Второй dt `dt:nth-of-type(2n)`
Описание dd
Третий dt
Описание dd `dd:nth-last-of-type(2)`
Четвёртый dt `dt:nth-of-type(2n)`
Описание dd

```
/*+ hide="CSS к примеру выше"  no-beautify */
dt:nth-of-type(2n) {
  /* чётные dt (соседи с другими тегами игнорируются) */
  background: #eee;
}

dd:nth-last-of-type(2) {
  /* второй dd снизу */
  color: red;
}
```

Как видим, селектор `dt:nth-of-type(2n)` выбрал каждый второй элемент `dt`, причём другие элементы (`dd`) в подсчётах не участвовали.

Селекторы атрибутов

На атрибут целиком:

- `[attr]` – атрибут установлен,
- `[attr="val"]` – атрибут равен `val`.

На начало атрибута:

- `[attr^="val"]` – атрибут начинается с `val`, например `"value"`.
- `[attr|="val"]` – атрибут равен `val` *или* начинается с `val-`, например равен `"val-1"`.

На содержание:

- `[attr*="val"]` – атрибут содержит подстроку `val`, например равен `"myvalue"`.
- `[attr~="val"]` – атрибут содержит `val` как одно из значений через пробел.

Например: `[attr~="delete"]` верно для `"edit delete"` и неверно для `"undelete"` или `"no-delete"`.

На конец атрибута:

- `[attr$="val"]` – атрибут заканчивается на `val`, например равен `"myval"`.

Другие псевдоклассы

- `:not(селектор)` – все, кроме подходящих под селектор.
- `:focus` – в фокусе.
- `:hover` – под мышью.
- `:empty` – без детей (даже без текстовых).
- `:checked`, `:disabled`, `:enabled` – состояния `INPUT`.
- `:target` – этот фильтр сработает для элемента, `ID` которого совпадает с анкором `#...` текущего URL.

  Например, если на странице есть элемент с `id="intro"`, то правило `:target { color: red }` подсветит его в том случае, если текущий URL имеет вид `http://...#intro`.

Псевдоэлементы ::before, ::after

«Псевдоэлементы» – различные вспомогательные элементы, которые браузер записывает или может записать в документ.

При помощи *псевдоэлементов* `::before` и `::after` можно добавлять содержимое в начало и конец элемента:

```
<style>
  li::before {
    content: " [[ ";
  }

  li::after {
    content: " ]] ";
  }
</style>

Обратите внимание: содержимое добавляется <b>внутрь</b> LI.

<ul>
  <li>Первый элемент</li>
  <li>Второй элемент</li>
</ul>
```

Обратите внимание: содержимое добавляется **внутрь** LI.

- [[Первый элемент]]
- [[Второй элемент]]

Псевдоэлементы `::before` / `::after` добавили содержимое в начало и конец каждого `LI`.

`:before` или `::before`?

Когда-то все браузеры реализовали эти псевдоэлементы с одним двоеточием: `:after`/`:before`.

Стандарт с тех пор изменился и сейчас все, кроме IE8, понимают также современную запись с двумя двоеточиями. А для IE8 нужно по-прежнему одно.

Поэтому если вам важна поддержка IE8, то имеет смысл использовать одно двоеточие.

Практика

Вы можете использовать информацию выше как справочную для решения задач ниже, которые уже реально покажут, владеете вы CSS-селекторами или нет.

Задачи

Выберите элементы селектором

важность: 5

HTML-документ:

```
<input type="checkbox">
<input type="checkbox" checked>
<input type="text" id="message">

<h3 id="widget-title">Сообщения:</h3>
<ul id="messages">
  <li id="message-1">Сообщение 1</li>
  <li id="message-2">Сообщение 2</li>
  <li id="message-3" data-action="delete">Сообщение 3</li>
  <li id="message-4" data-action="edit do-not-delete">Сообщение 4</li>
  <li id="message-5" data-action="edit delete">Сообщение 5</li>
  <li><a href="#">...</a></li>
</ul>

<a href="http://site.com/list.zip">Ссылка на архив</a>
<a href="http://site.com/list.pdf">..И на PDF</a>
```

Задания:

1. Выбрать `input` типа `checkbox`.
2. Выбрать `input` типа `checkbox`, НЕ отмеченный.
3. Найти все элементы с `id=message` или `message-*`.
4. Найти все элементы с `id=message-*`.
5. Найти все ссылки с расширением `href="...zip"`.
6. Найти все элементы с атрибутом `data-action`, содержащим `delete` в списке (через пробел).
7. Найти все элементы, у которых ЕСТЬ атрибут `data-action`, но он НЕ содержит `delete` в списке (через пробел).
8. Выбрать все чётные элементы списка `#messages`.
9. Выбрать один элемент сразу за заголовком `h3#widget-title` на том же уровне вложенности.
10. Выбрать все ссылки, следующие за заголовком `h3#widget-title` на том же уровне вложенности.
11. Выбрать ссылку внутри последнего элемента списка `#messages`.

Открыть песочницу для задачи.

К решению

Отступ между элементами, размер одна строка

важность: 4

Есть список `UL/LI`.

```
Текст вверху без отступа от списка.
<ul>
  <li>Маша</li>
  <li>Паша</li>
  <li>Даша</li>
  <li>Женя</li>
  <li>Саша</li>
  <li>Гоша</li>
</ul>
Текст внизу без отступа от списка.
```

Размеры шрифта и строки заданы стилем:

```
body {
  font: 14px/1.5 Georgia, serif;
}
```

Сделайте, чтобы между элементами был вертикальный отступ.

- Размер отступа: ровно 1 строка.
- Нужно добавить только одно правило CSS с одним псевдоселектором, можно использовать CSS3.
- Не должно быть лишних отступов сверху и снизу списка.

Результат:

Текст вверху без отступа от списка.
- Маша
- Паша
- Даша
- Женя
- Саша
- Гоша

Текст внизу без отступа от списка.

Открыть песочницу для задачи.

К решению

Отступ между парами, размером со строку

важность: 4

Есть список `UL/LI`.

```
Текст вверху без отступа от списка.
<ul>
  <li>Маша</li>
  <li>Паша</li>
  <li>Даша</li>
  <li>Женя</li>
  <li>Саша</li>
  <li>Гоша</li>
</ul>
Текст внизу без отступа от списка.
```

Размеры шрифта и строки заданы стилем:

```
body {
  font: 14px/1.5 Georgia, serif;
}
```

Сделайте, чтобы между каждой парой элементов был вертикальный отступ.

- Размер отступа: ровно 1 строка.
- Нужно добавить только одно правило CSS, можно использовать CSS3.
- Не должно быть лишних отступов сверху и снизу списка.

Результат:

Текст вверху без отступа от списка.
- Маша
- Паша
- Даша
- Женя
- Саша
- Гоша

Текст внизу без отступа от списка.

Открыть песочницу для задачи.

К решению

CSS-спрайты

CSS-спрайт – способ объединить много изображений в одно, чтобы:

1. Сократить количество обращений к серверу.
2. Загрузить несколько изображений сразу, включая те, которые понадобятся в будущем.
3. Если у изображений сходная палитра, то объединённое изображение будет меньше по размеру, чем совокупность исходных картинок.

Рассмотрим, как это работает, на примере дерева:

```
<ul>
  <li class="open">
    <div class="icon"></div>
    <div class="text">Раздел 1
      <br>В две строки</div>
    <ul>
      <li class="closed">
        <div class="icon"></div>
        <div class="text">Раздел 1.1 в одну строку</div>
      </li>
      <li class="leaf">
        <div class="icon"></div>
        <div class="text">Страница 1.2
          <br> в две строки</div>
      </li>
    </ul>
  </li>
  <li class="closed">
    <div class="icon"></div>
    <div class="text">Раздел 2
      <br>В две строки</div>
  </li>
</ul>
```

Раздел 1
В две строки

Раздел 1.1 в одну строку

Страница 1.2
в две строки

Раздел 2
В две строки

Сейчас «плюс», «минус» и «статья» – три отдельных изображения. Объединим их в спрайт.

Шаг 1. Использовать background

Первый шаг к объединению изображений в «спрайт» – показывать их через `background`., а не через тег `IMG`.

В данном случае он уже сделан. Стиль для дерева:

```
.icon {
  width: 16px;
  height: 16px;
  float: left;
}

.open .icon {
  cursor: pointer;
  background: url(minus.gif);
}

.closed .icon {
  cursor: pointer;
  background: url(plus.gif);
}

.leaf .icon {
  cursor: text;
  background: url(article.gif);
}
```

Шаг 2. Объединить изображения

Составим из нескольких изображений одно `icons.gif`, расположив их, например, по вертикали.

Из , и получится одна картинка: 

Шаг 3. Показать часть спрайта в «окошке»

А теперь самое забавное. Размер `DIV'a` для иконки – жёстко фиксирован:

```
/*+ no-beautify */
.icon {
  width: 16px;
  height: 16px;
  float: left;
}
```

Это значит, что если поставить `background'ом` объединённую картинку, то вся она не поместится, будет видна только верхняя часть:

Пример раздела

Если бы высота иконки была больше, например, `16x48`, как в примере ниже, то было бы видно и остальное:

Пример раздела

...Но так как там всего `16px`, то помещается только одно изображение.

Шаг 4. Сдвинуть спрайт

Сдвиг фона `background-position` позволяет выбирать, какую именно часть спрайта видно.

В спрайте `icons.gif` изображения объединены так, что сдвиг на `16px` покажет следующую иконку:

```
/*+ no-beautify */
.icon {
  width: 16px;
  height: 16px;
  float: left;
  background: url(icons.gif) no-repeat;
}

.open .icon {
  background-position: 0 -16px; /* вверх на 16px */
  cursor: pointer;
}

.closed .icon {
  background-position: 0 0px; /* по умолчанию */
  cursor: pointer;
}

.leaf .icon {
  background-position: 0 -32px; /* вверх на 32px */
  cursor: text;
}
```

Результат:

Раздел 1
В две строки
Раздел 1.1 в одну строку
Страница 1.2
в две строки
Раздел 2
В две строки

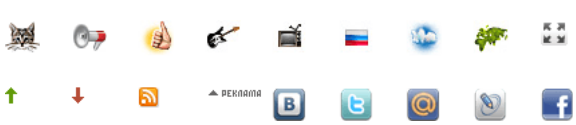
- В спрайт могут объединяться изображения разных размеров, т.е. сдвиг может быть любым.
- Сдвигать можно и по горизонтали и по вертикали.

Отступы

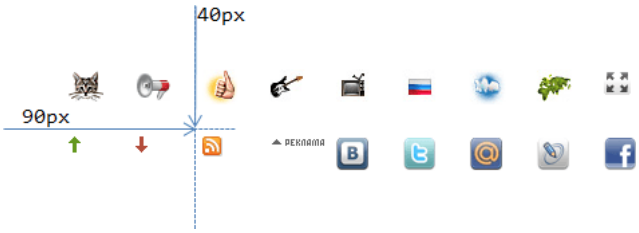
Обычно отступы делаются `margin/padding`, но иногда их бывает удобно предусмотреть в спрайте.

Тогда если элемент немного больше, чем размер изображения, то в «окошке» не появится лишнего.

Пример спрайта с отступами:



Иконка RSS находится в нём на координатах `(90px, 40px)`:



Это значит, что чтобы показать эту иконку, нужно сместить фон:

```
background-position: -90px -40px;
```

При этом в левом-верхнем углу фона как раз и будет эта иконка:

Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость

Элемент, в котором находится иконка (в рамке), больше по размеру, чем картинка.

Его стиль:

```
/*+ no-beautify */
.rss {
  width: 35px; /* ширина/высота больше чем размер иконки */
  height: 35px;
  border: 1px solid black;
  float: left;
  background-image: url(sprite.png);
  background-position: -90px -40px;
}
```

Если бы в спрайте не было отступов, то в такое большое «окошко» наверняка влезли бы другие иконки.

Итого

Когда использовать для изображений `IMG`, а когда – `CSS background`?

Решение лучше всего принимать, исходя из принципов семантической вёрстки.

Задайте вопрос – что здесь делает изображение? Является ли оно самостоятельным элементом страницы (фотография, аватар посетителя), или же оформляет что-либо (иконка узла дерева)?

Элемент `IMG` следует использовать в первом случае, а для оформления у нас есть CSS.

Спрайты позволяют:

1. Сократить количество обращений к серверу.
2. Загрузить несколько изображений сразу, включая те, которые понадобятся в будущем.
3. Если у изображений сходная палитра, то объединённое изображение будет меньше по размеру, чем совокупность исходных картинок.

Если фоновое изображение нужно повторять по горизонтали или вертикали, то спрайты тоже подойдут – изображения в них нужно располагать в этом случае так, чтобы при повторении не были видны соседи, т.е., соответственно, вертикально или горизонтально, но не «решёткой».

Далее мы встретимся со спрайтами при создании интерфейсов, чтобы кнопка при наведении меняла своё изображение. Один спрайт будет содержать все состояния кнопки, а переключение внешнего вида – осуществляться при помощи сдвига `background-position`.

Для автоматизированной сборки спрайтов используются специальные инструменты, например SmartSprites.

Правила форматирования CSS

Для того, чтобы CSS легко читался, полезно соблюдать пять правил форматирования.

Каждое свойство – на отдельной строке

Так – неверно:

```
/*+ no-beautify */
#snapshot-box h2 { padding: 0 0 6px 0; font-weight: bold; position: absolute; left: 0; top: 0; }
```

Так – правильно:

```
/*+ no-beautify */
#snapshot-box h2 {
  position: absolute;
  left: 0;
  top: 0;
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Цель – лучшая читаемость, проще найти и поправить свойство.

Каждый селектор – на отдельной строке

Неправильно:

```
/*+ no-beautify */
#snapshot-box h2, #profile-box h2, #order-box h2 {
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Правильно:

```
/*+ no-beautify */
#snapshot-box h2,
#profile-box h2,
#order-box h2 {
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Цель – лучшая читаемость, проще найти селектор.

Свойства, сильнее влияющие на документ, идут первыми

Рекомендуется располагать свойства в следующем порядке:

1. Сначала положение элемента относительно других: `position`, `left/right/top/bottom`, `float`, `clear`, `z-index`.
2. Затем размеры и отступы: `width`, `height`, `margin`, `padding` ...
3. Рамка `border`, она частично относится к размерам.
4. Общее оформление содержимого: `list-style-type`, `overflow` ...
5. Цветовое и стилевое оформление: `background`, `color`, `font` ...

Общая логика сортировки: «от общего – к локальному и менее важному».

При таком порядке свойства, определяющие структуру документа, будут видны наиболее отчётливо, в начале, а самые незначительно влияющие (например цвет) – в конце.

Например:

```
/*+ no-beautify */
#snapshot-box h2 {
  position: absolute; /* позиционирование */
  left: 0;
  top: 0;

  padding: 0 0 6px 0; /* размеры и отступы */

  color: red;         /* стилевое оформление */
  font-weight: bold;
}
```

Свойство без префикса пишется последним.

Например:

```
-webkit-box-shadow:0 0 100px 20px #000;
box-shadow:0 0 100px 20px #000;
```

Это нужно, чтобы стандартная (окончательная) реализация всегда была важнее, чем временные браузерные.

Организация CSS-файлов проекта

Стили можно разделить на две основные группы:

1. Блоки-компоненты имеют свой CSS. Например, CSS для диалогового окна, CSS для профиля посетителя, CSS для меню.

   Такой CSS идёт «в комплекте» с модулем, его удобно выделять в отдельные файлы и подключать через `@import`.

   Конечно, при разработке будет много CSS-файлов, но при выкладке проекта система сборки и сжатия CSS заменит директивы `@import` на их содержимое, объединяя все CSS в один файл.

2. Страничный и интегрирующий CSS.

   Этот CSS описывает общий вид страницы, расположение компонент и их дополнительную стилизацию, зависящую от места на странице и т.п.

```
/*+ no-beautify */
.tab .profile { /* профиль внутри вкладки */
  float: left;
  width: 300px;
  height: 200px;
}
```

   Важные страничные блоки можно выделять особыми комментариями:

```
/** ============================
 *  Профиль посетителя
 *  ============================
*/

.profile {
  border: 1px solid gray;
}

.profile h2 {
  color: blue;
  font-size: 1.8em;
}
```

CSS-препроцессоры, такие как SASS, LESS, Stylus, Autoprefixer делают написание CSS сильно удобнее...

Выберите один из них и используйте. Единственно, они добавляют дополнительную предобработку CSS, которую нужно учесть, и желательно, на сервере.

Сундучок с инструментами

Полезные расширения Firefox и Chrome

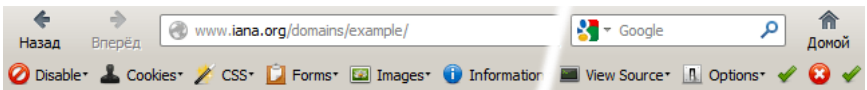
Здесь мы посмотрим ряд полезных расширений, которые, надеюсь, смогут вам пригодиться.

Если какое-то из расширений вас заинтересует – его можно поставить. В Firefox расширения устанавливаются из меню «Инструменты → Дополнения», в Chrome – через меню «Настройки → Расширения → (внизу) Ещё расширения» или напрямую из «магазина» `https://chrome.google.com/webstore/category/home`.

Для расширений других браузеров такого единого способа нет.

Web Developer (FF, Ch)

Это расширение добавляет наверх браузера панель с разнообразными инструментами:



Большинство возможностей, скорее, полезны для верстальщиков, но и для разработчиков кое-что есть.

Например:

- Disable Cache – полностью отключает кэш браузера
- Disable Cookies – браузер перестанет посылать и принимать Cookie. Впрочем, Firebug это тоже умеет (вкладка Cookie)
- Disable JavaScript – нужно редко, но иногда полезно
- Resize на нужное разрешение – у разработчиков мониторы большие, а у пользователей разные. Можно добавить своё разрешение.

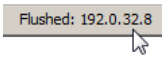
Иконок там много, поэтому наверняка вы чем-то не будете пользоваться. Лишние иконки можно убрать. В Firefox(Win) это делается так: Правый клик на панель → Настроить → Перетащить лишние иконки с панели.

DNS Flusher (FF) / DNS Flusher for Chrome

Это расширение позволяет сбросить кэш DNS одним кликом.

Оно нужно в тех случаях, когда вы меняете адреса в файле `hosts` и хотите, чтобы изменения вступили в действие тут же.

При установке в статусной строке, снизу, появляется кнопочка с IP-адресом. По клику на ней кэш DNS сбрасывается.



Более удобный Firebug (FF)

Здесь собраны расширения, улучшающие работу отладчика Firebug для Firefox.

Fire Rainbow

Подсветка кода для Firebug

FireQuery

Для разработки под jQuery – выводит в разных местах вспомогательную информацию.

Firebug Autocompleter

Включает автодополнение для консоли, когда она в многострочном режиме.

CSS-X-Fire

Позволяет редактировать CSS в Firebug и сохранять изменения, интегрировано с редакторами от JetBrains (IntelliJ IDEA и другие).

JsonView (FF,Ch)

Даёт возможность открыть JSON-документ прямо в браузере.

Обычно браузер не понимает `Content-Type: application/json` и пытается сохранить JSON-файлы.

Но если в нём стоит это расширение, то он покажет файл в удобном виде, с возможностью навигации:


```
{
    hey: "guy",
    anumber: 243,
  - anobject: {
        whoa: "nuts",
      - anarray: [
            1,
            2,
            "thr<h1>ee"
        ],
        more: "stuff"
    },
    awesome: true,
    bogus: false,
    meaning: null,
    japanese: "明日がある。",
    link: http://jsonview.com,
    notLink: "http://jsonview.com is great"
}
```


Xml Tree (Ch)

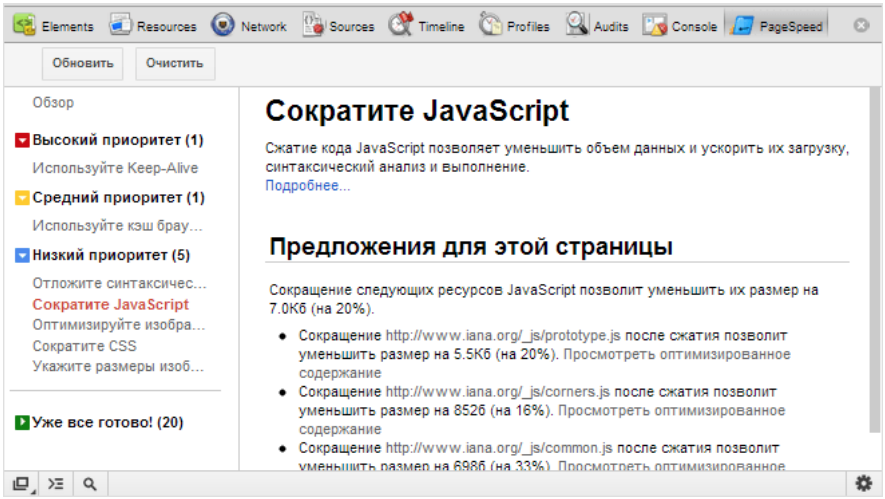
Расширение для просмотра XML для Chrome.

Остальные браузеры умеют это делать «из коробки».

YSlow, PageSpeed Insights (FF, Ch)

Эти два расширения позволяют по-быстрому оценить скорость загрузки страницы, проанализировать массу возможных причин тормозов и получить советы по улучшению производительности.

Они очень похожи, но не идентичны, так что можно поставить оба.



AdBlock (FF, Ch)

Расширение для отключения назойливой рекламы и баннеров.

Режет не всё, но многое. Да, оно не для разработки, но настолько полезное, что я не смог удержаться и опубликовал его здесь.

Кстати, насчёт разработки… Бывает так, что AdBlock прячет рекламу на сайтах, которые мы разрабатываем. К хорошему быстро привыкаешь, и если юзер будет слать вам ошибки на страницах (а у вас их нет) – проверьте! Может быть, стоит отключить AdBlock для конкретной страницы и посмотреть без него? Исключения можно поставить в настройках расширения.

DownloadHelper (FF)

Расширение для скачивания видео из Youtube, Blip TV и других хостингов.

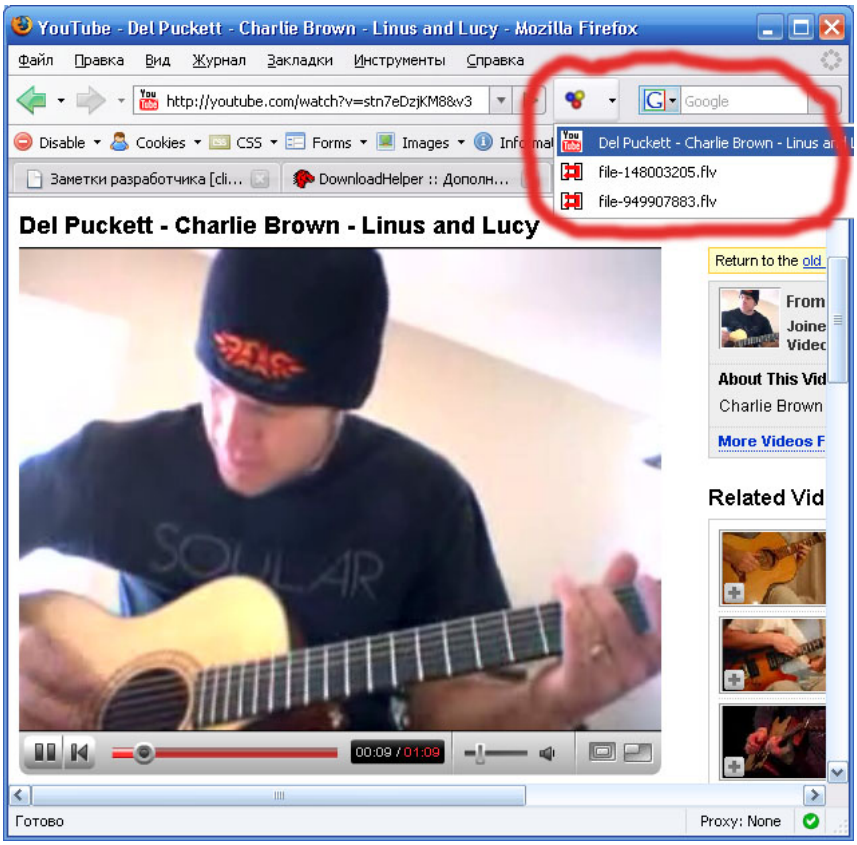
С одной стороны, это расширение тоже напрямую не связано с разработкой…

…Но с другой – бывает ли, что работая на компьютере вы получаете ссылку на интересное видео с конференции? Или заходите на сайт и видите там набор прекрасных выступлений, которые хочется скачать и посмотреть в более удобное время с планшетника. Скажем, во время поездки, когда нет интернет.

Если бывает, то это расширение – для вас. Скачиваем отличные видео и смотрим, когда захотим.

Способ использования:

1. Зайти на страницу.
2. Начать смотреть видео.
3. Увидеть, как иконка расширения «ожила», кликнуть на нее и выбрать видео в нужном разрешении (если есть несколько).
4. Файл будет скачан в папку `dwhelper` (по умолчанию), место можно поменять в настройках.



P.S.

В этот список расширений я включил самое любимое и полезное, что связано именно с расширениями браузера.

Есть дополнительные инструменты, которые становятся в систему, они идут в отдельном разделе.

У вас есть, что добавить? Расширения, которые вам очень помогли? Укажите их в комментариях.

Скриптуемый отладочный прокси Fiddler

Fiddler – прокси, который работает с трафиком между Вашим компьютером и удаленным сервером, и позволяет инспектировать и менять его.

Fiddler можно расширять с помощью скриптов на языке JScript.NET (писать их очень просто), менять меню программы, и вообще – замечательный инструмент.

Использовать его можно с любым браузером.

Эта статья описывает Fiddler 2.4. В вашей версии Fiddler какие-то возможности могут измениться, какие-то образоваться.

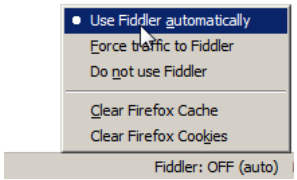
Режимы подключения

У Fiddler есть 2 способа подключения.

1. Первый – это просто запустить его. При этом он автоматически будет работать для программ, использующих WinINET.

   Это, например, Internet Explorer, Chrome, приложения MS Office.

   Firefox тоже автоматически подхватит Fiddler, за счёт того что при установке Fiddler в него ставится соответствующее расширение:



2. Второй – это явно поставить Fiddler как прокси для браузера, по адресу `127.0.0.1:8888` . Например, так можно сделать для Opera, если уж не хочется перезапускать.

Fiddler не под Windows

Если вы работаете не под Windows, то Fiddler можно поставить в виртуальную машину.

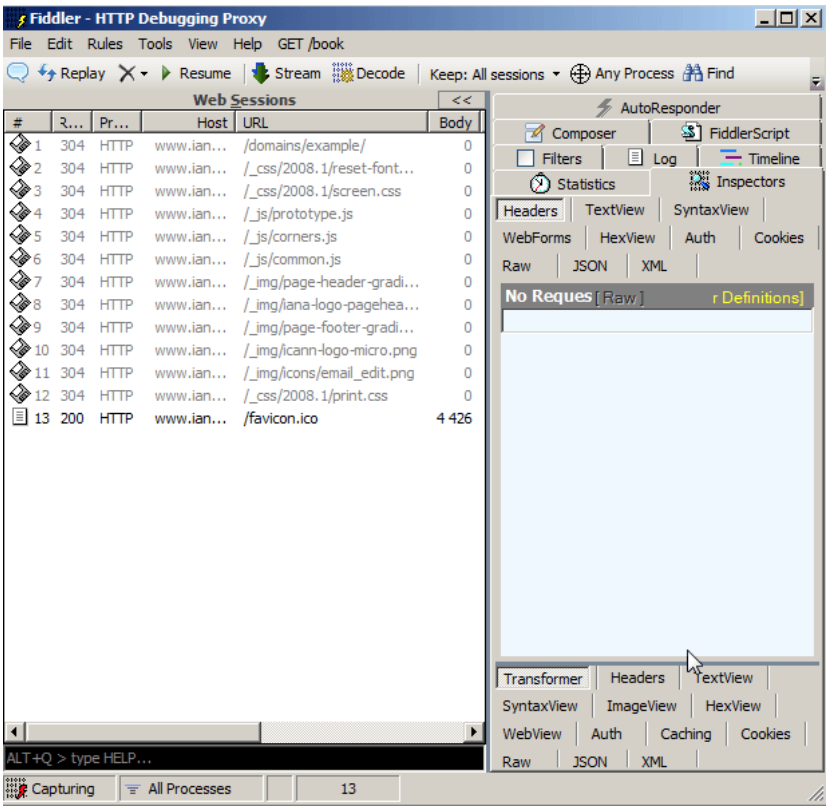
Чтобы сделать возможной подключение внешних браузеров, нужно включить настройках Fiddler: Tools → Fiddler Options → Connections(вкладка) галочку «Allow remote clients to connect». После этого Fiddler станет доступен как прокси на интерфейсе `0.0.0.0` , так что можно будет браузеру из внешней ОС

указать в качестве прокси виртуальную машину. И пользоваться Fiddler.

Если вы так захотите поступить, то вдобавок возьмите удобный переключатель прокси, например Elite Proxy Switcher под Firefox или Proxy Pick для IE, чтобы переключение на прокси осуществлялось в один клик.

Операции над запросами

При заходе в Fiddler, открывается окно запросов слева и рабочие вкладки справа.



Как видно, опций много, на изображении они еле-еле помещаются. И, поверьте, возможностей – ещё больше.

Возможности

- В окне запросов слева можно просматривать и выбирать запросы, смотреть их заголовки, сохранять их на диск все вместе или по отдельности.
- AutoResponder – позволяет подставить свой файл вместо сервера.

  Например, приятель попросил поправить скрипт `vasya.js` на сайте, но доступа к серверу не дал.

  С Fiddler задача решается просто – сохраняете скрипт на диске, в AutoResponder указываете, что `vasya.js` нужно брать с диска, а не с сайта – и исправляете, что нужно, перезагружаете страницу, проверяете – всё с комфортом.
- Composer – позволяет составить запрос на сервер вручную.

  Например, вы хотите сделать такой же AJAX-запрос, как только что делали. Для этого можно просто выбрать его слева и нажать кнопку Replay (слева-сверху).

  А если хочется поменять? Нет ничего проще – выбираем справа Composer и перетаскиваем запрос слева в него. После этого исправляем, что хотим и Execute.
- Filters – позволяет назначить действия в зависимости от вида запроса. Опции станут понятны после перехода на вкладку.
- FiddlerScript – основа мощи Fiddler: скрипт, задающий функционал. Его редактированием можно добавить или удалить пункты меню, колонки в списке запросов, и, вообще, поменять почти всё.

  Язык программирования JScript.NET, который здесь используется, может взаимодействовать с Windows в полном объеме, включая коммуникацию с базой данных, Word, Excel.

Правила

Слева-сверху в меню находится пункт Rules (правила). В нём изначально находятся некоторые возможности FiddlerScript, которые поставляются «из коробки».

Хотите посмотреть, как ваш сайт будет грузиться «на GPRS»? Выбирайте Rules → Performance → Simulate Modem speeds.

Для добавления новых правил можно их задать через пункт «Customize Rules» (на JScript.NET, разумеется). В открывающемся скрипте есть пункты меню и их реализация.

При наступлении любого события из обширного списка, Fiddler вызывает соответствующий обработчик из правил. Например, onBeforeRequest, onShutdown. Стандартные правила отлично прокомментированы, и писать новые весьма просто.

FiddlerScript позволяет манипулировать заголовками, запросом, менять ширину канала, управлять выводом запроса в Fiddler и так далее и т.п.

Брейкпойнт на запросе

В меню Rules → Automatic Breakpoints можно включить автоматическое прерывание Fiddler при обработке запроса.

После этого, если сделать запрос в браузере, подключенном к Fiddler, то его выполнение зависнет, а в левом окошке Fiddler этот запрос будет отмечен специальным значком.

Если выбрать такой подвисший запрос мышкой, то во вкладке SessionInspector им можно управлять: менять сам запрос и ответ сервера (после Break on Response, когда сервер уже ответил).

Задавать прерывание на запросах определенного вида также можно через Filters.

Отладка HTTPS

Fiddler является прокси, а HTTPS шифруется от браузера до сервера-получателя, поэтому по умолчанию Fiddler не имеет доступа к содержимому HTTPS-запросов.

Чтобы его получить, Fiddler должен сыграть роль хакера-перехватчика: расшифровывать запросы, и потом отправлять дальше. Это возможно, если инсталлировать специальный сертификат: Tools → Fiddler Options → HTTPS(вкладка) → выбрать все галочки.

После окончания отладки этот сертификат можно убрать.

Скачать

Fiddler можно бесплатно скачать с сайта разработчика. Там же доступна документация и видео.

К фиддлеру прилагается галерея расширений http://www.fiddlertool.com/fiddler2/extensions.asp.

Примеры скриптов для Fiddler, которые дают общее представление о том, на что он может быть способен: http://www.fiddlertool.com/fiddler/dev/scriptsamples.asp.

IE HTTP Analyzer

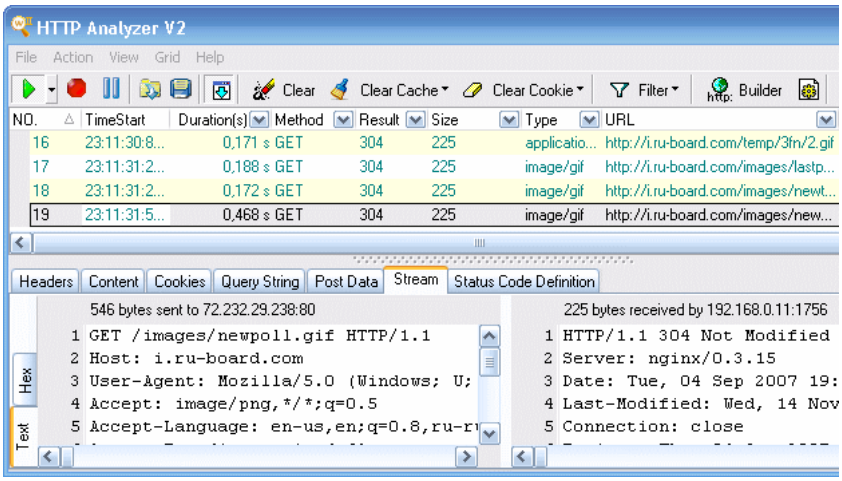
Под Windows есть как минимум два весьма основных инструмента для отладки на уровне HTTP.

Один из них – Fiddler. Другой – IE HTTP Analyzer, который представляет собой «другой угол зрения» на задачу.

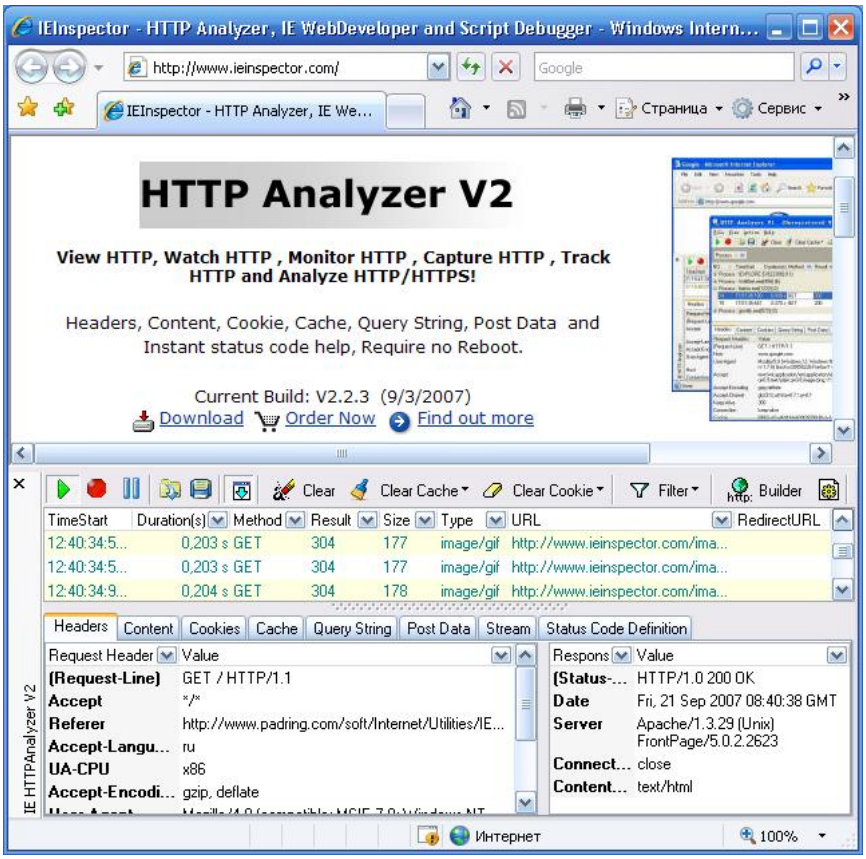
Возможности

IE HTTP Analyzer – платный, и имеет две ипостаси.

Первая – это обычная программа, запущенная в отдельном окне. При нажатии на кнопку «Play» – начинает перехват HTTP для всех браузеров, включая Firefox.



Вторая – наподобие Firebug, раскрывающаяся полоса внизу браузера Internet Explorer. Это – плагин, вызывается из панели или меню браузера.



Существуют другие программы похожего вида, но, перебрав несколько, я остановился на этой, как самой удобной.

Основные возможности сводятся к удобной отладке HTTP-запросов, да и кнопочки на скриншотах выше достаточно очевидны.

Просмотр и создание HTTP-запросов.

Этим сейчас никого не удивишь, если бы не одно БОЛЬШОЕ НО.

Программа дает возможность удобно просматривать потоковые запросы.

Т.е, например, отлаживаете Вы чат. А он держит непрерывный iframe, через который сервер передает данные. Программы типа Fiddler будут честно ждать конца запроса, чтобы показать Вам его, а IE HTTP Analyzer не менее честно будет сбрасывать поток пред Ваши светлы очи.

Запросы могут поступать все новые и новые, но в панели инспектора есть кнопка  , которая управляет автопереходом к следующему запросу. При анализе потока она должна быть выключена.

Эта «потоковость» IE Http Analyzer- один из важных его плюсов.

Конструирование фильтров

В отличие от правил на языке JScript.NET, которые используются для фильтрации в Fiddler, здесь фильтры на интересующие Вас запросы можно конструировать из стандартных компонент, объединяя их условиями ИЛИ/И.

Дополнительные возможности

Очистка куков и кеша браузера. Проверено, работает.

Программа платная.

Живет на http://www.ieinspector.com/httpanalyzer/ .

Регулярные выражения

Регулярные выражения – мощный способ поиска и замены для строк.

Паттерны и флаги

Регулярные выражения – мощное средство поиска и замены в строке.

В JavaScript регулярные выражения реализованы отдельным объектом `RegExp` и интегрированы в методы строк.

Регэкспы

Регулярное выражение (оно же «регэксп», «регулярка» или просто «рег»), состоит из *паттерна* (он же «шаблон») и необязательных *флагов*.

Синтаксис создания регулярного выражения:

```
var regexp = new RegExp("шаблон", "флаги");
```

Как правило, используют более короткую запись (шаблон внутри слешей `"/"`):

```
var regexp = /шаблон/; // без флагов
var regexp = /шаблон/gmi; // с флагами gmi (изучим их дальше)
```

Слеши `"/"` говорят JavaScript о том, что это регулярное выражение. Они играют здесь ту же роль, что и кавычки для обозначения строк.

Использование

Основа регулярного выражения – паттерн. Это строка, которую можно расширить специальными символами, которые делают поиск намного мощнее.

В простейшем случае, если флагов и специальных символов нет, поиск по паттерну – то же самое, что и обычный поиск подстроки:

```
var str = "Я люблю JavaScript!"; // будем искать в этой строке

var regexp = /лю/;
alert( str.search(regexp) ); // 2
```

Сравните с обычным поиском:

```
var str = "Я люблю JavaScript!";

var substr = "лю";
alert( str.indexOf(substr) ); // 2
```

Как видим, то же самое, разве что для регэкспа использован метод search – он как раз работает с регулярными выражениями, а для подстроки – indexOf.

Но это соответствие лишь кажущееся. Очень скоро мы усложним регулярные выражения, и тогда увидим, что они гораздо мощнее.

Цветовые обозначения

Здесь и далее в тексте используется следующая цветовая схема:

- регэксп (регулярное выражение) – `красный`
- строка – `синий`
- результат – `зеленый`

Флаги

Регулярные выражения могут иметь флаги, которые влияют на поиск.

В JavaScript их всего три:

`i`

Если этот флаг есть, то регэксп ищет независимо от регистра, то есть не различает между `A` и `a`.

`g`

Если этот флаг есть, то регэксп ищет все совпадения, иначе – только первое.

`m`

Многострочный режим.

Самый простой для понимания из этих флагов – безусловно, `i`.

Пример его использования:

```
var str = "Я люблю JavaScript!"; // будем искать в этой строке

alert( str.search( /ЛЮ/ ) ); // -1
alert( str.search( /ЛЮ/i ) ); // 2
```

1. С регом `/ЛЮ/` вызов вернул `-1`, что означает «не найдено» (как и в `indexOf`),
2. С регом `/ЛЮ/i` вызов нашёл совпадение на позиции 2, так как стоит флаг `i`, а значит «лю» тоже подходит.

Другие флаги мы рассмотрим в последующих главах.

Итого

- Регулярное выражение состоит из шаблона и необязательных флагов `g`, `i` и `m`.
- Поиск по регулярному выражению без флагов и спец. символов, которые мы изучим далее – это то же самое, что и обычный поиск подстроки в строке. Но флаги и спец. символы, как мы увидим далее, могут сделать его гораздо мощнее.
- Метод строки `str.search(regexp)` возвращает индекс, на котором найдено совпадение.

Методы RegExp и String

Регулярные выражения в JavaScript являются объектами класса RegExp.

Кроме того, методы для поиска по регулярным выражениям встроены прямо в обычные строки `String`.

К сожалению, общая структура встроенных методов слегка запутана, поэтому мы сначала рассмотрим их по отдельности, а затем – рецепты по решению стандартных задач с ними.

str.search(reg)

Этот метод мы уже видели.

Он возвращает позицию первого совпадения или `-1`, если ничего не найдено.

```
var str = "Люблю регэкспы я, но странною любовью";

alert( str.search( /лю/i ) ); // 0
```

Ограничение метода **`search`** – он всегда ищет только первое совпадение.

Нельзя заставить `search` искать дальше первого совпадения, такой синтаксис попросту не предусмотрен. Но есть другие методы, которые это умеют.

str.match(reg) без флага g

Метод `str.match` работает по-разному, в зависимости от наличия или отсутствия флага `g`, поэтому сначала мы разберём вариант, когда его нет.

В этом случае `str.match(reg)` находит только одно, первое совпадение.

Результат вызова – это массив, состоящий из этого совпадения, с дополнительными свойствами `index` – позиция, на которой оно обнаружено и `input` – строка, в которой был поиск.

Например:

```
var str = "ОЙ-Ой-ой";

var result = str.match( /ой/i );

alert( result[0] ); // ОЙ  (совпадение)
alert( result.index ); // 0 (позиция)
alert( result.input ); // ОЙ-Ой-ой (вся поисковая строка)
```

У этого массива не всегда только один элемент.

Если часть шаблона обозначена скобками, то она станет отдельным элементом массива.

Например:

```
var str = "javascript - это такой язык";

var result = str.match( /JAVA(SCRIPT)/i );

alert( result[0] ); // javascript (всё совпадение полностью)
alert( result[1] ); // script (часть совпадения, соответствующая скобкам)
alert( result.index ); // 0
alert( result.input ); // javascript - это такой язык
```

Благодаря флагу `i` поиск не обращает внимание на регистр буквы, поэтому находит `javascript`. При этом часть строки, соответствующая `SCRIPT`, выделена в отдельный элемент массива.

Позже мы ещё вернёмся к скобочным выражениям, они особенно удобны для поиска с заменой.

str.match(reg) с флагом g

При наличии флага `g`, вызов `match` возвращает обычный массив из всех совпадений.

Никаких дополнительных свойств у массива в этом случае нет, скобки дополнительных элементов не порождают.

Например:

```
var str = "ОЙ-Ой-ой";

var result = str.match( /ой/ig );
```

```
alert( result ); // ОЙ, Ой, ой
```

Пример со скобками:

```
var str = "javascript - это такой язык";

var result = str.match( /JAVA(SCRIPT)/gi );

alert( result[0] ); // javascript
alert( result.length ); // 1
alert( result.index ); // undefined
```

Из последнего примера видно, что элемент в массиве ровно один, и свойства `index` также нет. Такова особенность глобального поиска при помощи `match` – он просто возвращает все совпадения.

Для расширенного глобального поиска, который позволит получить все позиции и, при желании, скобки, нужно использовать метод RegExp#exec, которые будет рассмотрен далее.

> ⚠ В случае, если совпадений не было, **`match`** возвращает **`null`**
>
> Обратите внимание, это важно – если `match` не нашёл совпадений, он возвращает не пустой массив, а именно `null`.
>
> Это важно иметь в виду, чтобы не попасть в такую ловушку:
>
> ```
> var str = "Ой-йой-йой";
>
> // результат match не всегда массив!
> alert(str.match(/лю/gi).length) // ошибка! нет свойства length у null
> ```

str.split(reg|substr, limit)

Разбивает строку в массив по разделителю – регулярному выражению `regexp` или подстроке `substr`.

Обычно мы используем метод `split` со строками, вот так:

```
alert('12-34-56'.split('-')) // [12, 34, 56]
```

Можно передать в него и регулярное выражение, тогда он разобьёт строку по всем совпадениям.

Тот же пример с регэкспом:

```
alert('12-34-56'.split(/-/)) // [12, 34, 56]
```

str.replace(reg, str|func)

Швейцарский нож для работы со строками, поиска и замены любого уровня сложности.

Его простейшее применение – поиск и замена подстроки в строке, вот так:

```
// заменить дефис на двоеточие
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

При вызове со строкой замены **`replace`** всегда заменяет только первое совпадение.

Чтобы заменить *все* совпадения, нужно использовать для поиска не строку `"-"`, а регулярное выражение `/-/g`, причём обязательно с флагом `g`:

```
// заменить дефис на двоеточие
alert( '12-34-56'.replace( /-/g, ":" ) )  // 12:34:56
```

В строке для замены можно использовать специальные символы:

Спецсимволы	Действие в строке замены
`$$`	Вставляет `"$"`.
`$&`	Вставляет всё найденное совпадение.
`$``	Вставляет часть строки до совпадения.
`$'`	Вставляет часть строки после совпадения.
`$*n*`	где `n` -- цифра или двузначное число, обозначает `n`-ю по счёту скобку, если считать слева-направо.

Пример использования скобок и `$1`, `$2`:

```
var str = "Василий Пупкин";

alert(str.replace(/(Василий) (Пупкин)/, '$2, $1')) // Пупкин, Василий
```

Ещё пример, с использованием `$&` :

```
var str = "Василий Пупкин";

alert(str.replace(/Василий Пупкин/, 'Великий $&!')) // Великий Василий Пупкин!
```

Для ситуаций, которые требуют максимально «умной» замены, в качестве второго аргумента предусмотрена функция.

Она будет вызвана для каждого совпадения, и её результат будет вставлен как замена.

Например:

```
var i = 0;

// заменить каждое вхождение "ой" на результат вызова функции
alert("ОЙ-Ой-ой".replace(/ой/gi, function() {
  return ++i;
})); // 1-2-3
```

В примере выше функция просто возвращала числа по очереди, но обычно она основывается на поисковых данных.

Эта функция получает следующие аргументы:

1. `str` – найденное совпадение,
2. `p1, p2, ..., pn` – содержимое скобок (если есть),
3. `offset` – позиция, на которой найдено совпадение,
4. `s` – исходная строка.

Если скобок в регулярном выражении нет, то у функции всегда будет ровно 3 аргумента: `replacer(str, offset, s)` .

Используем это, чтобы вывести полную информацию о совпадениях:

```
// вывести и заменить все совпадения
function replacer(str, offset, s) {
  alert( "Найдено: " + str + " на позиции: " + offset + " в строке: " + s );
  return str.toLowerCase();
}

var result = "ОЙ-Ой-ой".replace(/ой/gi, replacer);
alert( 'Результат: ' + result ); // Результат: ой-ой-ой
```

С двумя скобочными выражениями – аргументов уже 5:

```
function replacer(str, name, surname, offset, s) {
  return surname + ", " + name;
}

var str = "Василий Пупкин";

alert(str.replace(/(Василий) (Пупкин)/, replacer)) // Пупкин, Василий
```

Функция – это самый мощный инструмент для замены, какой только может быть. Она владеет всей информацией о совпадении и имеет доступ к замыканию, поэтому может всё.

regexp.test(str)

Теперь переходим к методам класса `RegExp` .

Метод `test` проверяет, есть ли хоть одно совпадение в строке `str` . Возвращает `true/false` .

Работает, по сути, так же, как и проверка `str.search(reg) != -1` , например:

```
var str = "Люблю регэкспы я, но странною любовью";

// эти две проверки идентичны
alert( /лю/i.test(str) ) // true
alert( str.search(/лю/i) != -1 ) // true
```

Пример с отрицательным результатом:

```
var str = "Ой, цветёт калина...";

alert( /javascript/i.test(str) ) // false
alert( str.search(/javascript/i) != -1 ) // false
```

regexp.exec(str)

Для поиска мы уже видели методы:

- `search` – ищет индекс
- `match` – если регэксп без флага `g` – ищет совпадение с подрезультатами в скобках
- `match` – если регэксп с флагом `g` – ищет все совпадения, но без скобочных групп.

Метод `regexp.exec` дополняет их. Он позволяет искать и все совпадения и скобочные группы в них.

Он ведёт себя по-разному, в зависимости от того, есть ли у регэкспа флаг `g`.

- Если флага `g` нет, то `regexp.exec(str)` ищет и возвращает первое совпадение, является полным аналогом вызова `str.match(reg)`.
- Если флаг `g` есть, то вызов `regexp.exec` возвращает первое совпадение и *запоминает* его позицию в свойстве `regexp.lastIndex`. Последующий поиск он начнёт уже с этой позиции. Если совпадений не найдено, то сбрасывает `regexp.lastIndex` в ноль.

Это используют для поиска всех совпадений в цикле:

```
var str = 'Многое по JavaScript можно найти на сайте http://javascript.ru';

var regexp = /javascript/ig;
var result;

alert( "Начальное значение lastIndex: " + regexp.lastIndex );

while (result = regexp.exec(str)) {
  alert( 'Найдено: ' + result[0] + ' на позиции:' + result.index );
  alert( 'Свойство lastIndex: ' + regexp.lastIndex );
}

alert( 'Конечное значение lastIndex: ' + regexp.lastIndex );
```

Здесь цикл продолжается до тех пор, пока `regexp.exec` не вернёт `null`, что означает «совпадений больше нет».

Найденные результаты последовательно помещаются в `result`, причём находятся там в том же формате, что и `match` – с учётом скобок, со свойствами `result.index` и `result.input`.

Поиск с нужной позиции

Можно заставить `regexp.exec` искать сразу с нужной позиции, если поставить `lastIndex` вручную:

```
var str = 'Многое по JavaScript можно найти на сайте http://javascript.ru';

var regexp = /javascript/ig;
regexp.lastIndex = 40;

alert( regexp.exec(str).index ); // 49, поиск начат с 40-й позиции
```

Итого, рецепты

Методы становятся гораздо понятнее, если разбить их использование по задачам, которые нужны в реальной жизни.

Для поиска только одного совпадения:

- Найти позицию первого совпадения – `str.search(reg)`.
- Найти само совпадение – `str.match(reg)`.
- Проверить, есть ли хоть одно совпадение – `regexp.test(str)` или `str.search(reg) != -1`.
- Найти совпадение с нужной позиции – `regexp.exec(str)`, начальную позицию поиска задать в `regexp.lastIndex`.

Для поиска всех совпадений:

- Найти массив совпадений – `str.match(reg)`, с флагом `g`.
- Получить все совпадения, с подробной информацией о каждом – `regexp.exec(str)` с флагом `g`, в цикле.

Для поиска-и-замены: : - Замена на другую строку или результат функции -- `str.replace(reg, str|func)`

Для разбивки строки на части:

- `str.split(str|reg)`

Зная эти методы, мы уже можем использовать регулярные выражения.

Конечно, для этого желательно хорошо понимать их синтаксис и возможности, так что переходим к ним дальше.

Классы и спецсимволы

Рассмотрим практическую задачу – есть телефонный номер `"+7(903)-123-45-67"`, и нам нужно найти в этой строке цифры. А остальные символы нас не интересуют.

Для поиска символов определённого вида в регулярных выражениях предусмотрены «классы символов».

Класс символов – это специальное обозначение, под которое подходит любой символ из определённого набора.

Например, есть класс «любая цифра». Он обозначается `\d`. Это обозначение вставляется в шаблон, и при поиске под него подходит любая цифра.

То есть, регулярное выражение `/\d/` ищет ровно одну цифру:

```
var str = "+7(903)-123-45-67";

var reg = /\d/;

// не глобальный регэксп, поэтому ищет только первую цифру
alert( str.match(reg) ); // 7
```

...Ну а для поиска всех цифр достаточно добавить к регэкспу флаг `g`:

```
var str = "+7(903)-123-45-67";

var reg = /\d/g;

alert( str.match(reg) ); // массив всех совпадений: 7,9,0,3,1,2,3,4,5,6,7
```

Важнейшие классы: \d \s \w

Это был класс для цифр.

Конечно же, есть и другие.

Наиболее часто используются:

`\d` (от английского «digit» – «цифра»)

Цифра, символ от `0` до `9`.

`\s` (от английского «space» – «пробел»)

Пробельный символ, включая табы, переводы строки и т.п.

`\w` (от английского «word» – «слово»)

Символ «слова», а точнее – буква латинского алфавита или цифра или подчёркивание `'_'`. Не-английские буквы не являются `\w`, то есть русская буква не подходит.

Например, `\d\s\w` обозначает цифру, за которой идёт пробельный символ, а затем символ слова.

Регулярное выражение может содержать одновременно и обычные символы и классы.

Например, `CSS\d` найдёт строку `CSS`, с любой цифрой после неё:

```
var str = "Стандарт CSS4 - это здорово";
var reg = /CSS\d/

alert( str.match(reg) ); // CSS4
```

И много классов подряд:

```
alert( "Я люблю HTML5!".match(/\s\w\w\w\w\d/) ); // 'HTML5'
```

Совпадение (каждому классу в регэкспе соответствует один символ результата):

Граница слова \b

Граница слова `\b` – это особый класс.

Он интересен тем, что обозначает не символ, а границу между символами.

Например, `\bJava\b` найдёт слово `Java` в строке `Hello, Java!`, но не в строке `Hello, Javascript!`.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, Javascript!".match(/\bJava\b/) ); // null
```

Граница имеет «нулевую ширину» в том смысле, что обычно символам регулярного выражения соответствуют символы строки, но не в этом случае.

Граница – это проверка.

При поиске движок регулярных выражений идёт по шаблону и одновременно по строке, пытаясь построить соответствие. Когда он видит `\b`, то проверяет, что текущая позиция в строке подходит под одно из условий:

- Начало текста, если первый символ `\w`.
- Конец текста, если последний символ `\w`.
- Внутри текста, если с одной стороны `\w`, а с другой – не `\w`.

Например, в строке `Hello, Java!` под `\b` подходят следующие позиции:

Как правило, `\b` используется, чтобы искать отдельно стоящее слово. Не на русском конечно, хотя подобную проверку, как мы увидим далее, можно легко сделать для любого языка. А вот на английском, как в примере выше или для чисел, которые являются частным случаем `\w` – легко.

Например, регэксп `\b\d\d\b` ищет отдельно двузначные числа. Иными словами, он требует, чтобы до и после `\d\d` был символ, отличный от `\w` (или начало/конец текста).

Обратные классы

Для каждого класса существует «обратный ему», представленный такой же, но заглавной буквой.

«Обратный» – означает, что ему соответствуют все остальные символы, например:

`\D`

Не-цифра, то есть любой символ кроме `\d`, например буква.

`\S`

Не-пробел, то есть любой символ кроме `\s`, например буква.

`\W`

Любой символ, кроме `\w`, то есть не латинница, не подчёркивание, не цифра. В частности, русские буквы принадлежат этому классу.

`\B`

Проверка, обратная `\b`.

В начале этой главы мы видели, как получить из телефона `+7(903)-123-45-67` все цифры.

Первый способ – найти все цифры через `match(/\d/g)`.

Обратные классы помогут реализовать альтернативный – найти все НЕцифры и удалить их из строки:

```
var str = "+7(903)-123-45-67";

alert( str.replace(/\D/g, "") ); // 79031234567
```

Пробелы – обычные символы

Заметим, что в регулярных выражениях пробел – такой же символ, как и другие.

Обычно мы не обращаем внимание на пробелы. Для нашего взгляда строки `1-5` и `1 - 5` почти идентичны.

Однако, если регэксп не учитывает пробелов, то он не сработает.

Попытаемся найти цифры, разделённые дефисом:

```
alert( "1 - 5".match(/\d-\d/) ); // null, нет совпадений!
```

Поправим это, добавив в регэксп пробелы:

```
alert( "1 - 5".match(/\d - \d/) ); // работает, пробелы вокруг дефиса
```

Конечно же, пробелы в регэкспе нужны лишь тогда, когда мы их ищем. Лишние пробелы (как и любые лишние символы) могут навредить:

```
alert( "1-5".match(/\d - \d/) ); // null, так как в строке 1-5 нет пробелов
```

Короче говоря, в регулярном выражении все символы имеют значение. Даже (и тем более) – пробелы.

Точка – любой символ

Особым классом символов является точка `"."`.

В регулярном выражении, точка `"."` обозначает *любой символ*, кроме перевода строки:

```
alert( "Z".match(/./) ); // найдено Z
```

Посередине регулярного выражения:

```
var re = /CS.4/;

alert( "CSS4".match(re) ); // найдено "CSS4"
alert( "CS-4".match(re) ); // найдено "CS-4"
alert( "CS 4".match(re) ); // найдено "CS 4" (пробел тоже символ)
```

Обратим внимание – точка означает именно «произвольный символ».

То есть какой-то символ на этом месте в строке должен быть:

```
alert( "CS4".match(/CS.4/) ); // нет совпадений, так как для точки нет символа
```

Экранирование специальных символов

В регулярных выражениях есть и другие символы, имеющие особый смысл.

Они используются, чтобы расширить возможности поиска.

Вот их полный список: `[ \ ^ $ . | ? * + ( )`.

Не пытайтесь запомнить его – когда мы разберёмся с каждым из них по отдельности, он запомнится сам собой.

Чтобы использовать специальный символ в качестве обычного, он должен быть *экранирован*.

Или, другими словами, перед символом должен быть обратный слэш `'\'`.

Например, нам нужно найти точку `'.'`. В регулярном выражении она означает «любой символ, кроме новой строки», поэтому чтобы найти именно сам символ «точка» – её нужно экранировать: `\.`.

```
alert( "Глава 5.1".match(/\d\.\d/) ); // 5.1
```

Круглые скобки также являются специальными символами, так что для поиска именно скобки нужно использовать `\(`. Пример ниже ищет строку `"g()"`:

```
alert( "function g()".match(/g\(\)/) ); // "g()"
```

Сам символ слэш `'/'`, хотя и не является специальными символом в регулярных выражениях, но открывает-закрывает регэксп в синтаксисе `/...pattern.../`, поэтому его тоже нужно экранировать.

Так выглядит поиск слэша `'/'`:

```
alert( "/".match(/\//) ); // '/'
```

Ну и, наконец, если нам нужно найти сам обратный слэш `\`, то его нужно просто задублировать.

Так выглядит поиск обратного слэша `"\"`:

```
alert( "1\2".match(/\\/) ); // '\'
```

Итого

Мы рассмотрели классы для поиска типов символов:

- `\d` – цифры.
- `\D` – не-цифры.
- `\s` – пробельные символы, переводы строки.
- `\S` – всё, кроме `\s`.
- `\w` – латинница, цифры, подчёркивание `'_'`.
- `\W` – всё, кроме `\w`.
- `'.'` – точка обозначает любой символ, кроме перевода строки.

Если хочется поискать именно сочетание `"\d"` или символ «точка», то его экранируют обратным слэшем, вот так: `\.`

Заметим, что регулярное выражение может также содержать перевод строки `\n`, табуляцию `\t` и прочие спецсимволы для строк. Конфликта с классами не происходит, так как для них зарезервированы другие буквы.

Задачи

Найдите время

Время имеет формат `часы:минуты`. И часы и минуты состоят из двух цифр, например: `09:00`.

Напишите регулярное выражение для поиска времени в строке: `Завтрак в 09:00.`

P.S. В этой задаче выражению позволительно найти и некорректное время, например `25:99`.

К решению

Наборы и диапазоны [...]

Если в регулярном выражении несколько символов или символьных классов заключены в квадратные скобки `[…]`, то это означает "искать любой символ из указанных в `[…]`".

Набор

Например, `[eao]` означает любой символ из этих трёх: `'a'`, `'e'`, или `'o'`.

Такое обозначение называют *набором*. Наборы используются в регулярном выражении наравне с обычными символами:

```
// найти [г или т], а затем "оп"
alert( "Гоп-стоп".match(/[гт]оп/gi) ); // "Гоп", "топ"
```

Обратим внимание: несмотря на то, что в наборе указано несколько символов, в совпадении должен присутствовать *ровно один* из них.

Поэтому в примере ниже нет результатов:

```
// найти "В", затем [у или а], затем "ля"
alert( "Вуаля".match(/В[уа]ля/) ); // совпадений нет
```

Поиск подразумевает:

- `В`,
- затем *одну* из букв набора `[уа]`,
- а затем `ля`

Таким образом, совпадение было бы для строки `Вуля` или `Валя`.

Диапазоны

Квадратные скобки могут также содержать *диапазоны символов*.

Например, `[a-z]` – произвольный символ от `a` до `z`, `[0-5]` – цифра от `0` до `5`.

В примере ниже мы будем искать `"x"`, после которого идёт два раза любая цифра или буква от A до F:

```
// найдёт "xAF"
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) );
```

Обратим внимание, в слове `Exception` есть сочетание `xce`, но оно не подошло, потому что буквы в нём маленькие, а в диапазоне `[0-9A-F]` – большие.

Если хочется искать и его тоже, можно добавить в скобки диапазон `a-f`: `[0-9A-Fa-f]`. Или же просто указать у всего регулярного выражения флаг `i`.

Символьные классы – всего лишь более короткие записи для диапазонов, в частности:

- \d – то же самое, что `[0-9]`,
- \w – то же самое, что `[a-zA-Z0-9_]`,
- \s – то же самое, что `[\t\n\v\f\r ]` плюс несколько юникодных пробельных символов.

В квадратных скобках можно использовать и диапазоны и символьные классы – вместе.

Например, нам нужно найти все слова в тексте. Если они на английском – это достаточно просто:

```
var str = "The sun is rising!";

alert( str.match(/\w+/g) ); // The, sun, is, rising
```

В этом примере мы забежали немного вперёд и использовали pattern `\w+`, что означает один или более символов, подходящих под класс pattern `\w`. Позже мы рассмотрим `+` детальнее, а пока – давайте посмотрим, найдутся ли слова на русском?

```
var str = "Солнце встаёт!";

alert( str.match(/\w+/g) ); // null
```

Ничего не найдено! Это можно понять, ведь `\w` – это именно английская букво-цифра, как можно видеть из аналога `[a-zA-Z0-9_]`.

Чтобы находило слово на русском – нужно использовать диапазон, например `/[а-я]/`.

А чтобы на обоих языках – и то и другое вместе:

```
var str = "Солнце (the sun) встаёт!";

alert( str.match(/[\wа-я]+/gi) ); // Солнце, the, sun, вста, т
```

...Присмотритесь внимательно к предыдущему примеру! Вы видите странность? Оно не находит букву `ё`, более того – считает её разрывом в слове. Причина – в кодировке юникод, она подробно раскрыта в главе Строки.

Буква `ё` лежит в стороне от основной кириллицы и её следует добавить в диапазон дополнительно, вот так:

```
var str = "Солнце (the sun) встаёт!";

alert( str.match(/[\wа-яё]+/gi) ); // Солнце, the, sun, встаёт
```

Теперь всё в порядке.

Диапазоны «кроме»

Кроме обычных, существуют также *исключающие* диапазоны: **`[^…]`**.

Квадратные скобки, начинающиеся со знака каретки: `[^…]` находят любой символ, *кроме указанных*.

Например:

- `[^aeyo]` – любой символ, кроме `'a'`, `'e'`, `'y'`, `'o'`.
- `[^0-9]` – любой символ, кроме цифры, то же что `\D`.
- `[^\s]` – любой не-пробельный символ, то же что `\S`.

Пример ниже ищет любые символы, кроме букв, цифр и пробелов:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // "@", "."
```

Не нужно экранирование

Обычно, если мы хотим искать именно точку, а не любой символ, или именно символ `\`, то мы используем экранирование: указываем `\.` или `\\`.

В квадратных скобках большинство специальных символов можно использовать без экранирования, если конечно они не имеют какой-то особый смысл именно внутри квадратных скобок.

То есть, «как есть», без экранирования можно использовать символы:

- Точка `'.'`.
- Плюс `'+'`.
- Круглые скобки `'( )'`.
- Дефис `'-'`, если он находится в начале или конце квадратных скобок, то есть не выделяет диапазон.
- Символ каретки `'^'`, если не находится в начале квадратных скобок.
- А также открывающая квадратная скобка `'['`.

То есть, точка `"."` в квадратных скобках означает не «любой символ», а обычную точку.

Регэксп `[.,]` ищет один из символов «точка» или «запятая».

В примере ниже регэксп `[-().^+]` ищет один из символов `-().^`. Они не экранированы:

```
// Без экранирования
var re = /[-().^+]/g;

alert( "1 + 2 - 3".match(re) ); // найдёт +, -
```

...Впрочем, даже если вы решите «на всякий случай» заэкранировать эти символы, поставив перед ними обратный слэш `\` – вреда не будет:

```
// Всё заэкранировали
var re = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(re) ); // тоже работает: +, -
```

Задачи

Java[^script]

Найдет ли регэксп `/Java[^script]/` что-нибудь в строке `Java`?

А в строке `JavaScript`?

К решению

Найдите время в одном из форматов

Время может быть в формате `часы:минуты` или `часы-минуты`. И часы и минуты состоят из двух цифр, например `09:00`, `21-30`.

Напишите регулярное выражение для поиска времени:

```
var re = /ваше выражение/;
alert( "Завтрак в 09:00. Обед - в 21-30".match(re) ); // 09:00, 21-30
```

К решению

Квантификаторы +, *, ? и {n}

Рассмотрим ту же задачу, что и ранее – взять телефон вида `+7(903)-123-45-67` и найти все числа в нём. Но теперь нас интересуют не цифры по отдельности, а именно числа, то есть результат вида `7, 903, 123, 45, 67`.

Для поиска цифр по отдельности нам было достаточно класса `\d`. Но здесь нужно искать *числа* – последовательности из 1 или более цифр.

Количество {n}

Количество повторений символа можно указать с помощью числа в фигурных скобках: `{n}`.

Такое указание называют *квантификатором* (от англ. quantifier).

У него есть несколько подформ записи:

Точное количество: **`{5}`**

Регэксп `\d{5}` обозначает ровно 5 цифр, в точности как `\d\d\d\d\d`.

Следующий пример находит пятизначное число.

```
alert( "Мне 12345 лет".match(/\d{5}/) ); //  "12345"
```

Количество от-до: **`{3,5}`**

Для того, чтобы найти, например, числа размером от трёх до пяти знаков, нужно указать границы в фигурных скобках: `\d{3,5}`

```
alert( "Мне не 12, а 1234 года".match(/\d{3,5}/) ); // "1234"
```

Последнее значение можно и не указывать. Тогда выражение `\d{3,}` найдет числа, длиной от трех цифр:

```
alert( "Мне не 12, а 345678 лет".match(/\d{3,}/) ); // "345678"
```

В случае с телефоном нам нужны числа – одна или более цифр подряд. Этой задаче соответствует регулярное выражение `\d{1,}`:

```
var str = "+7(903)-123-45-67";

alert( str.match(/\d{1,}/g) ); // 7,903,123,45,67
```

Короткие обозначения

Для самых часто востребованных квантификаторов есть специальные короткие обозначения.

+

Означает «один или более», то же что `{1,}`.

Например, `\d+` находит числа – последовательности из 1 или более цифр:

```
var str = "+7(903)-123-45-67";
alert( str.match(/\d+/g) ); // 7,903,123,45,67
```

?

Означает «ноль или один», то же что и `{0,1}`. По сути, делает символ необязательным.

Например, регэксп `ou?r` найдёт `o`, после которого, возможно, следует `u`, а затем `r`.

Этот регэксп найдёт `or` в слове `color` и `our` в `colour`:

```
var str = "Можно писать color или colour (британский вариант)";
alert( str.match(/colou?r/g) ); // color, colour
```

*****

Означает «ноль или более», то же что `{0,}`. То есть, символ может повторяться много раз или вообще отсутствовать.

Пример ниже находит цифру, после которой идёт один или более нулей:

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```

Сравните это с `'+'` (один или более):

```
alert( "100 10 1".match(/\d0+/g) ); // 100, 10
```

Ещё примеры

Эти квантификаторы принадлежат к числу самых важных «строительных блоков» для сложных регулярных выражений, поэтому мы рассмотрим ещё примеры.

Регэксп «десятичная дробь» (число с точкой внутри): **`\d+\.\d+`**

В действии:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Регэксп «открывающий HTML-тег без атрибутов», такой как **`<span>`** или **`<p>`**: **`/<[a-z]+>/i`**

Пример:

```
alert( "<BODY> ... </BODY>".match(/<[a-z]+>/gi) ); // <BODY>
```

Это регулярное выражение ищет символ `'<'`, за которым идут одна или более букв английского алфавита, и затем `'>'`.

Регэксп «открывающий HTML-тег без атрибутов» (лучше): **`/<[a-z][a-z0-9]*>/i`**

Здесь регулярное выражение расширено: в соответствие со стандартом, HTML-тег может иметь символ цифры на любой позиции, кроме первой, например `<h1>`.

```
alert( "<h1>Привет!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Регэксп «открывающий или закрывающий HTML-тег без атрибутов»: **`/<\/?[a-z][a-z0-9]*>/i`**

В предыдущий паттерн добавили необязательный слэш `/?` перед тегом. Его понадобилось заэкранировать, чтобы JavaScript не принял его за конец шаблона.

```
alert( "<h1>Привет!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

Точнее – значит сложнее

В этих примерах мы видим общее правило, которое повторяется из раза в раз: чем точнее регулярное выражение, тем оно длиннее и сложнее.

Например, для HTML-тегов, скорее всего, подошло бы и более короткое регулярное выражение `<\w+>`.

Так как класс `\w` означает "любая цифра или английская буква или `'_'`", то под такой регэксп подойдут и не теги, например `<_>`. Однако он гораздо проще, чем более точный регэксп `<[a-z][a-z0-9]*>`.

Подойдёт ли нам `<\w+>` или нужно использовать именно `<[a-z][a-z0-9]*>`?

В реальной жизни допустимы оба варианта. Ответ на подобные вопросы зависит от того, насколько реально важна точность и насколько сложно потом будет отфильтровать лишние совпадения (если появятся).

Задачи

Как найти многоточие... ?

важность: 5

Напишите регулярное выражения для поиска многоточий: трёх или более точек подряд.

Проверьте его:

```
var reg = /ваше выражение/g;
alert( "Привет!... Как дела?.....".match(reg) ); // ..., .....
```

К решению

Регулярное выражение для цвета

Напишите регулярное выражение для поиска HTML-цвета, заданного как `#ABCDEF`, то есть `#` и содержит затем 6 шестнадцатеричных символов.

Пример использования:

```
var re = /*...ваше регулярное выражение...*/

var str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert( str.match(re) )  // #121212,#AA00ef
```

К решению

Найдите положительные числа

Создайте регэксп, который ищет все положительные числа, в том числе и с десятичной точкой.

Пример использования:

```
var re = /* ваш регэксп */

var str = "1.5 0 12. 123.4.";

alert( str.match(re) );   // 1.5, 0, 12, 123.4
```

К решению

Найдите десятичные числа

Создайте регэксп, который ищет все числа, в том числе и с десятичной точкой, в том числе и отрицательные.

Пример использования:

```
var re = /* ваш регэксп */

var str = "-1.5 0 2 -123.4.";

alert( str.match(re) );   // -1.5, 0, 2, -123.4
```

К решению

Жадные и ленивые квантификаторы

Квантификаторы – с виду очень простая, но на самом деле очень хитрая штука.

Необходимо очень хорошо понимать, как именно происходит поиск, если конечно мы хотим искать что-либо сложнее чем `/\d+/`.

Для примера рассмотрим задачу, которая часто возникает в типографике – заменить в тексте кавычки вида `"..."` (их называют «английские кавычки») на «кавычки-ёлочки»: `«...»`.

Для этого нужно сначала найти все слова в таких кавычках.

Соответствующее регулярное выражение может выглядеть так: `/".+"/g`, то есть мы ищем кавычку, после которой один или более произвольный символ, и в конце опять кавычка.

Однако, если попробовать применить его на практике, даже на таком простом случае...

```
var reg = /".+"/g;

var str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // "witch" and her "broom"
```

...Мы увидим, что оно работает совсем не так, как задумано!

Вместо того, чтобы найти два совпадения `"witch"` и `"broom"`, оно находит одно: `"witch" and her "broom"`.

Это как раз тот случай, когда *жадность* – причина всех зол.

Жадный поиск

Чтобы найти совпадение, движок регулярных выражений обычно использует следующий алгоритм:

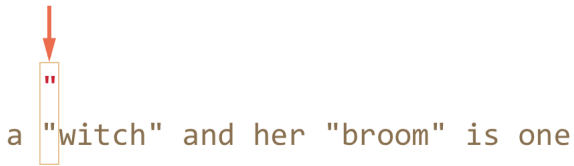
- Для каждой позиции в поисковой строке
  - Проверить совпадение на данной позиции
  - Посимвольно, с учётом классов и квантификаторов сопоставив с ней регулярное выражение.

Это общие слова, гораздо понятнее будет, если мы проследим, что именно он делает для регэкспа `".+"`.

1. Первый символ шаблона – это кавычка `"`.

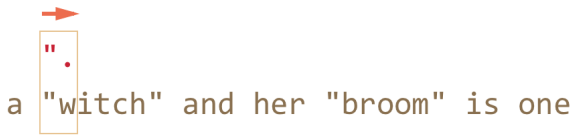
   Движок регулярных выражений пытается сопоставить её на 0-й позиции в строке, но символ `a`, поэтому на 0-й позиции соответствия явно нет.

   Далее он переходит 1ю, 2ю позицию в исходной строке и, наконец, обнаруживает кавычку на 3-й позиции:

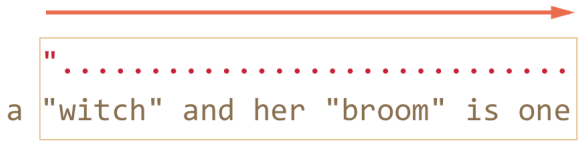
2. Кавычка найдена, далее движок проверяет, есть ли соответствие для остальной части паттерна.

   В данном случае следующий символ шаблона: `.` (точка). Она обозначает «любой символ», так что следующая буква строки `'w'` вполне подходит:

3. Далее «любой символ» повторяется, так как стоит квантификатор `.+`. Движок регулярных выражений берёт один символ за другим, до тех пор, пока у него это получается.

   В данном случае это означает «до конца строки»:

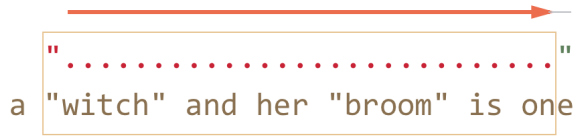
   

4. Итак, текст закончился, движок регулярных выражений больше не может найти «любой символ», он закончил повторения для `.+` и переходит к следующему символу шаблона.

   Следующий символ шаблона – это кавычка. Её тоже необходимо найти, чтобы соответствие было полным. А тут – беда, ведь поисковый текст завершился!

   Движок регулярных выражений понимает, что, наверное, взял многовато `.+` и начинает отступать обратно.

Иными словами, он сокращает текущее совпадение на один символ:

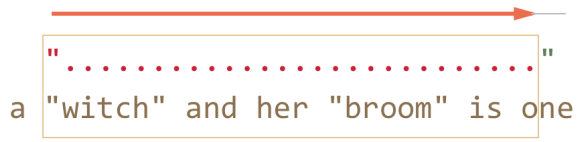


Это называется «фаза возврата» или «фаза бэктрекинга» (backtracking – англ.).

Теперь `.+` соответствует почти вся оставшаяся строка, за исключением одного символа, и движок регулярных выражений ещё раз пытается подобрать соответствие для остатка шаблона, начиная с оставшейся части строки.

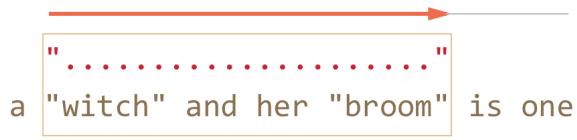
Если бы последним символом строки была кавычка `'"'`, то на этом бы всё и закончилось. Но последний символ `'e'`, так что совпадения нет.

5. ...Поэтому движок уменьшает число повторений `.+` ещё на один символ:



Кавычка `'"'` не совпадает с `'n'`. Опять неудача.

6. Движок продолжает отступать, он уменьшает количество повторений точки `'.'` до тех пор, пока остаток паттерна, то есть в данном случае кавычка `'"'`, не совпадёт:



7. Совпадение получено. Дальнейший поиск по оставшейся части строки `is one` новых совпадений не даст.

Возможно, это не совсем то, что мы ожидали.

В жадном режиме (по умолчанию) регэксп повторяет квантификатор настолько много раз, насколько это возможно, чтобы найти соответствие.

То есть, любой символ `.+` повторился максимальное количество раз, что и привело к такой длинной строке.

А мы, наверное, хотели, чтобы каждая строка в кавычках была независимым совпадением? Для этого можно переключить квантификатор `+` в «ленивый» режим, о котором будет речь далее.

Ленивый режим

Ленивый режим работы квантификаторов – противоположность жадному, он означает «повторять минимальное количество раз».

Его можно включить, если поставить знак вопроса `'?'` после квантификатора, так что он станет таким: `*?` или `+?` или даже `??` для `'?'`.

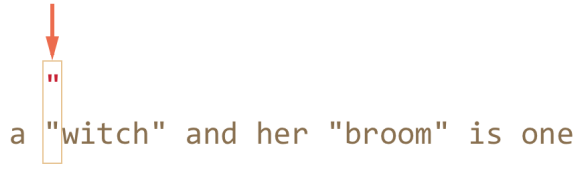
Чтобы не возникло путаницы – важно понимать: обычно `?` сам является квантификатором (ноль или один). Но если он стоит *после другого квантификатора (или даже после себя)*, то обретает другой смысл – в этом случае он меняет режим его работы на ленивый.

Регэксп `/".+?"/g` работает, как задумано – находит отдельно `witch` и `broom`:

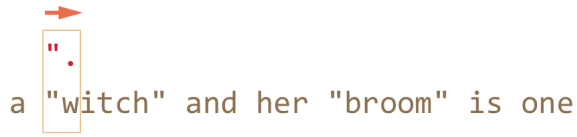
```
var reg = /".+?"/g;
var str = 'a "witch" and her "broom" is one';
alert( str.match(reg) ); // witch, broom
```

Чтобы в точности понять, как поменялась работа квантификатора, разберём поиск по шагам.

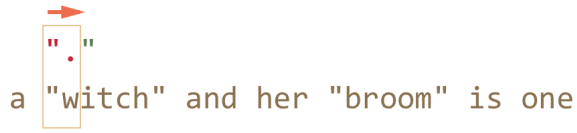
1. Первый шаг – тот же, кавычка `'"'` найдена на 3-й позиции:



2. Второй шаг – тот же, находим произвольный символ `'.'`:

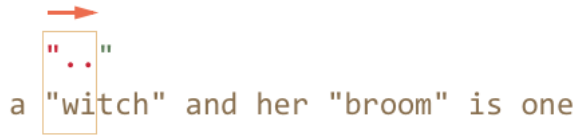


3. А вот дальше – так как стоит ленивый режим работы `+`, то движок не повторит точку (произвольный символ) ещё раз, а останавливается на достигнутом и пытается проверить, есть ли соответствие остальной части шаблона, то есть `'"'`:



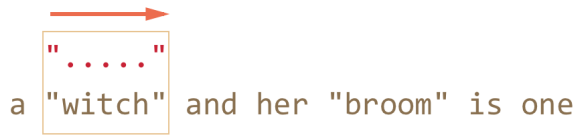
Если бы остальная часть шаблона на данной позиции совпала, то совпадение было бы найдено. Но в данном случае – нет, символ `'i'` не равен „"".

4. Движок регулярных выражений увеличивает количество повторений точки на одно и пытается найти соответствие остатку шаблона ещё раз:



Опять неудача. Тогда поисковой движок увеличивает количество повторений ещё и ещё...

5. Только на пятом шаге поисковой движок наконец находит соответствие для остатка паттерна:



6. Так как поиск происходит с флагом `g`, то он продолжается с конца текущего совпадения, давая ещё один результат:

В примере выше продемонстрирована работа ленивого режима для `+?`. Квантификаторы `+?` и `??` ведут себя аналогично – «ленивый» движок увеличивает количество повторений только в том случае, если для остальной части шаблона на данной позиции нет соответствия.

Ленивость распространяется только на тот квантификатор, после которого стоит `?`.

Прочие квантификаторы остаются жадными.

Например:

```
alert( "123 456".match(/\d+ \d+?/g) ); // 123 4
```

1. Подшаблон `\d+` пытается найти столько цифр, сколько возможно (работает жадно), так что он находит `123` и останавливается, поскольку символ пробела `' '` не подходит под `\d`.
2. Далее в шаблоне пробел, он совпадает.
3. Далее в шаблоне идёт `\d+?`.

   Квантификатор указан в ленивом режиме, поэтому он находит одну цифру `4` и пытается проверить, есть ли совпадение с остатком шаблона.

   Но после `\d+?` в шаблоне ничего нет.

   Ленивый режим без необходимости лишний раз квантификатор не повторит.

   Так как шаблон завершился, то искать дальше, в общем-то нечего. Получено совпадение `123 4`.
4. Следующий поиск продолжится с `5`, но ничего не найдёт.

Конечные автоматы и не только

Современные движки регулярных выражений могут иметь более хитрую реализацию внутренних алгоритмов, чтобы искать быстрее.

Однако, чтобы понять, как работает регулярное выражение, и строить регулярные выражения самому, знание этих хитрых алгоритмов ни к чему. Они служат лишь внутренней оптимизации способа поиска, описанного выше.

Кроме того, сложные регулярные выражения плохо поддаются всяким оптимизациям, так что поиск вполне может работать и в точности как здесь описано.

Альтернативный подход

В данном конкретном случае, возможно искать строки в кавычках, оставаясь в жадном режиме, с использованием регулярного выражения `"[^"]+"`:

```
var reg = /"[^"]+"/g;

var str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // witch, broom
```

Регэксп `"[^"]+"` даст правильные результаты, поскольку ищет кавычку `'"'`, за которой идут столько не-кавычек (исключающие квадратные скобки), сколько возможно.

Так что вторая кавычка автоматически прекращает повторения `[^"]+` и позволяет найти остаток шаблона `"`.

Эта логика ни в коей мере не заменяет ленивые квантификаторы!

Она просто другая. И то и другое бывает полезно.

Давайте посмотрим пример, когда нужен именно такой вариант, а ленивые квантификаторы не подойдут.

Например, мы хотим найти в тексте ссылки вида `<a href="..." class="doc">`, с любым содержанием `href`.

Какое регулярное выражение для этого подойдёт?

Первый вариант может выглядеть так: `/<a href=".*" class="doc">/g`.

Проверим его:

```
var str = '...<a href="link" class="doc">...';
var reg = /<a href=".*" class="doc">/g;

// Сработало!
alert( str.match(reg) ); // <a href="link" class="doc">
```

А если в тексте несколько ссылок?

```
var str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
var reg = /<a href=".*" class="doc">/g;

// Упс! Сразу две ссылки!
alert( str.match(reg) ); // <a href="link1" class="doc">... <a href="link2" class="doc">
```

На этот раз результат неверен.

Жадный `.*` взял слишком много символов.

Соответствие получилось таким:

```
<a href="....................................." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Модифицируем шаблон – добавим ленивость квантификатору `.*?`:

```
var str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
var reg = /<a href=".*?" class="doc">/g;

// Сработало!
alert( str.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Теперь всё верно, два результата:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Почему теперь всё в порядке – для внимательного читателя, после объяснений, данных выше в этой главе, должно быть полностью очевидно.

Поэтому не будем останавливаться здесь на деталях, а попробуем ещё пример:

```
var str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
var reg = /<a href=".*?" class="doc">/g;

// Неправильное совпадение!
alert( str.match(reg) ); // <a href="link1" class="wrong">... <p style="" class="doc">
```

Совпадение – не ссылка, а более длинный текст.

Получилось следующее:

1. Найдено совпадение `<a href="` .
2. Лениво ищем `.*?` , после каждого символа проверяя, есть ли совпадение остальной части шаблона.

   Подшаблон `.*?` будет брать символы до тех пор, пока не найдёт `class="doc">` .

   В данном случае этот поиск закончится уже за пределами ссылки, в теге `<p>` , вообще не имеющем отношения к `<a>` .
3. Получившееся совпадение:

```
<a href="..................................." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

Итак, ленивость нам не помогла.

Необходимо как-то прекратить поиск `.*` , чтобы он не вышел за пределы кавычек.

Для этого мы используем более точное указание, какие символы нам подходят, а какие нет.

Правильный вариант: `[^"]*` . Этот шаблон будет брать все символы до ближайшей кавычки, как раз то, что требуется.

Рабочий пример:

```
var str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
var str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
var reg = /<a href="[^"]*" class="doc">/g;

// Работает!
alert( str1.match(reg) ); // null, совпадений нет, и это верно
alert( str2.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Итого

Квантификаторы имеют два режима работы:

Жадный

Режим по умолчанию – движок регулярных выражений повторяет его по-максимуму. Когда повторять уже нельзя, например нет больше цифр для `\d+` , он продолжает поиск с оставшейся части текста. Если совпадение найти не удалось – отступает обратно, уменьшая количество повторений.

Ленивый

При указании после квантификатора символа `?` он работает в ленивом режиме. То есть, он перед каждым повторением проверяет совпадение оставшейся части шаблона на текущей позиции.

Как мы видели в примере выше, ленивый режим – не панацея от «слишком жадного» забора символов. Альтернатива – более аккуратно настроенный «жадный», с исключением символов. Как мы увидим далее, можно исключать не только символы, но и целые подшаблоны.

Задачи

Совпадение для /d+? d+/

Что будет при таком поиске, когда сначала стоит ленивый, а потом жадный квантификаторы?

```
"123 456".match(/\d+? \d+/g) ); // какой результат?
```

К решению

Различие между "[^"]*" и ".*?"

Регулярные выражения `"[^"]*"` и `".*?"` – при выполнении одинаковы?

Иначе говоря, существует ли такая строка, на которой они дадут разные результаты? Если да – дайте такую строку.

К решению

Найти HTML-комментарии

Найдите все HTML-комментарии в тексте:

```
var re = ..ваш регэксп..

var str = '.. <!-- Мой -- комментарий \n тест --> ..  <!----> .. ';

alert( str.match(re) ); // '<!-- Мой -- комментарий \n тест -->', '<!---->'
```

К решению

Найти HTML-теги

Создайте регулярное выражение для поиска всех (открывающихся и закрывающихся) HTML-тегов вместе с атрибутами.

Пример использования:

```
var re = /* ваш регэксп */

var str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(re) ); // '<a href="/">', '<input type="radio" checked>', '<b>'
```

В этой задаче можно считать, что тег начинается с `<`, заканчивается `>` и может содержать внутри любые символы, кроме `<` и `>`.

Но хотя бы один символ внутри тега должен быть: `<>` – не тег.

К решению

Скобочные группы

Часть шаблона может быть заключена в скобки `(...)`. Такие выделенные части шаблона называют «скобочными выражениями» или «скобочными группами».

У такого выделения есть два эффекта:

1. Он позволяет выделить часть совпадения в отдельный элемент массива при поиске через String#match или RegExp#exec.
2. Если поставить квантификатор после скобки, то он применится *ко всей скобке*, а не всего лишь к одному символу.

Пример

В примере ниже, шаблон `(go)+` находит один или более повторяющихся `'go'`:

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Без скобок, шаблон `/go+/` означал бы `g`, после которого идёт одна или более `o`, например: `goooo`. А скобки «группируют» `(go)` вместе.

Содержимое группы

Скобки нумеруются слева направо. Поисковой движок запоминает содержимое каждой скобки и позволяет обращаться к нему – в шаблоне и строке замены и, конечно же, в результатах.

Например, найти HTML-тег можно шаблоном `<.*?>`.

После поиска мы захотим что-то сделать с результатом. Для удобства заключим содержимое `<...>` в скобки: `<(.*?)>`. Тогда оно будет доступно отдельно.

При поиске методом String#match в результирующем массиве будет сначала всё совпадение, а далее – скобочные группы. В шаблоне `<(.*?)>` скобочная группа только одна:

```
var str = '<h1>Привет, мир!</h1>';
var reg = /<(.*?)>/;

alert( str.match(reg) ); // массив: <h1>, h1
```

Заметим, что метод String#match выдаёт скобочные группы только при поиске без флага `/.../g`. В примере выше он нашёл только первое совпадение `<h1>`, а закрывающий `</h1>` не нашёл, поскольку без флага `/.../g` ищется только первое совпадение.

Для того, чтобы искать и с флагом `/.../g` и со скобочными группами, используется метод RegExp#exec:

```
var str = '<h1>Привет, мир!</h1>';
var reg = /<(.*?)>/g;

var match;

while ((match = reg.exec(str)) !== null) {
  // сначала выведет первое совпадение: <h1>,h1
```

```
  // затем выведет второе совпадение: </h1>,/h1
  alert(match);
}
```

Теперь найдено оба совпадения `<(.*?)>`, каждое – массив из полного совпадения и скобочных групп (одна в данном случае).

Вложенные группы

Скобки могут быть и вложенными. В этом случае нумерация также идёт слева направо.

Например, при поиске тега в `<span class="my">` нас может интересовать:

1. Содержимое тега целиком: `span class="my"`.
2. В отдельную переменную для удобства хотелось бы поместить тег: `span`.
3. Также может быть удобно отдельно выделить атрибуты `class="my"`.

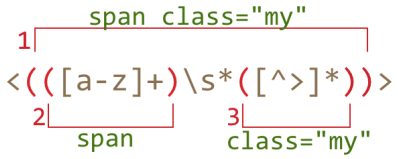
Добавим скобки в регулярное выражение:

```
var str = '<span class="my">';

var reg = /<(([a-z]+)\s*([^>]*))>/;

alert( str.match(reg) ); // <span class="my">, span class="my", span, class="my"
```

Вот так выглядят скобочные группы:



На нулевом месте – всегда совпадение полностью, далее – группы. Нумерация всегда идёт слева направо, по открывающей скобке.

В данном случае получилось, что группа 1 включает в себя содержимое групп 2 и 3. Это совершенно нормальная ситуация, которая возникает, когда нужно выделить что-то отдельное внутри большей группы.

Даже если скобочная группа необязательна и не входит в совпадение, соответствующий элемент массива существует (и равен **`undefined`**).

Например, рассмотрим регэксп `a(z)?(c)?`. Он ищет `"a"`, за которой не обязательно идёт буква `"z"`, за которой необязательно идёт буква `"c"`.

Если напустить его на строку из одной буквы `"a"`, то результат будет таков:

```
var match = 'a'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // a
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

Массив получился длины `3`, но все скобочные группы – `undefined`.

А теперь более сложная ситуация, строка `ack`:

```
var match = 'ack'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac, всё совпадение
alert( match[1] ); // undefined, для (z)? ничего нет
alert( match[2] ); // c
```

Длина массива результатов по-прежнему `3`. Она постоянна. А вот для скобочной группы `(z)?` в ней ничего нет, поэтому результат: `["ac", undefined, "c"]`.

Исключение из запоминания через ?:

Бывает так, что скобки нужны, чтобы квантификатор правильно применился, а вот запоминать их содержимое в массиве не нужно.

Скобочную группу можно исключить из запоминаемых и нумеруемых, добавив в её начало `?:`.

Например, мы хотим найти `(go)+`, но содержимое скобок (go) в отдельный элемент массива выделять не хотим.

Для этого нужно сразу после открывающей скобки поставить `?:`, то есть: `(?:go)+`.

Например:

```
var str = "Gogo John!";
var reg = /(?:go)+ (\w+)/i;
```

```
var result = str.match(reg);

alert( result.length ); // 2
alert( result[1] ); // John
```

В примере выше массив результатов имеет длину `2` и содержит только полное совпадение и результат `(\w+)`. Это удобно в тех случаях, когда содержимое скобок нас не интересует.

Задачи

Найдите цвет в формате #abc или #abcdef

Напишите регулярное выражение, которое находит цвет в формате `#abc` или `#abcdef`. То есть, символ `#`, после которого идут 3 или 6 шестнадцатиричных символа.

Пример использования:

```
var re = /* ваш регэксп */

var str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(re) ); // #3f3 #AA0ef
```

P.S. Значения из любого другого количества букв, кроме 3 и 6, такие как `#abcd`, не должны подходить под регэксп.

К решению

Разобрать выражение

Арифметическое выражение состоит из двух чисел и операции между ними, например:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

Список операций: `"+"`, `"-"`, `"*"` и `"/"`.

Также могут присутствовать пробелы вокруг оператора и чисел.

Напишите функцию, которая будет получать выражение и возвращать массив из трёх аргументов:

1. Первое число.
2. Оператор.
3. Второе число.

К решению

Обратные ссылки: \n и $n

Скобочные группы можно не только получать в результате.

Движок регулярных выражений запоминает их содержимое, и затем его можно использовать как в самом паттерне, так и в строке замены.

Группа в строке замены

Ссылки в строке замены имеют вид `$n`, где `n` – это номер скобочной группы.

Вместо `$n` подставляется содержимое соответствующей скобки:

```
var name = "Александр Пушкин";

name = name.replace(/([а-яё]+) ([а-яё]+)/i, "$2, $1");
alert( name ); // Пушкин, Александр
```

В примере выше вместо `$2` подставляется второе найденное слово, а вместо `$1` – первое.

Группа в шаблоне

Выше был пример использования содержимого групп в строке замены. Это удобно, когда нужно реорганизовать содержимое или создать новое с использованием старого.

Но к скобочной группе можно также обратиться в самом поисковом шаблоне, ссылкой вида `\номер`.

Чтобы было яснее, рассмотрим это на реальной задаче – необходимо найти в тексте строку в кавычках. Причём кавычки могут быть одинарными `'...'` или двойными `"..."` – и то и другое должно искаться корректно.

Как такие строки искать?

Можно в регэкспе предусмотреть произвольные кавычки: `['"](.*?)['"]`. Такой регэксп найдёт строки вида `"..."`, `'...'`, но он даст неверный ответ в случае, если одна кавычка ненароком оказалась внутри другой, как например в строке `"She's the one!"`:

```
var str = "He said: \"She's the one!\".";

var reg = /['"](.*?)['"]/g;

// Результат не соответствует замыслу
alert( str.match(reg) ); // "She'
```

Как видно, регэксп нашёл открывающую кавычку `"`, затем текст, вплоть до новой кавычки `'`, которая закрывает соответствие.

Для того, чтобы попросить регэксп искать закрывающую кавычку – такую же, как открывающую, мы обернём её в скобочную группу и используем обратную ссылку на неё:

```
var str = "He said: \"She's the one!\".";

var reg = /(['"])(.*?)\1/g;

alert( str.match(reg) ); // "She's the one!"
```

Теперь работает верно! Движок регулярных выражений, найдя первое скобочное выражение – кавычку `(['"])`, запоминает его и далее `\1` означает «найти то же самое, что в первой скобочной группе».

Обратим внимание на два нюанса:

- Чтобы использовать скобочную группу в строке замены – нужно использовать ссылку вида `$1`, а в шаблоне – обратный слэш: `\1`.
- Чтобы в принципе иметь возможность обратиться к скобочной группе – не важно откуда, она не должна быть исключена из запоминаемых при помощи `?:`. Скобочные группы вида `(?:...)` не участвуют в нумерации.

Задачи

Найдите пары тегов

ББ-тег имеет вид `[имя]...[/имя]`, где имя – слово, одно из: `b`, `url`, `quote`.

Например:

```
[b]текст[/b]
[url]http://ya.ru[/url]
```

ББ-теги могут быть вложенными, но сам в себя тег быть вложен не может, например:

```
Допустимо:
[url] [b]http://ya.ru[/b] [/url]
[quote] [b]текст[/b] [/quote]

Нельзя:
[b][b]текст[/b][/b]
```

Создайте регулярное выражение для поиска ББ-тегов и их содержимого.

Например:

```
var re = /* регулярка */

var str = "..[url]http://ya.ru[/url]..";
alert( str.match(re) ); // [url]http://ya.ru[/url]
```

Если теги вложены, то нужно искать самый внешний тег (при желании можно будет продолжить поиск в его содержимом):

```
var re = /* регулярка */

var str = "..[url][b]http://ya.ru[/b][/url]..";
alert( str.match(re) ); // [url][b]http://ya.ru[/b][/url]
```

К решению

Альтернация (или) |

Альтернация – термин в регулярных выражениях, которому в русском языке соответствует слово «ИЛИ». Она обозначается символом вертикальной черты `|` и позволяет выбирать между вариантами.

Например, нам нужно найти языки программирования: HTML, PHP, Java и JavaScript.

Соответствующее регулярное выражение: `html|php|java(script)?`.

Пример использования:

```
var reg = /html|php|css|java(script)?/gi

var str = "Сначала появился HTML, затем CSS, потом JavaScript"

alert( str.match(reg) ) // 'HTML', 'CSS', 'JavaScript'
```

Мы уже знаем похожую вещь – квадратные скобки. Они позволяют выбирать между символами, например `gr[ae]y` найдёт `gray`, либо `grey`.

Альтернация работает уже не посимвольно, а на уровне фраз и подвыражений. Регэксп `A|B|C` обозначает поиск одного из выражений: `A`, `B` или `C`, причём в качестве выражений могут быть другие, сколь угодно сложные регэкспы.

Для указания границ альтернации используют скобки `(...)`, например: `before(XXX|YYY)after` будет искать `beforeXXXafter` или `beforeYYYafter`.

Задачи

Найдите языки программирования

Существует много языков программирования, например Java, JavaScript, PHP, C, C++.

Напишите регулярное выражение, которое найдёт их все в строке «Java JavaScript PHP C++ C»

К решению

Найдите строки в кавычках

Найдите в тексте при помощи регэкспа строки в двойных кавычках `"..."`.

В строке поддерживается экранирование при помощи слеша – примерно в таком же виде, как в обычных строках JavaScript. То есть, строка может содержать любые символы, экранированные слэшем, в частности: `\"`, `\n`, и даже сам слэш в экранированном виде: `\\`.

Здесь особо важно, что двойная кавычка после слэша не оканчивает строку, а считается её частью. В этом и состоит основная сложность задачи, которая без этого условия была бы элементарной.

Пример совпадающих строк:

```
.. "test me" ..  (обычная строка)
.. "Скажи \"Привет\"!" ... (строка с кавычками внутри)
.. "\r\n\\" ..  (строка со спец. символами и слэшем внутри)
```

Заметим, что в JavaScript такие строки удобнее всего задавать в одинарных кавычках, и слеши придётся удвоить (в одинарных кавычках они являются экранирующими символами):

Пример задания тестовой строки в JavaScript:

```
var str = ' .. "test me" .. "Скажи \\"Привет\\"!" .. "\\r\\n\\\\" .. ';

// эта строка будет такой:
alert(str); //  .. "test me" .. "Скажи \"Привет\"!" .. "\r\n\\" ..
```

К решению

Найдите тег style

Напишите регулярное выражение, которое будет искать в тексте тег `<style>`. Подходят как обычный тег `<style>`, так и вариант с атрибутами `<style type="...">`.

Но регулярное выражение не должно находить `<styler>`!

Использование:

```
var re = ваш регэксп

alert( "<style> <styler> <style test>".match(re) ); // <style>, <style test>
```

К решению

Начало строки ^ и конец $

Знак каретки `'^'` и доллара `'$'` имеют в регулярном выражении особый смысл. Их называют «якорями» (anchor – англ.).

Каретка `^` совпадает в начале текста, а доллар `$` – в конце.

Якоря являются не символами, а проверками.

До этого мы говорили о регулярных выражениях, которые ищут один или несколько символов. Если совпадение есть – эти символы включаются в результат.

А якоря – не такие. Когда поиск ходит до якоря – он проверяет, есть ли соответствие, если есть – продолжает идти по шаблону, не прибавляя ничего к результату.

Каретку `^` обычно используют, чтобы указать, что регулярное выражение необходимо проверить именно с начала текста.

Например, без каретки найдёт все числа:

```
var str = '100500 попугаев съели 500100 бананов!';
alert( str.match(/\d+/ig) ); // 100500, 500100 (нашло все числа)
```

А с кареткой – только первое:

```
var str = '100500 попугаев съели 500100 бананов!';
alert( str.match(/^\d+/ig) ); // 100500 (только в начале строки)
```

Знак доллара `$` используют, чтобы указать, что паттерн должен заканчиваться в конце текста.

Аналогичный пример с долларом для поиска числа в конце:

```
var str = '100500 попугаев съели 500100';
alert( str.match(/\d+$/ig) ); // 500100
```

Оба якоря используют одновременно, если требуется, чтобы шаблон охватывал текст с начала и до конца. Обычно это требуется при валидации.

Например, мы хотим проверить, что в переменной `num` хранится именно десятичная дробь.

Ей соответствует регэксп `\d+\.\d+`. Но простой поиск найдёт дробь в любом тексте:

```
var num = "ля-ля 12.34";
alert( num.match(/\d+\.\d+/ig) ); // 12.34
```

Наша же задача – проверить, что `num` *целиком* соответствует паттерну `\d+\.\d+`.

Для этого обернём шаблон в якоря `^...$`:

```
var num = "ля-ля 12.34";
alert( num.match(/^\d+\.\d+$/ig) ); // null, не дробь

var num = "12.34";
alert( num.match(/^\d+\.\d+$/ig) ); // 12.34, дробь!
```

Теперь поиск ищет начало текста, за которым идёт число, затем точка, ещё число и конец текста. Это как раз то, что нужно.

Задачи

Регэксп ^$

Предложите строку, которая подойдёт под регулярное выражение `^$`.

К решению

Проверьте MAC-адрес

MAC-адрес сетевого интерфейса состоит из шести двузначных шестнадцатеричных чисел, разделённых двоеточием.

Например: `'01:32:54:67:89:AB'`.

Напишите регулярное выражение, которое по строке проверяет, является ли она корректным MAC-адресом.

Использование:

```
var re = ваш регэксп

alert( re.test('01:32:54:67:89:AB') ); // true
```

```
alert( re.test('0132546789AB') ); // false (нет двоеточий)

alert( re.test('01:32:54:67:89') ); // false (5 чисел, а не 6)

alert( re.test('01:32:54:67:89:ZZ') ) // false (ZZ в конце)
```

К решению

Многострочный режим, флаг "m"

Многострочный режим включается, если у регэкспа есть флаг `/m`.

В этом случае изменяется поведение `^` и `$`.

В многострочном режиме якоря означают не только начало/конец текста, но и начало/конец строки.

Начало строки ^

В примере ниже текст состоит из нескольких строк. Паттерн `/^\d+/gm` берёт число с начала каждой строки:

```
var str = '1е место: Винни\n' +
  '2е место: Пятачок\n' +
  '33е место: Слонопотам';

alert( str.match(/^\d+/gm) ); // 1, 2, 33
```

Обратим внимание – без флага `/m` было бы найдено только первое число:

```
var str = '1е место: Винни\n' +
  '2е место: Пятачок\n' +
  '33е место: Слонопотам';

alert( str.match(/^\d+/g) ); // 1
```

Это потому что в обычном режиме каретка `^` – это только начало текста, а в многострочном – начало любой строки.

Движок регулярных выражений двигается по тексту, и как только видит начало строки, начинает искать там `\d+`.

Конец строки $

Символ доллара `$` ведёт себя аналогично.

Регулярное выражение `[а-я]+$` в следующем примере находит последнее слово в каждой строке:

```
var str = '1е место: Винни\n' +
  '2е место: Пятачок\n' +
  '33е место: Слонопотам';

alert( str.match(/[а-я]+$/gim) ); // Винни,Пятачок,Слонопотам
```

Без флага `m` якорь `$` обозначал бы конец всего текста, и было бы найдено только последнее слово.

Якорь $ против \n

Для того, чтобы найти конец строки, можно использовать не только `$`, но и символ `\n`.

Но, в отличие от `$`, символ `\n` во-первых берёт символ в результат, а во-вторых – не совпадает в конце текста (если, конечно, последний символ – не конец строки).

Посмотрим, что будет с примером выше, если вместо `[а-я]+$` использовать `[а-я]+\n`:

```
var str = '1е место: Винни\n' +
  '2е место: Пятачок\n' +
  '33е место: Слонопотам';

alert( str.match(/[а-я]+\n/gim) );
/*
Винни
,Пятачок
*/
```

Всего два результата: `Винни\n` (с символом перевода строки) и `Пятачок\n`. Последнее слово «Слонопотам» здесь не даёт совпадения, так как после него нет перевода строки.

Итого

В мультистрочном режиме:

- Символ `^` означает начало строки.
- Символ `$` означает конец строки.

Оба символа являются проверками, они не добавляют ничего к результату. Про них также говорят, что «они имеют нулевую длину».

Предпросмотр (неготово)

Требуется добавить главу про предпросмотр lookahead.

Чёрная дыра бэктрекинга

Некоторые регулярные выражения, с виду являясь простыми, могут выполняться оооочень долго, и даже «подвешивать» интерпретатор JavaScript.

Рано или поздно, с этим сталкивается любой разработчик, потому что нечаянно создать такое регулярное выражение – легче лёгкого.

Типична ситуация, когда регулярное выражение до поры до времени работает нормально, и вдруг на каком-то тексте как начнёт «подвешивать» интерпретатор и есть 100% процессора.

Это может стать уязвимостью. Например, если JavaScript выполняется на сервере, то при разборе данных, присланных посетителем, он может зависнуть, если использует подобный регэксп. На клиенте тоже возможно подобное, при использовании регэкспа для подсветки синтаксиса.

Такие уязвимости «убивали» почтовые сервера и системы обмена сообщениями и до появления JavaScript, и наверно будут «убивать» и после его исчезновения. Так что мы просто обязаны с ними разобраться.

Пример

План изложения у нас будет таким:

1. Сначала посмотрим на проблему в реальной ситуации.
2. Потом упростим реальную ситуацию до «корней» и увидим, откуда она берётся.

Рассмотрим, например, поиск по HTML.

Мы хотим найти теги с атрибутами, то есть совпадения вида `<a href="..." class=doc ...>`.

Самый простой способ это сделать – `<[^>]*>`. Но он же и не совсем корректный, так как тег может выглядеть так: `<a test="<>" href="#">`. То есть, внутри «закавыченного» атрибута может быть символ `>`. Простейший регэксп на нём остановится и найдёт `<a test="<>`.

Соответствие:

```
<[^>]*....>
<a test="<>" href="#">
```

А нам нужен весь тег.

Для того, чтобы правильно обрабатывать такие ситуации, нужно учесть их в регулярном выражении. Оно будет иметь вид `<тег (ключ=значение)*>`.

Если перевести на язык регэкспов, то: `<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>`:

1. `<\w+` – начало тега
2. `(\s*\w+=(\w+|"[^"]*")\s*)*` – произвольное количество пар вида `слово=значение`, где «значение» может быть также словом `\w+`, либо строкой в кавычках `"[^"]*"`.

Мы пока не учитываем все детали грамматики HTML, ведь строки возможны и в „одинарных“ кавычках, но на данный момент этого достаточно. Главное, что регулярное выражение получилось в меру простым и понятным.

Испытаем полученный регэксп в действии:

```
var reg = /<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>/g;

var str='...<a test="<>" href="#">... <b>...';

alert( str.match(reg) ); // <a test="<>" href="#">, <b>
```

Отлично, всё работает! Нашло как длинный тег `<a test="<>" href="#">`, так и одинокий `<b>`.

А теперь – демонстрация проблемы.

Если запустить пример ниже, то он может подвесить браузер:

```
var reg = /<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>/g;

var str = "<tag a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  \
a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b";

// Этот поиск будет выполняться очень, очень долго
alert( str.match(reg) );
```

Некоторые движки регулярных выражений могут в разумное время разобраться с таким поиском, но большинство – нет.

В чём дело? Почему несложное регулярное выражение на такой небольшой строке «виснет» наглухо?

Упростим ситуацию, удалив тег и возможность указывать строки в кавычках:

```
// только атрибуты, разделённые пробелами
var reg = /<(\s*\w+=\w+\s*)*>/g;

var str = "<a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  \
a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b";

// Этот поиск будет выполняться очень, очень долго
alert( str.match(reg) );
```

То же самое.

На этом мы закончим с демонстрацией «практического примера» и перейдём к разбору происходящего.

Бектрекинг

В качестве ещё более простого регулярного выражения, рассмотрим `(\d+)*$`.

В большинстве движков регэкспов, например в Chrome или IE, этот поиск выполняется очень долго (осторожно, может «подвесить» браузер):

```
alert( '123456789012345678901234567890123456789z'.match(/(\d+)*$/) );
```

В чём же дело, что не так с регэкспом?

Внимательный читатель, посмотрев на него, наверняка удивится, ведь он «какой-то странный». Квантификатор `*` здесь выглядит лишним.

Если хочется найти число, то с тем же успехом можно искать `\d+$`.

Да, этот регэксп носит искусственный характер, но, разобравшись с ним, мы поймём и практический пример, данный выше. Причина их медленной работы одинакова.

В целом, с регэкспом «всё так», синтаксис вполне допустимый. Проблема в том, как выполняется поиск по нему.

Посмотрим, что происходит при поиске в строке `123456789z`:

1. Первым делом, движок регэкспов пытается найти `\d+`. Плюс `+` является жадным по умолчанию, так что он хватает все цифры, какие может:

   ```
   \d+.......
   (123456789)z
   ```

2. Затем движок пытается применить звёздочку вокруг скобок `(\d+)*`, но больше цифр нет, так что звёздочка не даёт повторений.

   Затем в шаблоне идёт символ конца строки `$`, а в тексте – символ `z`.

   ```
            X
   \d+.......$
   (123456789)z
   ```

   Соответствия нет.

3. Так как соответствие не найдено, то «жадный» плюс `+` отступает на один символ (бэктрекинг).

   Теперь `\d+` – это все цифры, за исключением последней:

   ```
   \d+.......
   (12345678)9z
   ```

4. После бэктрекинга, `\d+` содержит всё число, кроме последней цифры. Движок снова пытается найти совпадение, уже с новой позиции (`9`).

   Звёздочка `(\d+)*` теперь может быть применена – она даёт число `9`:

   ```
   \d+.......\d+
   (12345678)(9)z
   ```

   Движок пытается найти `$`, но это ему не удаётся – на его пути опять `z`:

   ```
              X
   \d+.......\d+
   (12345678)(9)z
   ```

   Так как совпадения нет, то поисковой движок отступает назад ещё раз.

5. Теперь первое число `\d+` будет содержать 7 цифр, а остаток строки `89` становится вторым `\d+`:

```
            x
\d+......\d+
(1234567)(89)z
```

Увы, всё ещё нет соответствия для `$`.

Поисковой движок снова должен отступить назад. При этом последний жадный квантификатор отпускает символ. В данном случае это означает, что укорачивается второй `\d+`, до одного символа `8`, и звёздочка забирает следующий `9`.

```
              x
\d+......\d+\d+
(1234567)(8)(9)z
```

6. ...И снова неудача. Второе и третье `\d+` отступили по-максимуму, так что сокращается снова первое число, до `123456`, а звёздочка берёт оставшееся:

```
            x
\d+.......\d+
(123456)(789)z
```

Снова нет совпадения. Процесс повторяется, последний жадный квантификатор `+` отпускает один символ (`9`):

```
              x
\d+.....\d+ \d+
(123456)(78)(9)z
```

7. ...И так далее.

Получается, что движок регулярных выражений перебирает все комбинации из `123456789` и их подпоследовательности. А таких комбинаций очень много.

На этом месте умный читатель может воскликнуть: «Во всём виноват бэктрекинг? Давайте включим ленивый режим – и не будет никакого бэктрекинга!»

Что ж, заменим `\d+` на `\d+?` и посмотрим (аккуратно, может подвесить браузер):

```
alert( '12345678901234567890123456789123456789z'.match(/(\d+?)*$/) );
```

Не помогло!

Ленивые регулярные выражения делают то же самое, но в обратном порядке.

Просто подумайте о том, как будет в этом случае работать поисковой движок.

Некоторые движки регулярных выражений содержат хитрые проверки и конечные автоматы, которые позволяют избежать бесконечного перебора или кардинально ускорить его, но не все движки и не всегда.

Возвращаясь к примеру выше – при поиске `<(\s*\w+=\w+\s*)*>` в строке `<a=b a=b a=b a=b` происходит то же самое.

Поиск успешно начинается, выбирается некая комбинация из `\s*\w+=\w+\s*`, которая, так как в конце нет `>`, оказывается не подходящей. Движок честно отступает, пробует другую комбинацию – и так далее.

Что делать?

Проблема – в сверхмноговариантном переборе.

Движок регулярных выражений перебирает кучу возможных вариантов скобок там, где это не нужно.

Например, в регэкспе `(\d+)*$` нам (людям) очевидно, что в `(\d+)` откатываться не нужно. От того, что вместо одного `\d+` у нас два независимых `\d+\d+`, ничего не изменится.

Без разницы:

```
\d+........
(123456789)z

\d+...\d+....
(1234)(56789)z
```

Если вернуться к более реальному примеру `<(\s*\w+=\w+\s*)*>` то сам алгоритм поиска, который у нас в голове, предусматривает, что мы «просто» ищем тег, а потом пары `атрибут=значение` (сколько получится).

Никакого «отката» здесь не нужно.

В современных регулярных выражениях для решения этой проблемы придумали «possessive» (сверхжадные? неоткатные? точный перевод пока не устоялся) квантификаторы, которые вообще не используют бэктрегинг.

То есть, они даже проще, чем «жадные» – берут максимальное количество символов и всё. Поиск продолжается дальше. При несовпадении никакого возврата не происходит.

Это, с одной стороны, уменьшает количество возможных результатов, но, с другой стороны, в ряде случаев очевидно, что возврат (уменьшение количество повторений квантификатора) результата не даст. А только потратит время, что как раз и доставляет проблемы. Как раз такие ситуации и описаны выше.

Есть и другое средство – «атомарные скобочные группы», которые запрещают перебор внутри скобок, по сути позволяя добиваться того же, что и сверхжадные квантификаторы,

К сожалению, в JavaScript они не поддерживаются.

Однако, можно получить подобный эффект при помощи предпросмотра. Подробное описание соответствия с учётом синтаксиса сверхжадных квантификаторов и атомарных групп есть в статьях Regex: Emulate Atomic Grouping (and Possessive Quantifiers) with LookAhead и Mimicking Atomic Groups, здесь же мы останемся в рамках синтаксиса JavaScript.

Взятие максимального количества повторений `a+` без отката выглядит так: `(?=(a+))\1`.

То есть, иными словами, предпросмотр `?=` ищет максимальное количество повторений `a+`, доступных с текущей позиции. А затем они «берутся в результат» обратной ссылкой `\1`. Дальнейший поиск – после найденных повторений.

Откат в этой логике в принципе не предусмотрен, поскольку предпросмотр «откатываться» не умеет. То есть, если предпросмотр нашёл 5 штук `a+`, и в результате поиск не удался, то он не будет откатываться на 4 повторения. Эта возможность в предпросмотре отсутствует, а в данном случае она как раз и не нужна.

Исправим регэксп для поиска тега с атрибутами `<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>`, описанный в начале главы. Используем предпросмотр, чтобы запретить откат на меньшее количество пар `атрибут=значение`:

```
// регэксп для пары атрибут=значение
var attr = /(\s*\w+=(\w+|"[^"]*")\s*)/

// используем его внутри регэкспа для тега
var reg = new RegExp('<\\w+(?=(' + attr.source + '*))\\1>', 'g');

var good = '...<a test="<>" href="#">... <b>...';

var bad = "<tag a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b\
  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b";

alert( good.match(reg) ); // <a test="<>" href="#">, <b>
alert( bad.match(reg) ); // null (нет результатов, быстро)
```

Отлично, всё работает! Нашло как длинный тег `<a test="<>" href="#">`, так и одинокий `<b>`.

О всякой всячине

Статьи на разные темы, которые не вошли в другие разделы.

Эволюция шаблонных систем для JavaScript

Различных шаблонных систем – много.

Они постепенно эволюционировали и развивались.

В этой главе мы разберём, как шёл этот процесс, какие шаблонки «родились», какие бонусы нам даёт использование той или иной шаблонной системы.

Микрошаблоны

Микрошаблоны (англ. microtemplate) мы уже видели на примере `_.template`.

Это HTML со вставками переменных и произвольным JS.

Пример:

```
<div class="menu">
  <span class="title"><%-title%></span>
  <ul>
    <% items.forEach(function(item) { %>
    <li><%-item%></li>
    <% }); %>
  </ul>
</div>
```

Шаблонная система компилирует этот код в JavaScript-функцию с минимальными модификациями, и она уже, запустившись с данными, генерирует результат.

Достоинства и недостатки такого подхода:

Недостатки

- Жёстко привязан к языку JavaScript.
- При ошибке в шаблоне приходится лезть внутрь «страшной» функции

Достоинства

- Простая и быстрая шаблонная система
- Внутри JS-функции доступен полноценный браузерный отладчик, функция хоть и страшна, но понятна.

Код в шаблоне

Включение произвольного JS-кода в шаблон, в теории, позволяет делать в нём всё, что угодно. Но обратная сторона медали – шаблон вместо внятного HTML может стать адским нагромождением разделителей вперемешку с вычислениями. Что рекомендуется делать в шаблонах, а что нет?

Можно разделить код на два типа с точки зрения шаблонизации:

- Бизнес-логика – код, *формирующий данные*, основной код приложения.
- Презентационная логика – код, описывающий, как *показываются данные*.

Например, код, получающий данные с сервера для вывода в таблице – бизнес-логика, а код, форматирующий даты для вывода – презентационная логика.

В шаблонах допустима лишь презентационная логика.

Кросс-платформенность

Зачастую, нужно использовать один и тот же шаблон и в браузере и на сервере.

Например, серверный код генерирует HTML со списком сообщений, а JavaScript на клиенте добавляет к нему новые по мере появления.

...Но как использовать на сервере шаблон с JavaScript, если его основной язык – PHP, Ruby, Java?

Эту проблему можно обойти. На сервер, использующем PHP, Ruby, Java или какой-то другой язык, дополнительно ставится виртуальная машина V8 и настраивается интеграция с ней. Почти все платформы это умеют.

После этого становится возможным запускать JavaScript-шаблоны и передавать им данные в виде объектов, массивов и так далее.

Этот подход может показаться искусственным, но на самом деле он вполне жизнеспособен и используется в ряде крупных проектов.

Прекомпиляция

Эта шаблонка и большинство других систем, которые мы рассмотрим далее, допускают *прекомпиляцию*.

То есть, можно заранее, до выкладывания сайта на «боевой сервер», обработать шаблоны, создать из них JS-функции, объединить их в единый файл и далее, в «боевом окружении» использовать уже их.

Современные системы сборки (brunch, grunt с плагинами и другие) позволяют делать это удобно, а также хранить шаблоны в разных файлах, каждый – в нужной директории с JS-кодом для виджета.

Хелперы и фильтры

JavaScript-вставки не всегда просты и элегантны. Иногда, чтобы что-то сделать, нужно написать порядочно кода.

Для того, чтобы сделать шаблоны компактнее и проще, в них стали добавлять фильтры и хелперы.

- Хелпер (англ. helper) – вспомогательная функция, которая доступна в шаблонах и используется для решения часто возникающих задач.

  В `_.template`, чтобы объявить хелпер, можно просто сделать глобальную функцию. Но это слишком грубо, так не делают. Гораздо лучше – использовать объект `_.templateSettings.imports`, в котором можно указать, какие функции добавлять в шаблоны, или опцию `imports` для `_.template`.

  Пример хелпера – функция `t(phrase)`, которая переводит `phrase` на текущий язык:

  ```
  _.templateSettings.imports.t = function(phrase) {
    // обычно функция перевода немного сложнее, но здесь это не важно
    if (phrase == "Hello") return "Привет";
  }

  // в шаблоне используется хелпер t для перевода
  var compiled = _.template("<div><%=t('Hello')%></div>");
  alert( compiled() ); // <div>Привет</div>
  ```

  Такой хелпер очень полезен для мультиязычных сайтов, когда один шаблон нужно выводить на десяти языках. Нечто подобное используется почти во всех языках и платформах, не только в JavaScript.

- Фильтр – это функция, которая трансформирует данные, например, форматирует дату, сортирует элементы массива и так далее.

  Обычно для фильтров предусмотрен специальный «особо простой и короткий» синтаксис.

  Например, в системе шаблонизации EJS, которая по сути такая же, но мощнее, чем `_.template`, фильтры задаются через символ `|`, внутри разделителя `<%=: ... %>`.

  Чтобы вывести `item` с большой буквы, можно вместо `<%=item%>` написать `<%=: item | capitalize %>`. Чтобы выводить отсортированный массив, можно использовать `<%=: items | sort %>` и так далее.

Свой язык

Для того, чтобы сделать шаблон ещё короче, а также с целью «отвязать» их от JavaScript, ряд шаблонных систем предлагают свой язык.

Например:

- Mustache
- Handlebars
- Closure Templates
- ...тысячи их...

Шаблон для меню в Handlerbars, к примеру, будет выглядеть так:

```
<div class="menu">
  <span class="title">{{title}}</span>
  <ul>
    {{#each items}}
    <li>{{item}}</li>
    {{/each}}
  </ul>
<div>
```

Как видно, *вместо* JavaScript-конструкций здесь используются хелперы. В примере выше `{{#each}} ... {{/each}}` – «блочный» хелпер: он показывает своё содержимое для каждого элемента `items` и является альтернативой `forEach`.

Есть и другие встроенные в шаблонизатор хелперы, можно легко делать свои.

Использование такого шаблона:

```
// текст шаблона должен быть в переменной tmpl
var compiled = Handlebars.compile(tmpl);

var result = compiled({
  title: "Сладости",
  items: ["Торт", "Пирожное", "Пончик"]
});
```

Библиотека шаблонизации Handlebars «понимает» этот язык. Вызов `Handlebars.compile` принимает строку шаблона, разбивает по разделителям и, аналогично предыдущему виду шаблонов, делает JavaScript-функцию, которая затем по данным выдаёт строку-результат.

Запрет на встроенный JS

Если «свой язык шаблонизатора» очень прост, то библиотеку для его поддержки можно легко написать под PHP, Ruby, Java и других языках, которые тем самым научатся понимать такие шаблоны.

Если шаблонка действительна нацелена на кросс-платформенность, то явные JS-вызовы в ней запрещены. Всё делается через хелперы.

Если же нужна какая-то логика, то она либо выносится во внешний код, либо делается через новый хелпер – он отдельно пишется на JavaScript (для клиента) и для сервера (на его языке). Получается полная совместимость.

Это создаёт определённые сложности. Например, в Handlebars есть хелпер `{{#if cond}} ... {{/if}}`, который выводит содержимое, если истинно условие `cond`. При этом вместо `cond` нельзя поставить, к примеру, `a > b` или вызов `str.toUpperCase()`, будет ошибка. Все вычисления должны быть сделаны на этапе передачи данных в шаблон.

Так сделано как раз для переносимости шаблонной системы на другие языки, но на практике не очень-то удобно.

Продвинутые кросс-платформенные шаблонизаторы, в частности, Closure Templates, обладают более мощным языком и умеют самостоятельно разбирать и компилировать многие выражения.

Шаблонизация компонент

До этого мы говорили о шаблонных системах «общего назначения». По большому счёту, это всего лишь механизмы для преобразования одной строки в другую. Но при описании шаблона для компоненты мы хотим сгенерировать не просто строку, а DOM-элемент, и не просто генерировать, а в дальнейшем – с ним работать.

Современные шаблонные системы «заточены» на это. Они умеют создавать по шаблону DOM-элементы и автоматически выполнять после этого разные полезные действия.

Например:

- Можно сохранить важные подэлементы в свойства компоненты, чтобы было проще к ним обращаться из JavaScript.
- Можно автоматически назначать обработчики из методов компонента.
- Можно запомнить, какие данные относятся к каким элементам и в дальнейшем, при изменении данных автоматически обновлять DOM («привязка данных» – англ. data binding).

Одной из первых систем шаблонизации, которая поддерживает подобные возможности была Knockout.JS.

Попробуйте поменять значение `<input>` в примере ниже и вы увидите двухстороннюю привязку данных в действии:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.3.0/knockout-min.js"></script>

Поменяйте имя: <input data-bind="value: name, valueUpdate: 'input'">
<hr>
<h1>Привет, <span data-bind="text: name"></span>!</h1>

<script>
var user = {
  name: ko.observable("Вася")
};

ko.applyBindings(user, document.body);
</script>
```

Поменяйте имя: Вася

Привет, Вася!

Библиотека Knockout.JS создаёт объект `ko`, который и содержит все её возможности.

В этом примере работу начинает вызов `ko.applyBindings(user, document.body)`.

Его аргументы:

- `user` – объект с данными.
- `document.body` – DOM-элемент, который будет использован в качестве шаблона.

Он пробегает по всем подэлементам `document.body` и, если видит атрибут `data-bind`, то читает его и выполняет привязку данных.

Значение `<input data-bind="value: name, ...">` означает, что нужно привязать `input.value` к свойству `name` объекта данных.

Привязка осуществляется в две стороны:

1. Во-первых, библиотека ставит на `input` свой обработчик `oninput` (можно выбрать другие события, см. документацию), который будет обновлять `user.name`. То есть, изменение `input` автоматически меняет `user.name`
2. Во-вторых, свойство `user.name` создано как `ko.observable(...)`. Технически, `ko.observable(value)` – это функция-обёртка вокруг значения: геттер-сеттер, который умеет рассылать события при изменении.

   Например:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.3.0/knockout-min.js"></script>

<script>
var user = ko.observable("Вася");

// вызов user() возвращает значение
alert( user() ); // Вася

// вызов user.subscribe(func) ставит обработчик на изменение значения
user.subscribe(function(newValue) {
  alert("Новое значение: " + newValue);
});

// вызов user(newValue) меняет значение
user("Петя"); // сработает обработчик, назначенный выше
</script>
```

   Библиотека Knockout.JS ставит свой обработчик на изменение значения и при этом обновляет все привязки. Так что при изменении `user.name` меняется и `input.value`.

Далее в том же примере находится `<span data-bind="text: name">` – здесь атрибут означает привязку текста к `name`. Так как `<span>` по своей инициативе меняться не может, то привязка односторонняя, но если бы мог, то можно сделать и двухстороннюю, это несложно.

Вызов **`ko.applyBindings`** можно делать внутри компоненты, и таким образом устанавливать соответствия между её объектом и DOM.

Библиотека также поддерживает хранение шаблонов в `<script type="text/template">` – см. документацию template-binding, можно организовать прекомпиляцию, добавлять свои привязки и так далее.

Другие библиотеки

Есть другие библиотеки «продвинутой шаблонизации», которые добавляют свои возможности по работе с DOM, например:

- Ractive.JS
- Rivets.JS

Подобная шаблонная система является частью многих фреймворков, например:

- React.JS
- Angular.JS
- Ember.JS

Все эти фреймворки разные:

- Ember использует надстройку над Handlebars.
- React использует JSX (JavaScript XML syntax transform) – свой особый способ вставки разметки в JS-код, который нужно обязательно прекомпилировать перед запуском.
- Angular вместо работы со строками использует клонирование DOM-узлов.

При разработке современного веб-приложения имеет смысл выбрать продвинутую шаблонную систему или даже один из этих архитектурных фреймворков.

Итого

Системы шаблонизации, в порядке развития и усложнения:

- Микрошаблонизация – строка с JS-вставками, которая компилируется в функцию – самый простой вариант, минимальная работа для шаблонизатора.
- Собственный язык шаблонов – «особо простой» синтаксис для частых операций, с запретом на JS в случае, если нужна кросс-платформенность.
- Шаблонизация для компонентов – современные системы, которые умеют не только генерировать DOM, но и помогать с дальнейшей работой с ним.

Для того, чтобы использовать одни и те же шаблоны на клиенте и сервере, применяют либо кросс-платформенную систему шаблонизации, либо, чаще – интегрируют серверную часть с V8 и, возможно, с сервером Node.JS.

В главе было много ссылок на шаблонные системы. Все они достаточно современные, поддерживаемые и используются во многих проектах. Среди них вы наверняка найдёте нужную вам.

Книги по JS, HTML/CSS и не только

Мне часто задают вопрос: «Какую литературу порекомендуете?». На этой странице я предлагаю рекомендации по различным темам. Всего несколько книг на каждую тему, из большего количества все равно пришлось бы выбирать.

Кстати, по всем книжкам, особенно тем, которые касаются технологий, всегда ищите последнее издание.

P.S. Скачать книги здесь нельзя. Эта страница содержит только рекомендации.

CSS

CSS стоит изучать по одной из этих книг. Можно сразу по обеим.

- Большая книга CSS3. *Дэвид Макфарланд.*
- CSS. Каскадные таблицы стилей. Подробное руководство. *Эрик Мейер*

Конечно, стандарты тоже будут полезны. Подчас их точность куда проще, чем много страниц разъяснений.

JavaScript

Полезное чтение о языке, встроенных методах и конструкциях JavaScript:

- JavaScript. Подробное руководство. *Дэвид Флэнаган.*
- JavaScript. Шаблоны. *Стоян Стефанов.*

jQuery

Кроме документации:

- jQuery. Подробное руководство по продвинутому JavaScript. *Бер Бибо, Иегуда Кац.*

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) – это концепция построения программных систем на основе объектов и взаимодействия между ними. При изучении ООП рассматриваются полезные архитектурные приёмы, как организовать программу более эффективно.

Умение создавать объект, конструктор, вызывать методы – это основные, самые базовые «кирпичики». Их следует освоить первыми, например используя этот учебник. Затем, когда основы более-менее освоены, стоит уделить внимание теории объектно-ориентированной разработки:

- Объектно-ориентированный анализ и проектирование с примерами приложений. *Гради Буч и др.*.
- Приемы объектно-ориентированного проектирования. Паттерны проектирования. *Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.*

Регулярные выражения

- Регулярные выражения. *Джеффри Фридл.*

Эта книга описывает более широкий класс регэкспов, по сравнению с текущим JavaScript. С одной стороны, какая-то информация будет лишней, с другой – регулярные выражения вообще очень важная и полезная тема.

Алгоритмы и структуры данных

- Алгоритмы. Построение и анализ. *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.*

Есть и другая классика, например «Искусство программирования», Дональд Кнут, но она требует более серьёзной математической подготовки. Будьте готовы читать и вникать долго и упорно. Результат – апгрейд мозговых извилин и общего умения программировать.

Разработка и организация кода

- Совершенный код. *Стив Макконнелл.*

Это желательно изучать уже после получения какого-то опыта в программировании.

Асинхронное выполнение: setImmediate

Функция, отложенная через `setTimeout(..0)` выполнится не ранее следующего «тика» таймера, минимальная частота которого может составлять от 4 до 1000 мс. И, конечно же, это произойдёт после того, как все текущие изменения будут перерисованы.

Но нужна ли нам эта дополнительная задержка? Как правило, используя `setTimeout(func, 0)`, мы хотим перенести выполнение `func` на «ближайшее время после текущего кода», и какая-то дополнительная задержка нам не нужна. Если бы была нужна – мы бы её указали вторым аргументом вместо `0`.

Метод setImmediate(func)

Для того, чтобы поставить функцию в очередь на выполнение без задержки, в Microsoft предложили метод setImmediate(func). Он реализован в IE10+ и на платформе Node.JS.

У `setImmediate` единственный аргумент – это функция, выполнение которой нужно запланировать.

В других браузерах `setImmediate` нет, но его можно эмулировать, используя, к примеру, метод postMessage, предназначенный для пересылки сообщений от одного окна другому. Детали работы с `postMessage` вы найдёте в статье Общение окон с разных доменов: postMessage. Желательно читать её после освоения темы «События».

Полифилл для `setImmediate` через `postMessage`:

```
if (!window.setImmediate) window.setImmediate = (function() {
  var head = { }, tail = head; // очередь вызовов, 1-связный список

  var ID = Math.random(); // уникальный идентификатор

  function onmessage(e) {
    if(e.data != ID) return; // не наше сообщение
    head = head.next;
    var func = head.func;
    delete head.func;
    func();
  }

  if(window.addEventListener) { // IE9+, другие браузеры
    window.addEventListener('message', onmessage);
  } else { // IE8
    window.attachEvent( 'onmessage', onmessage );
  }

  return function(func) {
    tail = tail.next = { func: func };
    window.postMessage(ID, "*");
  };
}());
```

Есть и более сложные эмуляции, включая MessageChannel для работы с Web Workers и хитрый метод для поддержки IE8-: https://github.com/NobleJS/setImmediate. Все они по существу являются «хаками», направленными на то, чтобы обеспечить поддержку `setImmediate` в тех браузерах, где его нет.

Тест производительности

Чтобы сравнить реальную частоту срабатывания – измерим время на 100 последовательных вызовов при `setTimeout(..0)` по сравнению с `setImmediate`:

Запустите пример выше – и вы увидите реальную разницу во времени между `setTimeout(.., 0)` и `setImmediate`. Да, она может быть более в 50, 100 и более раз.

Позднее связывание "bindLate"

Обычный метод `bind` называется «ранним связыванием», поскольку фиксирует привязку сразу же.

Как только значения привязаны – они уже не могут быть изменены. В том числе, если метод объекта, который привязали, кто-то переопределит – «привязанная» функция этого не заметит.

Позднее связывание – более гибкое, оно позволяет переопределить привязанный метод когда угодно.

Раннее связывание

Например, попытаемся переопределить метод при раннем связывании:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}

var user = {
  sayHi: function() { alert('Привет!'); }
}

// привязали метод к объекту
var userSayHi = bind(user.sayHi, user);
```

```
// понадобилось переопределить метод
user.sayHi = function() { alert('Новый метод!'); }

// будет вызван старый метод, а хотелось бы - новый!
userSayHi(); // выведет "Привет!"
```

...Привязка всё ещё работает со старым методом, несмотря на то что он был переопределён.

Позднее связывание

При позднем связывании `bind` вызовет не ту функцию, которая была в `sayHi` на момент привязки, а ту, которая есть на момент вызова.**

Встроенного метода для этого нет, поэтому нужно реализовать.

Синтаксис будет таков:

```
var func = bindLate(obj, "method");
```

`obj`

Объект

`method`

Название метода (строка)

Код:

```
function bindLate(context, funcName) {
  return function() {
    return context[funcName].apply(context, arguments);
  };
}
```

Этот вызов похож на обычный `bind`, один из вариантов которого как раз и выглядит как `bind(obj, "method")`, но работает по-другому.

Поиск метода в объекте: **`context[funcName]`**, осуществляется при вызове, самой обёрткой.

Поэтому, если метод переопределили – будет использован всегда последний вариант.

В частности, пример, рассмотренный выше, станет работать правильно:

```
function bindLate(context, funcName) {
  return function() {
    return context[funcName].apply(context, arguments);
  };
}

var user = {
  sayHi: function() { alert('Привет!'); }
}

var userSayHi = bindLate(user, 'sayHi');

user.sayHi = function() { alert('Здравствуйте!'); }

userSayHi(); // Здравствуйте!
```

Привязка метода, которого нет

Позднее связывание позволяет привязать к объекту даже метод, которого ещё нет!

Конечно, предполагается, что к моменту вызова он уже будет определён ;).

Например:

```
function bindLate(context, funcName) {
  return function() {
    return context[funcName].apply(context, arguments);
  };
}

// метода нет
var user = {  };

// ..а привязка возможна!
var userSayHi = bindLate(user, 'sayHi');

// по ходу выполнения добавили метод..
user.sayHi = function() { alert('Привет!'); }

userSayHi(); // Метод работает: Привет!
```

В некотором смысле, позднее связывание всегда лучше, чем раннее. Оно удобнее и надежнее, так как всегда вызывает нужный метод, который в объекте сейчас.

Но оно влечет и небольшие накладные расходы – поиск метода при каждом вызове.

Итого

Позднее связывание ищет функцию в объекте в момент вызова.

Оно используется для привязки в тех случаях, когда метод *может быть переопределён* после привязки или *на момент привязки не существует*.

Обёртка для позднего связывания (без карринга):

```
function bindLate(context, funcName) {
  return function() {
    return context[funcName].apply(context, arguments);
  };
}
```

Sublime Text: шпаргалка

Одним из наиболее мощных, и при этом простых, редакторов является Sublime Text.

В нём хорошо развита система команд и горячих клавиш. На этой странице размещена «шпаргалка» с плагинами и самыми частыми сочетаниями клавиш, которые сильно упрощают жизнь.

Горячие клавиши

Для наибольшего удобства «шпаргалка» должна быть распечатана и повешена перед глазами, поэтому она сделана в виде 3-колоночного PDF.

Скачать шпаргалку в формате PDF

Эта шпаргалка – под Mac, для Windows сочетания похожи, обычно вместо Mac-клавиши `Cmd` под Windows будет `Ctrl`. А если в сочетании есть и `Cmd` и `Ctrl`, то под Windows будет `Ctrl+Shift`.

Вы часто используете сочетание, но его нет в списке? Поделитесь им в комментариях!

Плагины

Мои любимые плагины:

- Package Control
- sublime-emmet
- JsFormat
- SideBarEnhancements
- AdvancedNewFile
- sublime-jsdocs
- SublimeCodeIntel

Остальные:

- Alignment
- CSSComb
- EncodingHelper
- GoToRecent
- HTML5
- jQuery
- Prefixr
- View In Browser

Чтобы узнать о плагине подробнее – введите его название в Google.

Есть и другие хорошие плагины, кроме перечисленных. Кроме того, Sublime позволяет легко писать свои плагины и команды.

Выделение: Range, TextRange и Selection

В этой статье речь пойдет о документированных, но нечасто используемых объектах `Range`, `TextRange` и `Selection`. Мы рассмотрим вольный перевод спецификаций с понятными примерами и различные кроссбраузерные реализации.

Эта статья представляет собой обновлённый вариант статьи Александра Бурцева, которой уже нет онлайн. Публикуется с его разрешения, спасибо, Александр!

Range

`Range`. – это объект, соответствующий фрагменту документа, который может включать узлы и участки текста из этого документа. Наиболее подробно объект `Range` описан в спецификации DOM Range.

Чтобы понять о чем речь, обратимся к самому простому случаю `Range`, который будет подробно рассмотрен ниже – к выделениям. В приводимом ниже примере выделите несколько слов в предложении. Будет выводиться текстовое содержимое выделяемой области:

Соберем микс из жирности, *курсива* и ссылки и повыделяем здесь.

Но такие области можно создавать не только с помощью пользовательского выделения, но и из JavaScript-сценария, выполняя с ними определенные манипуляции. Однако, написать простой иллюстрирующий код сразу не выйдет, т.к. есть одно НО – Internet Explorer до версии 9. В Microsoft создали собственную реализацию – объект TextRange. Разберём каждую реализацию по-отдельности.

DOM-реализация Range (кроме IE8-)

`Range` состоит из двух граничных точек (boundary-points), соответствующих началу и концу области. Позиция любой граничной точки определяется в документе с помощью двух свойств: узел (node) и смещение (offset).

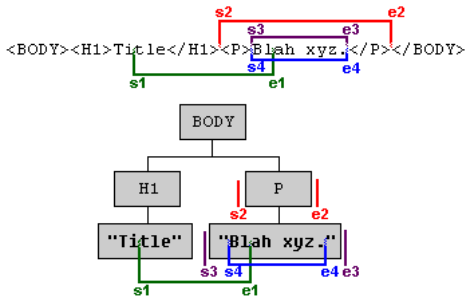
Контейнером (container) называют узел, содержащий граничную точку. Сам контейнер и все его предки называются родительскими контейнерами (ancestor containers) для граничной точки. Родительский контейнер, включающий обе граничные точки, называют корневым контейнером (root container).

```
<body><h1>Title</h1><p>Blah xyz</p></body>
```

На изображении выше граничные точки выделения лежат в текстовых узлах (`#text1` и `#text2`), которые являются контейнерами. Для левой границы родительскими контейнерами являются `#text1` , `H1` , `BODY` , для правой – `#text2` , `P` , `BODY` . Общий родитель для обоих граничных точек – `BODY` , этот элемент является корневым контейнером.

Если контейнер является текстовым узлом, то смещение определяется в символах от начала DOM-узла. Если контейнер является элементом (`Document` , `DocumentFragment` , `Element` ...), то смещение определяется в дочерних узлах.

Смотрим на иллюстрацию (источник):



Range	s		e	
	Node	Offset	Node	Offset
s1—e1	Text1	2	Text2	2
s2—e2	BODY	1	BODY	2
s3—e3	P	0	P	1
s4—e4	Text2	0	Text2	9

Граничные точки объекта `Range` s1 лежат в текстовых узлах, поэтому смещение задается в символах от начала узла. Для s2 граничные точки расставлены так, что включают весь абзац Blah xyz , поэтому контейнером является элемент `BODY` , и смещение считается в позициях дочерних узлов.

Объекты `Range` создаются с помощью вызова `document.createRange()` . Объект при этом создается пустой, и граничные точки нужно задать далее его методами `setStart` и `setEnd` . Смотрим пример.

HTML:

```
<div id="ex2">
  <h2>Соз|даем объект Range</h2>
  <p>От третье|го символа заголовка до десятого символа это абзаца.</p>
</div>

<button onclick="alert(domRangeCreate())">
  Создать Range и вывести его текст
</button>

<script>
  function domRangeCreate() {
    // Найдем корневой контейнер
    var root = document.getElementById('ex2');
    // Найдем контейнеры граничных точек (в данном случае тестовые)
    var start = root.getElementsByTagName('h2')[0].firstChild;
    var end = root.getElementsByTagName('p')[0].firstChild;
    if (root.createRange) {
      // Создаем Range
      var rng = root.createRange();
      // Задаем верхнюю граничную точку, передав контейнер и смещение
      rng.setStart(start, 3);
      // Аналогично для нижней границы
      rng.setEnd(end, 10);
      // Теперь мы можем вернуть текст, который содержится в полученной области
      return rng.toString();
    } else {
      return 'Вероятно, у вас IE8-, смотрите реализацию TextRange ниже';
    }
  }
</script>
```

В действии:

Соз|даем Range-объект

От третье|го символа заголовка до десятого символа это абзаца.

Создать Range и вывести его текст

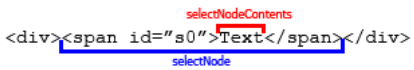
Рассмотрим вкратце свойства и методы Range:

- Свойство `commonAncestorContainer` вернет ссылку на наиболее вложенный корневой контейнер.
- Свойство `startContainer` (`endContainer`) вернет ссылку на контейнер верхней (нижней) граничной точки.
- Свойство `startOffset` (`endOffset`) вернет смещение для верхней (нижней) граничной точки.
- Свойство `collapsed` вернет `true` , если граничные точки имеют одинаковые контейнеры и смещение (`false` в противном случае).
- Метод `setStart` (`setEnd`) задает контейнер (ссылка на узел) и смещение (целочисленное значение) для соответствующих граничных точек. Пример выше.
- Методы `setStartBefore` , `setStartAfter` , `setEndBefore` , `setEndAfter` принимают в качестве единственного аргумента ссылку на узел и устанавливают граничные точки в соот-ии с естественной границей переданного узла. Например:

```
<span id="s1">First</span>
<span id="s2">Second</span>
```

```
var rng = document.createRange();
// Установит верхнюю граничную точку по левой границе спана #s1
rng.setStartBefore(document.getElementById('s1'));
// Установит нижнюю граничную точку по правой границе спана #s2
rng.setEndAfter(document.getElementById('s2'));
```

- Методы `selectNode` и `selectNodeContents` позволяют создать объект `Range` по границам узла, ссылку на который они принимают в качестве единственного аргумента. При использовании `selectNode` передаваемый узел также войдет в `Range` , в то время как `selectNodeContents` создаст объект только из содержимого узла:



- Метод `collapse` объединяет граничные точки объекта `Range` . В качестве единственного аргумента принимает булево значение (`true` – для объединения в верхней точке, `false` – в нижней). По-умолчанию `true` .
- Метод `toString` вернет текстовое содержимое объекта `Range` .
- Метод `cloneContents` вернет копию содержимого объекта `Range` в виде фрагмента документа.
- Метод `cloneRange` вернет копию самого объекта `Range` .
- Метод `deleteContents` удаляет всё содержимое объекта `Range` .
- Метод `detach` извлекает текущий объект из DOM, так что на него больше нельзя сослаться.
- Метод `insertNode` принимает в качестве единственного аргумента ссылку на узел (или фрагмент документа) и вставляет его в содержимое объекта `Range` в начальной точке.
- Метод `extractContents` вырезает содержимое объекта `Range` и возвращает ссылку на полученный фрагмент документа.
- Метод `surroundContents` помещает всё содержимое текущего объекта `Range` в новый родительский элемент, ссылка на который принимается в качестве единственного аргумента.
- Метод `compareBoundaryPoints` используется для сравнения граничных точек.

Для примера решим небольшую задачку. Найдём в текстовом узле фразу и подсветим её синим фоном.

```
<div id="ex3">
  Найдем в этом тексте слово "бабуля" и подсветим его синим фоном
</div>

<script>
  function domRangeHighlight(text) {
    // Получим текстовый узел
    var root = document.getElementById('ex3').firstChild;
    // и его содержимое
```

```
    var content = root.nodeValue;
    // Проверим есть ли совпадения с переданным текстом
    if (~content.indexOf(text)) {
      if (document.createRange) {
        // Если есть совпадение, и браузер поддерживает Range, создаем объект
        var rng = document.createRange();
        // Ставим верхнюю границу по индексу совпадения,
        rng.setStart(root, content.indexOf(text));
        // а нижнюю по индексу + длина текста
        rng.setEnd(root, content.indexOf(text) + text.length);
        // Создаем спан с синим фоном
        var highlightDiv = document.createElement('span');
        highlightDiv.style.backgroundColor = 'blue';
        // Обернем наш Range в спан
        rng.surroundContents(highlightDiv);
      } else {
        alert( 'Вероятно, у вас IE8-, смотрите реализацию TextRange ниже' );
      }
    } else {
      alert( 'Совпадений не найдено' );
    }
  }
</script>
```

В действии:

С остальными свойствами и методами поэкспериментируйте сами. Перейдем к реализации range в IE.

TextRange (для IE)

Объект `TextRange` в реализации MSIE – это текстовый диапазон нулевой и более длины. У данного диапазона также есть свои границы, «перемещать» которые можно на целое число текстовых единиц: character(символ), word (слово), sentence (предложение). То есть можно взять и сдвинуть границу на 2(5, 8 и т.д.) слова (символа, предложения) вправо (влево). При этом у объекта сохраняются данные о HTML-содержимом диапазона и есть методы взаимодействия с DOM.

Объект `TextRange` создается с помощью метода `createTextRange`, который можно вызывать в контексте элементов `BODY`, `BUTTON`, `INPUT` (большинство типов), `TEXTAREA`.

Простой пример с кнопкой:

```
<input id="buttonId" type="button" value="Test button" onclick="alert( ieTextRangeCreate() );" />

<script>
  function ieTextRangeCreate() {
    // Найдем кнопку
    var button = document.getElementById('buttonId');
    // Если мы в ИЕ
    if (button.createTextRange && button.createTextRange() != undefined) {
      // Создаем TextRange
      var rng = button.createTextRange();
      // И вернем текстовое содержимое полученного объекта
      return rng.text;
    } else {
      return 'Вероятно, у вас не IE, смотрите реализацию Range выше';
    }
  }
</script>
```

Test button

Рассмотрим свойства и методы объекта TextRange (не все, только самые необходимые):

- Свойство `boundingWidth` (boundingHeight) вернет ширину (высоту), которую занимает объект TextRange в пикселях.
- Свойство `boundingTop` (`boundingLeft`) вернет Y(X)-координату верхнего левого угла тестовой области относительно окна документа.
- Свойство `htmlText` вернет HTML-содержимое объекта.
- Свойство `text` вернет текстовое содержимое объекта (см. пример выше).
- Свойство `offsetTop` (`offsetLeft`) вернет Y(X)-координату верхнего левого угла тестовой области относительно предка.
- Метод `collapse` объединяет граничные точки диапазона. В качестве единственного аргумента принимает булево значение (`true` – для объединения в верхней точке, `false` – в нижней). По-умолчанию true.
- Метод `duplicate` клонирует имеющийся текстовый диапазон, возвращая новый, точно такой же.
- Метод `expand` расширяет текущий тектовый диапазон до единицы текста, переданной в качестве единственного текстового аргумента:
  - `"character"` – символ.
  - `"word"` – слово
  - `"sentence"` – предложение
  - `"textedit"` – сворачивает до первоначального диапазона.

  Вернет `true` (`false`) в случае успеха (неудачи).
- Метод `findText` ищет в диапазоне совпадения с текстовой строкой, передаваемой в качестве первого аргумента (без учета регистра). Если совпадение найдено, то границы диапазона сворачиваются до него. В качестве второго (необязательного) аргумента можно передать целое число, указывающее число символов от верхней точки, в которых нужно производить поиск. Далее в качестве аргументов можно перечислять INT-флаги, которые вам вряд ли понадобятся.

- Метод `getBookmark` возвращает в случае успешного вызова строку, по которой можно будет восстановить текущее состояние текстового диапазона с помощью метода `moveToBookmark`.
- Метод `inRange` принимает в качестве аргумента другой `TextRange` и проверяет, входит ли его текстовый диапазон в диапазон контекстного объекта. Возвращает булево значение.
- Метод `isEqual` проверяет является ли текущий `TextRange` идентичным переданному в качестве аргумента. Возвращает булево значение.
- Метод `move(sUnit [, iCount])` сворачивает текущий диапазон до нулевой длины и передвигает на единицу текста, переданного в качестве первого аргумента (character | word | sentence | textedit). В качестве второго (необязательного) аргумента можно передать число единиц, на которое следует передвинуть диапазон.
- Метод `moveEnd` (`moveStart`), аналогично методу move, передвигает верхнюю (нижнюю) границу диапазона на единицу текста, число которых также можно задать необязательным вторым параметром.
- Метод `moveToElementText` принимает в качестве аргумента ссылку на DOM-элемент и выставляет границы диапазона Textобъекта `Range` по границам полученного элемента.
- Метод `moveToPoint` принимает в качестве двух обязательных аргументов X и Y-координаты (в пикселях) относительно верхнего левого угла документа и переносит границы диапазона туда.
- Метод `parentElement` вернет ссылку на элемент, который полностью содержит диапазон объекта `TextRange` (или `null`).
- Метод `pasteHTML` заменяет HTML-содержимое текущего текстового диапазона на строку, переданную в качестве единственного аргумента.
- Метод `select` формирует выделение на основе содержимого объекта `TextRange`, о чем мы подробнее поговорим ниже.
- Метод `setEndPoint` принимает в качестве обязательных аргументов текстовый указатель и ссылку на другой `TextRange`, устанавливая в зависимости от значения указателя границы диапазона. Указатели могут быть следующими: „StartToEnd“, „StartToStart“, „EndToStart“, „EndToEnd“.

Также к `TextRange` применимы команды метода execCommand, который умеет делать текст жирным, курсивным, копировать его в буфер обмена (только IE) и т.п.

Для закрепления сделаем задачку по поиску текстового содержимого, аналогичную той, что была выше:

```
<div id="ex4">
  Найдем в этом тексте слово "бабуля" и подсветим его синим фоном
</div>

<script>
  function ieTextRangeHighlight(text) {
    // Получим ссылку на элемент, в котором будет происходить поиск
    var root = document.getElementById('ex4');
    // Получим значение его текстового потомка
    var content = root.firstChild.nodeValue;
    // Если есть совпадение
    if (~content.indexOf(text)) {
      // и мы в MSIE
      if (document.body.createTextRange) {
        // Создадим объект TextRange
        var rng = document.body.createTextRange();
        // Свернем его до root
        rng.moveToElementText(root);
        // Найдем текст и свернем диапазон до него
        if (rng.findText(text))
        // Заменим текстовый фрагмент на span с синим фоном
          rng.pasteHTML('<span style="background:blue;">' + text + '</span>');
      } else {
        alert( 'Вероятно, у вас не  IE, смотрите реализацию Range выше' );
      }
    } else {
      alert( 'Совпадений не найдено' );
    }
  }
</script>
```

В действии:

С остальными свойствами и методами поэкспериментируйте сами.

Selection

Всем знакомо выделение элементов на странице, когда, зажав левую кнопку мыши и передвигая курсор, мы выделяем нужный фрагмент. Или зажимаем Shift и жмём на стрелочки клавиатуры. Или еще как-то, неважно. В данной части статьи мы кроссбраузерно научимся решать две задачи: получать пользовательское выделение и устанавливать собственное.

Получаем пользовательское выделение

Эту задачу мы уже решали в самом начале статьи в примере с миксом. Теперь рассмотрим код:

```
function getSelectionText() {
  var txt = '';
  if (txt = window.getSelection) // Не IE, используем метод getSelection
    txt = window.getSelection().toString();
  } else { // IE, используем объект selection
    txt = document.selection.createRange().text;
  }
  return txt;
}
```

Все браузеры, кроме IE8- поддерживают метод `window.getSelection()`, который возвращает объект, схожий с рассмотренным ранее `Range`. У этого объекта есть точка начала выделения (anchor) и фокусная точка окончания (focus). Точки могут совпадать. Рассмотрим свойства и методы объекта `Selection`:

- Свойство `anchorNode` вернет контейнер, в котором начинается выделение. Замечу, что началом выделения считается та граница, от которой вы начали выделение. То есть, если вы выделяете справа налево, то началом будет именно правая граница. Это правило работает везде, кроме браузера Opera, в котором `anchorNode` вернет ссылку на узел левого края выделения.
- Свойство `anchorOffset` вернет смещение для начала выделения в пределах контейнера `anchorNode`.
- Свойства `focusNode` и `focusOffset` работают аналогично для фокусных точек, то есть точек окончания выделения. Opera и здесь отличилась, возвращает вместо фокусной точки узел правого края выделения.
- Свойство `rangeCount` возвращает число объектов `Range`, которые входят в полученное выделение. Это свойство полезно при использовании метода `addRange`.
- Метод `getRangeAt` принимает в качестве аргумента индекс объекта `Range` и возвращает сам объект. Если `rangeCount == 1`, то работать будет только `getRangeAt(0)`. Таким образом, мы можем получить объект `Range`, полностью соответствующий текущему выделению.
- Метод `collapse` сворачивает выделение в точку (каретку). Методу можно передать в качестве первого аргумента узел, в который нужно поместить каретку.
- Метод `extend` принимает в качестве аргументов ссылку на контейнер и смещение (`parentNode`, `offset`), и перемещает фокусную точку в это положение.
- Метод `collapseToStart` (`collapseToEnd`) перемещает фокусную (начальную) границу к начальной (фокусной), тем самым сворачивая выделение в каретку.
- Метод `selectAllChildren` принимает в качестве единственного аргумента ссылку на узел и добавляет всех его потомков в выделение.
- Метод `addRange` принимает в качестве аргумента объект `Range` и добавляет его в выделение. Таким образом можно увеличить количество объектов `Range`, число которых нам подскажет свойство `rangeCount`.
- Метод `removeRange` (`removeAllRanges`) удаляет переданный (все) объект `Range` из выделения.
- Метод `toString` вернет текстовое содержимое выделения.

IE предоставляет собственный интерфейс взаимодействия с выделениями – объект `selection` в контексте document. Для работы с этим объектом используются следующие методы:

- Метод `clear` убирает выделение вместе с содержимым.
- Метод `createRange` (ВАЖНО! Не путать со стандартным методом `document.createRange()` для создания объектов `Range`!) создает из содержимого выделения `TextRange`.
- Метод `empty` убирает выделение, но оставляет содержимое.

Надеюсь, теперь, после знакомства с обеими реализациями выделений, код выше стал более понятен.

Установка собственного выделения

Допустим, вам хочется, чтобы какой-то текстовый фрагмент на странице был выделен, как пользовательское выделение. Это нужно при клиентской реализации поиска по странице и некоторых других задач.

Проще всего решить эту задачу следующим образом:

1. Создать объект `Range` (`TextRange` для IE8-).
2. Перевести полученный объект в выделение.

Смотрим реализацию:

```
<div id="ex5">
  Снова будем выделять <span>бабулю</span>, на этот раз без поиска.
</div>

<script>
  function setSelection() {
    var target = document.getElementById('ex5').getElementsByTagName('span')[0];
    var rng, sel;
    if (document.createRange) {
      rng = document.createRange();
      rng.selectNode(target)
      sel = window.getSelection();
      sel.removeAllRanges();
      sel.addRange(rng);
    } else {
      var rng = document.body.createTextRange();
      rng.moveToElementText(target);
      rng.select();
    }
  }
</script>
```

В действии:

Снятие выделения

Код для снятия выделения, использующий соответствующие методы объектов `Selection`:

```
function clearSelection() {
  try {
    // современный объект Selection
    window.getSelection().removeAllRanges();
  } catch (e) {
    // для IE8-
    document.selection.empty();
  }
}
```

Итого

- В современных браузерах поддерживается стандартный объект Range
- В IE8- поддерживается только собственный объект TextRange.

Есть библиотеки, которые «исправляют» объект `TextRange`, добавляя ему нужные свойства из `Range`.

Код, получающий выделение, при использовании такой библиотеки может выглядеть так:

```
var range = getRangeObject();
  if (range) {
    alert( range );
    alert( range.startContainer.nodeValue );
    alert( range.startOffset );
    alert( range.endOffset );
  } else {
    alert( 'Ничего не выделено' );
  }
  }
```

В действии:

Код функций `getRangeObject(win)` для получения выделения в окне и `fixIERangeObject(range, win)` для исправления `TextRange` – в песочнице вместе с этим примером.

Применяем ООП: Drag'n'Drop++

Эта статья представляет собой продолжение главы Мышь: Drag'n'Drop более глубоко. Она посвящена более гибкой и расширяемой реализации переноса.

Рекомендуется прочитать указанную главу перед тем, как двигаться дальше.

В сложных приложениях Drag’n’Drop обладает рядом особенностей:

1. Перетаскиваются *элементы* из *зоны переноса* `dragZone` в *зону-цель* `dropTarget`. При этом сама зона не переносится.

   Например – два списка, нужен перенос элемента из одного в другой. В этом случае один список является зоной переноса, второй – зоной-целью.

   Возможно, что перенос осуществляется внутри одного и того же списка. При этом `dragZone == dropTarget`.
2. На странице может быть несколько разных зон переноса и зон-целей.
3. Обработка завершения переноса может быть асинхронной, с уведомлением сервера.
4. Должно быть легко добавить новый тип зоны переноса или зоны-цели, а также расширить поведение существующей.
5. Фреймворк для переноса должен быть расширяемым с учётом сложных сценариев.

Всё это вполне реализуемо. Но для этого фреймворк, описанный в статье Мышь: Drag'n'Drop более глубоко, нужно отрефакторить, и разделить на сущности.

Основные сущности

Всего будет 4 сущности:

`DragZone`

Зона переноса. С нее начинается перенос. Она принимает нажатие мыши и генерирует аватар нужного типа.

`DragAvatar`

Переносимый объект. Предоставляет доступ к информации о том, что переносится. Умеет двигать себя по экрану. В зависимости от вида переноса, может что-то делать с собой в конце, например, самоуничтожаться.

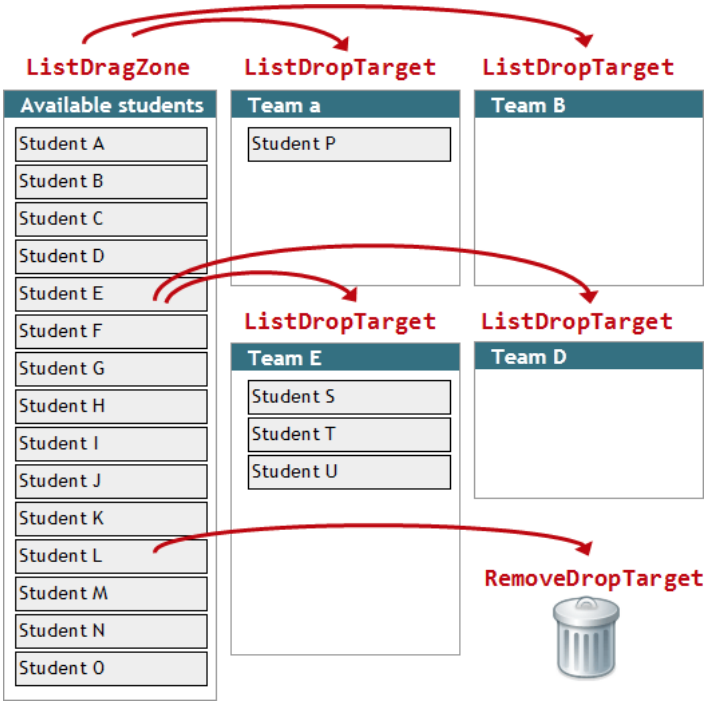
`DropTarget`

Зона-цель, на которую можно положить. В процессе переноса аватара над ней умеет рисовать на себе предполагаемое «место приземления». Обрабатывает окончание переноса.

`dragManager`

Единый объект, который стоит над всеми ними, ставит обработчики `mousedown/mousemove/mouseup` и управляет процессом. В терминах ООП, это не класс, а объект-синглтон, поэтому он с маленькой буквы.

На макете страницы ниже возможен перенос студентов из левого списка – вправо, в одну из команд или в «корзину»:



Здесь левый список является зоной переноса `ListDragZone`, а правые списки – это несколько зон-целей `ListDropTarget`. Кроме того, корзина также является зоной-целью отдельного типа `RemoveDropTarget`.

Пример

В этой статье мы реализуем пример, когда узлы дерева можно переносить внутри него. То есть, дерево, которое является одновременно `TreeDragZone` и `TreeDropTarget`.

Структура дерева будет состоять из вложенных списков с заголовком в `SPAN`:

```
<ul>
  <li><span>Заголовок 1</span>
    <ul>
      <li><span>Заголовок 1.1</span></li>
      <li><span>Заголовок 1.2</span></li>
      ...
    </ul>
  </li>
  ...
</ul>
```

При переносе:

- Для аватара нужно клонировать заголовок узла, на котором было нажатие.
- Узлы, на которые можно положить, при переносе подсвечиваются красным.
- Нельзя перенести узел сам в себя или в своего потомка.
- Дерево само поддерживает сортировку по алфавиту среди узлов.
- Обязательна расширяемость кода, поддержка большого количества узлов и т.п.

Возьмите за любой заголовок и поменяйте ему родителя.
В собственных детей перенести нельзя.
Потомки всегда отсортированы по алфавиту.

- Древо жизни (сверхмалая часть)
  - Грибы
    - Древесные
      - Чага
    - Наземные
      - Опята
      - Подосиновики
  - Животные
    - Земноводные
      - Лягушки
      - Саламандры
      - Тритоны
    - Млекопитающие
      - Коровы
      - Ослы
      - Собаки
      - Тигры

dragManager

Обязанность `dragManager` – обработка событий мыши и координация всех остальных сущностей в процессе переноса.

Готовьтесь, дальше будет много кода с комментариями.

Следующий код должен быть очевиден по смыслу, если вы читали предыдущую статью. Объект взят оттуда, и из него изъята лишняя функциональность, которая перенесена в другие сущности.

Если вызываемые в нём методы `onDrag*` непонятны – смотрите далее, в описание остальных объектов.

```
var dragManager = new function() {

  var dragZone, avatar, dropTarget;
  var downX, downY;

  var self = this;

  function onMouseDown(e) {

    if (e.which != 1) { // не левой кнопкой
      return false;
    }

    dragZone = findDragZone(e);

    if (!dragZone) {
      return;
    }

    // запомним, что элемент нажат на текущих координатах pageX/pageY
    downX = e.pageX;
    downY = e.pageY;

    return false;
  }

  function onMouseMove(e) {
    if (!dragZone) return; // элемент не зажат

    if (!avatar) { // элемент нажат, но пока не начали его двигать
      if (Math.abs(e.pageX - downX) < 3 && Math.abs(e.pageY - downY) < 3) {
        return;
      }
      // попробовать захватить элемент
      avatar = dragZone.onDragStart(downX, downY, e);

      if (!avatar) { // не получилось, значит перенос продолжать нельзя
        cleanUp(); // очистить приватные переменные, связанные с переносом
        return;
      }
    }

    // отобразить перенос объекта, перевычислить текущий элемент под курсором
    avatar.onDragMove(e);

    // найти новый dropTarget под курсором: newDropTarget
    // текущий dropTarget остался от прошлого mousemove
    // *оба значения: и newDropTarget и dropTarget могут быть null
    var newDropTarget = findDropTarget(e);

    if (newDropTarget != dropTarget) {
      // уведомить старую и новую зоны-цели о том, что с них ушли/на них зашли
      dropTarget && dropTarget.onDragLeave(newDropTarget, avatar, e);
      newDropTarget && newDropTarget.onDragEnter(dropTarget, avatar, e);
    }

    dropTarget = newDropTarget;

    dropTarget && dropTarget.onDragMove(avatar, e);

    return false;
  }
```

```
  function onMouseUp(e) {

    if (e.which != 1) { // не левой кнопкой
      return false;
    }

    if (avatar) { // если уже начали передвигать

      if (dropTarget) {
        // завершить перенос и избавиться от аватара, если это нужно
        // эта функция обязана вызвать avatar.onDragEnd/onDragCancel
        dropTarget.onDragEnd(avatar, e);
      } else {
        avatar.onDragCancel();
      }

    }

    cleanUp();
  }

  function cleanUp() {
    // очистить все промежуточные объекты
    dragZone = avatar = dropTarget = null;
  }

  function findDragZone(event) {
    var elem = event.target;
    while (elem != document && !elem.dragZone) {
      elem = elem.parentNode;
    }
    return elem.dragZone;
  }

  function findDropTarget(event) {
    // получить элемент под аватаром
    var elem = avatar.getTargetElem();

    while (elem != document && !elem.dropTarget) {
      elem = elem.parentNode;
    }

    if (!elem.dropTarget) {
      return null;
    }

    return elem.dropTarget;
  }

  document.ondragstart = function() {
    return false;
  }

  document.onmousemove = onMouseMove;
  document.onmouseup = onMouseUp;
  document.onmousedown = onMouseDown;
};
```

DragZone

Основная задача `DragZone` – создать аватар и инициализировать его. В зависимости от места, где произошел клик, аватар получит соответствующий подэлемент зоны.

Метод для создания аватара `_makeAvatar` вынесен отдельно, чтобы его легко можно было переопределить и подставить собственный тип аватара.

```
/**
 * Зона, из которой можно переносить объекты
 * Умеет обрабатывать начало переноса на себе и создавать "аватар"
 * @param elem DOM-элемент, к которому привязана зона
 */
function DragZone(elem) {
  elem.dragZone = this;
  this._elem = elem;
}

/**
 * Создать аватар, соответствующий зоне.
 * У разных зон могут быть разные типы аватаров
 */
DragZone.prototype._makeAvatar = function() {
  /* override */
};

/**
 * Обработать начало переноса.
 *
 * Получает координаты изначального нажатия мышки, событие.
 *
 * @param downX Координата изначального нажатия по X
 * @param downY Координата изначального нажатия по Y
 * @param event текущее событие мыши
 *
 * @return аватар или false, если захватить с данной точки ничего нельзя
 */
DragZone.prototype.onDragStart = function(downX, downY, event) {

  var avatar = this._makeAvatar();

  if (!avatar.initFromEvent(downX, downY, event)) {
    return false;
  }

  return avatar;
};
```

TreeDragZone

Объект зоны переноса для дерева, по существу, не вносит ничего нового, по сравнению с `DragZone`.

Он только переопределяет `_makeAvatar` для создания `TreeDragAvatar`.

```
function TreeDragZone(elem) {
  DragZone.apply(this, arguments);
}

extend(TreeDragZone, DragZone);

TreeDragZone.prototype._makeAvatar = function() {
  return new TreeDragAvatar(this, this._elem);
};
```

DragAvatar

Аватар создается только зоной переноса при начале Drag'n'Drop. Он содержит всю необходимую информацию об объекте, который переносится.

В дальнейшем вся работа происходит *только с аватаром*, сама зона напрямую не вызывается.

У аватара есть три основных свойства:

`_dragZone`

Зона переноса, которая его создала.

`_dragZoneElem`

Элемент, соответствующий аватару в зоне переноса. По умолчанию – DOM-элемент всей зоны. Это подходит в тех случаях, когда зона перетаскивается только целиком. При инициализации аватара значение этого свойства может быть уточнено, например изменено на подэлемент списка, который перетаскивается.

`_elem`

Основной элемент аватара, который будет двигаться по экрану. По умолчанию равен `_dragZoneElem`, т.е мы переносим сам элемент.

При инициализации мы можем также склонировать `_dragZoneElem`, или создать своё красивое представление переносимого элемента и поместить его в `_elem`.

```
/**
 * "Аватар" - элемент, который перетаскивается.
 *
 * В простейшем случае аватаром является сам переносимый элемент
 * Также аватар может быть клонированным элементом
 * Также аватар может быть иконкой и вообще чем угодно.
 */
function DragAvatar(dragZone, dragElem) {
  /** "родительская" зона переноса */
  this._dragZone = dragZone;

  /**
   * подэлемент родительской зоны, к которому относится аватар
   * по умолчанию - элемент, соответствующий всей зоне
   * может быть уточнен в initFromEvent
   */
  this._dragZoneElem = dragElem;

  /**
   * Сам элемент аватара, который будет носиться по экрану.
   * Инициализуется в initFromEvent
   */
  this._elem = dragElem;
}

/**
 * Инициализировать this._elem и позиционировать его
 * При необходимости уточнить this._dragZoneElem
 * @param downX Координата X нажатия мыши
 * @param downY Координата Y нажатия мыши
 * @param event Текущее событие мыши
 */
DragAvatar.prototype.initFromEvent = function(downX, downY, event) {
  /* override */
};

/**
 * Возвращает информацию о переносимом элементе для DropTarget
 * @param event
 */
DragAvatar.prototype.getDragInfo = function(event) {
  // тут может быть еще какая-то информация, необходимая для обработки конца или процесса переноса
  return {
    elem: this._elem,
    dragZoneElem: this._dragZoneElem,
    dragZone: this._dragZone
  };
};

/**
 * Возвращает текущий самый глубокий DOM-элемент под this._elem
 * Приватное свойство _currentTargetElem обновляется при каждом передвижении
 */
DragAvatar.prototype.getTargetElem = function() {
  return this._currentTargetElem;
};

/**
 * При каждом движении мыши перемещает this._elem
```

```
 * и записывает текущий элемент под this._elem в _currentTargetElem
 * @param event
 */
DragAvatar.prototype.onDragMove = function(event) {
  this._elem.style.left = event.pageX - this._shiftX + 'px';
  this._elem.style.top = event.pageY - this._shiftY + 'px';

  this._currentTargetElem = getElementUnderClientXY(this._elem, event.clientX, event.clientY);
};

/**
 * Действия с аватаром, когда перенос не удался
 * Например, можно вернуть элемент обратно или уничтожить
 */
DragAvatar.prototype.onDragCancel = function() {
  /* override */
};

/**
 * Действия с аватаром после успешного переноса
 */
DragAvatar.prototype.onDragEnd = function() {
  /* override */
};
```

TreeDragAvatar

Основные изменения – в методе `initFromEvent`, который создает аватар из узла, на котором был клик.

Обратите внимание, возможно что клик был не на заголовке `SPAN`, а просто где-то на дереве. В этом случае `initFromEvent` возвращает `false` и перенос не начинается.

```
function TreeDragAvatar(dragZone, dragElem) {
  DragAvatar.apply(this, arguments);
}

extend(TreeDragAvatar, DragAvatar);

TreeDragAvatar.prototype.initFromEvent = function(downX, downY, event) {
  if (event.target.tagName != 'SPAN') return false;

  this._dragZoneElem = event.target;
  var elem = this._elem = this._dragZoneElem.cloneNode(true);
  elem.className = 'avatar';

  // создать вспомогательные свойства shiftX/shiftY
  var coords = getCoords(this._dragZoneElem);
  this._shiftX = downX - coords.left;
  this._shiftY = downY - coords.top;

  // инициировать начало переноса
  document.body.appendChild(elem);
  elem.style.zIndex = 9999;
  elem.style.position = 'absolute';

  return true;
};

/**
 * Вспомогательный метод
 */
TreeDragAvatar.prototype._destroy = function() {
  this._elem.parentNode.removeChild(this._elem);
};

/**
 * При любом исходе переноса элемент-клон больше не нужен
 */
TreeDragAvatar.prototype.onDragCancel = function() {
  this._destroy();
};

TreeDragAvatar.prototype.onDragEnd = function() {
  this._destroy();
};
```

DropTarget

Именно на `DropTarget` ложится работа по отображению предполагаемой «точки приземления» аватара, а также, по завершению переноса, обработка результата.

Как правило, `DropTarget` принимает переносимый узел в себя, а вот как конкретно организован процесс вставки – нужно описать в классе-наследнике. Разные типы зон делают разное при вставке: `TreeDropTarget` вставляет элемент в качестве потомка, а `RemoveDropTarget` – удаляет.

```
/**
 * Зона, в которую объекты можно класть
 * Занимается индикацией передвижения по себе, добавлением в себя
 */
function DropTarget(elem) {
  elem.dropTarget = this;
  this._elem = elem;

  /**
   * Подэлемент, над которым в настоящий момент находится аватар
   */
  this._targetElem = null;
}

/**
 * Возвращает DOM-подэлемент, над которым сейчас пролетает аватар
 *
 * @return DOM-элемент, на который можно положить или undefined
```

```
 */
DropTarget.prototype._getTargetElem = function(avatar, event) {
  return this._elem;
};

/**
 * Спрятать индикацию переноса
 * Вызывается, когда аватар уходит с текущего this._targetElem
 */
DropTarget.prototype._hideHoverIndication = function(avatar) {
  /* override */
};

/**
 * Показать индикацию переноса
 * Вызывается, когда аватар пришел на новый this._targetElem
 */
DropTarget.prototype._showHoverIndication = function(avatar) {
  /* override */
};

/**
 * Метод вызывается при каждом движении аватара
 */
DropTarget.prototype.onDragMove = function(avatar, event) {

  var newTargetElem = this._getTargetElem(avatar, event);

  if (this._targetElem != newTargetElem) {

    this._hideHoverIndication(avatar);
    this._targetElem = newTargetElem;
    this._showHoverIndication(avatar);
  }
};

/**
 * Завершение переноса.
 * Алгоритм обработки (переопределить функцию и написать в потомке):
 * 1. Получить данные переноса из avatar.getDragInfo()
 * 2. Определить, возможен ли перенос на _targetElem (если он есть)
 * 3. Вызвать avatar.onDragEnd() или avatar.onDragCancel()
 *  Если нужно подтвердить перенос запросом на сервер, то avatar.onDragEnd(),
 *  а затем асинхронно, если сервер вернул ошибку, avatar.onDragCancel()
 *  При этом аватар должен уметь "откатываться" после onDragEnd.
 *
 * При любом завершении этого метода нужно (делается ниже):
 *  снять текущую индикацию переноса
 *  обнулить this._targetElem
 */
DropTarget.prototype.onDragEnd = function(avatar, event) {
  this._hideHoverIndication(avatar);
  this._targetElem = null;
};

/**
 * Вход аватара в DropTarget
 */
DropTarget.prototype.onDragEnter = function(fromDropTarget, avatar, event) {};

/**
 * Выход аватара из DropTarget
 */
DropTarget.prototype.onDragLeave = function(toDropTarget, avatar, event) {
  this._hideHoverIndication();
  this._targetElem = null;
};
```

Как видно, из кода выше, по умолчанию `DropTarget` занимается только отслеживанием и индикацией «точки приземления». По умолчанию, единственной возможной «точкой приземления» является сам элемент зоны. В более сложных ситуациях это может быть подэлемент.

Для применения в реальности необходимо как минимум переопределить обработку результата переноса в `onDragEnd`.

TreeDropTarget

`TreeDropTarget` содержит код, специфичный для дерева:

- Индикацию переноса над элементом: методы `_showHoverIndication` и `_hideHoverIndication`.
- Получение текущей точки приземления `_targetElem` в методе `_getTargetElem`. Ей может быть только заголовок узла дерева, причем дополнительно проверяется, что это не потомок переносимого узла.
- Обработка успешного переноса в `onDragEnd`, вставка исходного узла `avatar.dragZoneElem` в узел, соответствующий `_targetElem`.

```
function TreeDropTarget(elem) {
  TreeDropTarget.parent.constructor.apply(this, arguments);
}

extend(TreeDropTarget, DropTarget);

TreeDropTarget.prototype._showHoverIndication = function() {
  this._targetElem && this._targetElem.classList.add('hover');
};

TreeDropTarget.prototype._hideHoverIndication = function() {
  this._targetElem && this._targetElem.classList.remove('hover');
};

TreeDropTarget.prototype._getTargetElem = function(avatar, event) {
  var target = avatar.getTargetElem();
  if (target.tagName != 'SPAN') {
    return;
  }

  // проверить, может быть перенос узла внутрь самого себя или в себя?
  var elemToMove = avatar.getDragInfo(event).dragZoneElem.parentNode;
```

```
  var elem = target;
  while (elem) {
    if (elem == elemToMove) return; // попытка перенести родителя в потомка
    elem = elem.parentNode;
  }

  return target;
};

TreeDropTarget.prototype.onDragEnd = function(avatar, event) {

  if (!this._targetElem) {
    // перенос закончился вне подходящей точки приземления
    avatar.onDragCancel();
    return;
  }

  this._hideHoverIndication();

  // получить информацию об объекте переноса
  var avatarInfo = avatar.getDragInfo(event);

  avatar.onDragEnd(); // аватар больше не нужен, перенос успешен

  // вставить элемент в детей в отсортированном порядке
  var elemToMove = avatarInfo.dragZoneElem.parentNode; // <LI>
  var title = avatarInfo.dragZoneElem.innerHTML; // переносимый заголовок

  // получить контейнер для узлов дерева, соответствующий точке преземления
  var ul = this._targetElem.parentNode.getElementsByTagName('UL')[0];
  if (!ul) { // нет детей, создадим контейнер
    ul = document.createElement('UL');
    this._targetElem.parentNode.appendChild(ul);
  }

  // вставить новый узел в нужное место среди потомков, в алфавитном порядке
  var li = null;
  for (var i = 0; i < ul.children.length; i++) {
    li = ul.children[i];
    var childTitle = li.children[0].innerHTML;
    if (childTitle > title) {
      break;
    }
    li = null;
  }

  ul.insertBefore(elemToMove, li);

  this._targetElem = null;
};
```

Итого

Реализация Drag'n'Drop оказалась отличным способом применить ООП в JavaScript.

Исходный код примера целиком находится в песочнице.

- Синглтон `dragManager` и классы `Drag*` задают общий фреймворк. От них наследуются конкретные объекты. Для создания новых зон достаточно унаследовать стандартные классы и переопределить их.
- Мини-фреймворк для Drag'n'Drop, который здесь представлен, является переписанным и обновленным вариантом реальной библиотеки, на основе которой было создано много успешных скриптов переноса.

  В зависимости от ваших потребностей, вы можете расширить его, добавить перенос нескольких объектов одновременно, поддержку событий и другие возможности.
- На сегодняшний день в каждом серьезном фреймворке есть библиотека для Drag'n'Drop. Она работает похожим образом, но сделать универсальный перенос – штука непростая. Зачастую он перегружен лишним функционалом, либо наоборот – недостаточно расширяем в нужных местах. Понимание, как это все может быть устроено, на примере этой статьи, может помочь в адаптации существующего кода под ваши потребности.

Куки, document.cookie

Для чтения и записи cookie используется свойство `document.cookie`. Однако, оно представляет собой не объект, а строку в специальном формате, для удобной манипуляций с которой нужны дополнительные функции.

Чтение document.cookie

Наверняка у вас есть cookie, которые привязаны к этому сайту. Давайте полюбуемся на них. Вот так:

```
alert( document.cookie );
```

Эта строка состоит из пар `ключ=значение`, которые перечисляются через точку с запятой с пробелом `"; "`.

Значит, чтобы прочитать cookie, достаточно разбить строку по `"; "`, и затем найти нужный ключ. Это можно делать либо через `split` и работу с массивом, либо через регулярное выражение.

Функция getCookie(name)

Следующая функция `getCookie(name)` возвращает cookie с именем `name`:

```
// возвращает cookie с именем name, если есть, если нет, то undefined
function getCookie(name) {
  var matches = document.cookie.match(new RegExp(
    "(?:^|; )" + name.replace(/([\.$?*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + "=([^;]*)"
  ));
  return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Обратим внимание, что значение может быть любым. Если оно содержит символы, нарушающие форматирование, например, пробелы или `;`, то оно кодируется при помощи `encodeURIComponent`. Функция `getCookie` автоматически раскодирует его.

Запись в document.cookie

В `document.cookie` можно писать. При этом запись не перезаписывает существующие cookie, а дополняет к ним!

Например, такая строка поставит cookie с именем `userName` и значением `Vasya`:

```
document.cookie = "userName=Vasya";
```

...Однако, всё не так просто. У cookie есть ряд важных настроек, которые очень желательно указать, так как значения по умолчанию у них неудобны.

Эти настройки указываются после пары ключ=значение, каждое – после точки с запятой:

`path=/mypath`

Путь, внутри которого будет доступ к cookie. Если не указать, то имеется в виду текущий путь и все пути ниже него.

Как правило, используется `path=/`, то есть cookie доступно со всех страниц сайта.

`domain=site.com`

Домен, на котором доступно cookie. Если не указать, то текущий домен. Допустимо указывать текущий домен `site.com` и его поддомены, например `forum.site.com`.

Если указать специальную маску `.site.com`, то cookie будет доступно на сайте и всех его поддоменах. Это используется, например, в случаях, когда кука содержит данные авторизации и должна быть доступна как на `site.com`, так и на `forum.site.com`.

`expires=Tue, 19 Jan 2038 03:14:07 GMT`

Дата истечения куки в формате GMT. Получить нужную дату можно, используя объект `Date`. Его можно установить в любое время, а потом вызвать `toUTCString()`, например:

```
// +1 день от текущего момента
var date = new Date;
date.setDate(date.getDate() + 1);
alert( date.toUTCString() );
```

Если дату не указать, то cookie будет считаться «сессионным». Такое cookie удаляется при закрытии браузера. Если дата в прошлом, то кука будет удалена.

`secure`

Cookie можно передавать только по HTTPS.

Например, чтобы поставить cookie `name=value` по текущему пути с датой истечения через 60 секунд:

```
var date = new Date(new Date().getTime() + 60 * 1000);
document.cookie = "name=value; path=/; expires=" + date.toUTCString();
```

Чтобы удалить это cookie:

```
var date = new Date(0);
document.cookie = "name=; path=/; expires=" + date.toUTCString();
```

При удалении значение не важно. Можно его не указывать, как сделано в коде выше.

Функция setCookie(name, value, options)

Если собрать все настройки воедино, вот такая функция ставит куки:

```
function setCookie(name, value, options) {
  options = options || {};

  var expires = options.expires;

  if (typeof expires == "number" && expires) {
    var d = new Date();
    d.setTime(d.getTime() + expires * 1000);
    expires = options.expires = d;
```

```
  }
  if (expires && expires.toUTCString) {
    options.expires = expires.toUTCString();
  }

  value = encodeURIComponent(value);

  var updatedCookie = name + "=" + value;

  for (var propName in options) {
    updatedCookie += "; " + propName;
    var propValue = options[propName];
    if (propValue !== true) {
      updatedCookie += "=" + propValue;
    }
  }

  document.cookie = updatedCookie;
}
```

Аргументы:

name

название cookie

value

значение cookie (строка)

options

Объект с дополнительными свойствами для установки cookie:

expires

Время истечения cookie. Интерпретируется по-разному, в зависимости от типа:

- Число – количество секунд до истечения. Например, `expires: 3600` – кука на час.
- Объект типа Date – дата истечения.
- Если expires в прошлом, то cookie будет удалено.
- Если expires отсутствует или `0`, то cookie будет установлено как сессионное и исчезнет при закрытии браузера.

path

Путь для cookie.

domain

Домен для cookie.

secure

Если `true`, то пересылать cookie только по защищенному соединению.

Функция deleteCookie(name)

Здесь всё просто – удаляем вызовом `setCookie` с датой в прошлом.

```
function deleteCookie(name) {
  setCookie(name, "", {
    expires: -1
  })
}
```

Сторонние cookie

При работе с cookie есть важная тонкость, которая касается внешних ресурсов.

Теоретически, любой ресурс, который загружает браузер, может поставить cookie.

Например:

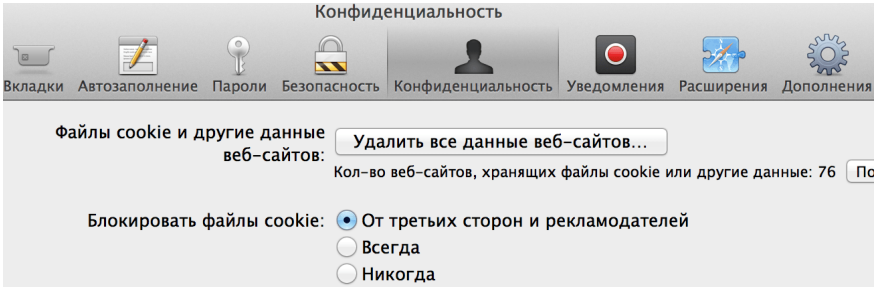
- Если на странице есть `<img src="http://mail.ru/counter.gif">`, то вместе с картинкой в ответ сервер может прислать заголовки, устанавливающие cookie.
- Если на странице есть `<iframe src="http://facebook.com/button.php">`, то во-первых сервер может вместе с `button.php` прислать cookie, а во-вторых JS-код внутри ифрейма может записать в `document.cookie`

При этом cookie будут принадлежать тому домену, который их поставил. То есть, на `mail.ru` для первого случая, и на `facebook.com` во втором.

Такие cookie, которые не принадлежат основной странице, называются «сторонними» (3rd party) cookies. Не все браузеры их разрешают.

Как правило, в настройках браузера можно поставить «Блокировать данные и файлы cookie сторонних сайтов» (Chrome).

В Safari такая настройка включена по умолчанию и выглядит так:



Тс-с-с. Большой брат смотрит за тобой.

Цель этого запрета – защитить посетителей от слежки со стороны рекламодателей, которые вместе с картинкой-баннером присылают и куки, таким образом помечая посетителей.

Например, на многих сайтах стоят баннеры и другая реклама Google Ads. При помощи таких cookie компания Google будет знать, какие именно сайты вы посещаете, сколько времени вы на них проводите и многое другое.

Как? Да очень просто – на каждом сайте загружается, к примеру, картинка с рекламой. При этом баннер берётся с домена, принадлежащего Google. Вместе с баннером Google ставит cookie со специальным уникальным идентификатором.

Далее, при следующем запросе на баннер, браузер пошлёт стандартные заголовки, которые включают в себя:

- Cookie с домена баннера, то есть уникальный идентификатор, который был поставлен ранее.
- Стандартный заголовок Referrer (его не будет при HTTPS!), который говорит, с какого сайта сделан запрос. Да, впрочем, Google и так знает, с какого сайта запрос, ведь идентификатор сайта есть в URL.

Так что Google может хранить в своей базе, какие именно сайты из тех, на которых есть баннер Google, вы посещали, когда вы на них были, и т.п. Этот идентификатор легко привязывается к остальной информации от других сервисов, и таким образом картина слежки получается довольно-таки глобальной.

Здесь я не утверждаю, что в конкретной компании Google всё именно так... Но во-первых, сделать так легко, во-вторых идентификаторы действительно ставятся, а в-третьих, такие знания о человеке позволяют решать, какую именно рекламу и когда ему показать. А это основная доля доходов Google, благодаря которой корпорация существует.

Возможно, компания Apple, которая выпустила Safari, поставила такой флаг по умолчанию именно для уменьшения влияния Google?

А если очень надо?

Итак, Safari запрещает сторонние cookie по умолчанию. Другие браузеры предоставляют такую возможность, если посетитель захочет.

А что, если ну очень надо поставить стороннюю cookie, и чтобы это было надёжно?

Такая задача действительно возникает, например, в системе кросс-доменной авторизации, когда есть несколько доменов 2-го уровня, и хочется, чтобы посетитель, который входит в один сайт, автоматически распознавался во всей сетке. При этом cookie для авторизации ставятся на главный домен – «мастер», а остальные сайты запрашивают их при помощи специального скрипта (и, как правило, копируют к себе для оптимизации, но здесь это не суть).

Ещё пример – когда есть внешний виджет, например, `iframe` с информационным сервисом, который можно подключать на разные сайты. И этот `iframe` должен знать что-то о посетителе, опять же, авторизация или какие-то настройки, которые хорошо бы хранить в cookie.

Есть несколько способов поставить 3rd-party cookie для Safari.

Использовать ифрейм.

Ифрейм является полноценным окном браузера. В нём должна быть доступна вся функциональность, в том числе cookie. Как браузер решает, что ифрейм «сторонний» и нужно запретить для него и его скриптов установку cookie? Критерий таков: «в ифрейме нет навигации». Если навигация есть, то ифрейм признаётся полноценным окном.

Например, в сторонний `iframe` можно сделать POST. И тогда, в ответ на POST, сервер может поставить cookie. Или прислать документ, который это делает. Ифрейм, в который прошёл POST, считается родным и надёжным.

Popup-окно

Другой вариант – использовать popup, то есть при помощи `window.open` открывать именно окно со стороннего домена, и уже там ставить cookie. Это тоже работает.

Редирект

Ещё одно альтернативное решение, которое подходит не везде – это сделать интеграцию со сторонним доменом, такую что на него можно сделать редирект, он ставит cookie и делает редирект обратно.

Дополнительно

- На Cookie наложены ограничения:
  - Имя и значение (после `encodeURIComponent`) вместе не должны превышать 4кб.
  - Общее количество cookie на домен ограничено 30-50, в зависимости от браузера.
  - Разные домены 2-го уровня полностью изолированы. Но в пределах доменов 3-го уровня куки можно ставить свободно с указанием `domain`.
  - Сервер может поставить cookie с дополнительным флагом `HttpOnly`. Cookie с таким параметром передаётся только в заголовках, оно никак не доступно из JavaScript.
- Иногда посетители отключают cookie. Отловить это можно проверкой свойства navigator.cookieEnabled

```
if (!navigator.cookieEnabled) {
  alert( 'Включите cookie для комфортной работы с этим сайтом' );
}
```

...Конечно, предполагается, что включён JavaScript. Впрочем, посетитель без JS и cookie с большой вероятностью не человек, а бот.

Итого

Файл с функциями для работы с cookie: cookie.js.

Intl: интернационализация в JavaScript

Общая проблема строк, дат, чисел в JavaScript – они «не в курсе» языка и особенностей стран, где находится посетитель.

В частности:

Строки

При сравнении сравниваются коды символов, а это неправильно, к примеру, в русском языке оказывается, что `"ё" > "я"` и `"а" > "я"`, хотя всем известно, что `я` – последняя буква алфавита и это она должна быть больше любой другой.

Даты

В разных странах принята разная запись дат. Где-то пишут 31.12.2014 (Россия), а где-то 12/31/2014 (США), где-то иначе.

Числа

В одних странах выводятся цифрами, в других – иероглифами, длинные числа разделяются где-то пробелом, где-то запятой.

Все современные браузеры, кроме IE10- (но есть библиотеки и для него) поддерживают стандарт ECMA 402, предназначенный решить эти проблемы навсегда.

Основные объекты

`Intl.Collator`

Умеет правильно сравнивать и сортировать строки.

`Intl.DateTimeFormat`

Умеет форматировать дату и время в соответствии с нужным языком.

`Intl.NumberFormat`

Умеет форматировать числа в соответствии с нужным языком.

Локаль

Локаль – первый и самый важный аргумент всех методов, связанных с интернационализацией.

Локаль описывается строкой из трёх компонентов, которые разделяются дефисом:

1. Код языка.
2. Код способа записи.
3. Код страны.

На практике не всегда указаны три, обычно меньше:

1. `ru` – русский язык, без уточнений.
2. `en-GB` – английский язык, используемый в Англии (`GB`).
3. `en-US` – английский язык, используемый в США (`US`).
4. `zh-Hans-CN` – китайский язык (`zh`), записываемый упрощённой иероглифической письменностью (`Hans`), используемый в китае.

Также через суффикс `-u-*` можно указать расширения локалей, например `"th-TH-u-nu-thai"` – тайский язык (`th`), используемый в Тайланде (`TH`), с записью чисел тайскими буквами (๐, ๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙) .

Стандарт, который описывает локали – RFC 5464, языки описаны в IANA language registry.

Все методы принимают локаль в виде строки или массива, содержащего несколько локалей в порядке предпочтения.

Если локаль не указана или `undefined` – берётся локаль по умолчанию, установленная в окружении (браузере).

Подбор локали localeMatcher

`localeMatcher` – вспомогательная настройка, которую тоже можно везде указать, она определяет способ подбора локали, если желаемая недоступна.

У него два значения:

- `"lookup"` – означает простейший порядок поиска путём обрезания суффикса, например `zh-Hans-CN` → `zh-Hans` → `zh` → локаль по умолчанию.

- `"best fit"` – использует встроенные алгоритмы и предпочтения браузера (или другого окружения) для выбора подходящей локали.

По умолчанию стоит **`"best fit"`**.

Если локалей несколько, например `["zh-Hans-CN", "ru-RU"]` то `localeMatcher` пытается подобрать наиболее подходящую локаль для первой из списка (китайская), если не получается – переходит ко второй (русской) и так далее. Если ни одной не нашёл, например на компьютере не совсем поддерживается ни китайский ни русский, то используется локаль по умолчанию.

Как правило, `"best fit"` является здесь наилучшим выбором.

Строки, Intl.Collator

Синтаксис:

```
// создание
var collator = new Intl.Collator([locales, [options]])
```

Параметры:

`locales`

Локаль, одна или массив в порядке предпочтения.

`options`

Объект с дополнительными настройками:

- `localeMatcher` – алгоритм выбора подходящей локали.
- `usage` – цель сравнения: сортировка `"sort"` или поиск `"search"`, по умолчанию `"sort"`.
- `sensitivity` – чувствительность: какие различия в символах учитывать, а какие – нет, варианты:
  - `base` – учитывать только разные символы, без диакритических знаков и регистра, например: `а ≠ б`, `е = ё`, `а = А`.
  - `accent` – учитывать символы и диакритические знаки, например: `а ≠ б`, `е ≠ ё`, `а = А`.
  - `case` – учитывать символы и регистр, например: `а ≠ б`, `е = ё`, `а ≠ А`.
  - `variant` – учитывать всё: символ, диакритические знаки, регистр, например: `а ≠ б`, `е ≠ ё`, `а ≠ А`, используется по умолчанию.
- `ignorePunctuation` – игнорировать знаки пунктуации: `true/false`, по умолчанию `false`.
- `numeric` – использовать ли численное сравнение: `true/false`, если `true`, то будет `12 > 2`, иначе `12 < 2`.
- `caseFirst` – в сортировке должны идти первыми прописные или строчные буквы, варианты: `"upper"` (прописные), `lower` (строчные) или `false` (стандартное для локали, также является значением по умолчанию). Не поддерживается IE11-.

В подавляющем большинстве случаев подходят стандартные параметры, то есть `options` указывать не нужно.

Использование:

```
var result = collator.compare(str1, str2);
```

Результат `compare` имеет значение `1` (больше), `0` (равно) или `-1` (меньше).

Например:

```
var collator = new Intl.Collator();

alert( "ёжик" > "яблоко" ); // true (ёжик больше, что неверно)
alert( collator.compare("ёжик", "яблоко") ); // -1 (ёжик меньше, верно)
```

Выше были использованы полностью стандартные настройки. Они различают регистр символа, но это различие можно убрать, если настроить чувствительность `sensitivity`:

```
var collator = new Intl.Collator();
alert( collator.compare("ЁжиК", "ёжик") ); // 1, разные

var collator = new Intl.Collator(undefined, {
  sensitivity: "accent"
});
alert( collator.compare("ЁжиК", "ёжик") ); // 0, одинаковые
```

Даты, Intl.DateFormatter

Синтаксис:

```
// создание
var formatter = new Intl.DateFormatter([locales, [options]])
```

Первый аргумент – такой же, как и в `Collator`, а в объекте `options` мы можем определить, какие именно части даты показывать (часы, месяц, год...) и в каком формате.

Полный список свойств `options`:

Свойство	Описание	Возможные значения	По умолчанию
`localeMatcher`	Алгоритм подбора локали	lookup,best fit	best fit
`formatMatcher`	Алгоритм подбора формата	`basic`, `best fit`	`best fit`
`hour12`	Включать ли время в 12-часовом формате	`true` -- 12-часовой формат, `false` -- 24-часовой	
`timeZone`	Временная зона	Временная зона, например `Europe/Moscow`	`UTC`
`weekday`	День недели	`narrow`, `short`, `long`	
`era`	Эра	`narrow`, `short`, `long`	
`year`	Год	`2-digit`, `numeric`	`undefined` или `numeric`
`month`	Месяц	`2-digit`, `numeric`, `narrow`, `short`, `long`	`undefined` или `numeric`
`day`	День	`2-digit`, `numeric`	`undefined` или `numeric`
`hour`	Час	`2-digit`, `numeric`	
`minute`	Минуты	`2-digit`, `numeric`	
second	Секунды	`2-digit`, `numeric`	
`timeZoneName`	Название таймзоны (нет в IE11)	`short`, `long`	

Все локали обязаны поддерживать следующие наборы настроек:

- weekday, year, month, day, hour, minute, second
- weekday, year, month, day
- year, month, day
- year, month
- month, day
- hour, minute, second

Если указанный формат не поддерживается, то настройка `formatMatcher` задаёт алгоритм подбора наиболее близкого формата: `basic` – по стандартным правилам и `best fit` – по умолчанию, на усмотрение окружения (браузера).

Использование:

```
var dateString = formatter.format(date);
```

Например:

```
var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru");
alert( formatter.format(date) ); // 31.12.2014

var formatter = new Intl.DateTimeFormat("en-US");
alert( formatter.format(date) ); // 12/31/2014
```

Длинная дата, с настройками:

```
var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric"
});

alert( formatter.format(date) ); // среда, 31 декабря 2014 г.
```

Только время:

```
var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {
  hour: "numeric",
  minute: "numeric",
  second: "numeric"
});

alert( formatter.format(date) ); // 12:30:00
```

Числа: Intl.NumberFormat

Форматтер `Intl.NumberFormat` умеет красиво форматировать не только числа, но и валюту, а также проценты.

Синтаксис:

```
var formatter = new Intl.NumberFormat([locales[, options]]);

formatter.format(number); // форматирование
```

Параметры, как и раньше – локаль и опции.

Список опций:

Свойство	Описание	Возможные значения	По умолчанию
`localeMatcher`	Алгоритм подбора локали	`lookup`, `best fit`	`best fit`
`style`	Стиль форматирования	`decimal`, `percent`, `currency`	`decimal`
`currency`	Алфавитный код валюты	См. [Список кодов валюты](http://www.currency-iso.org/en/home/tables/table-a1.html), например `USD`	
`currencyDisplay`	Показывать валюту в виде кода, локализованного символа или локализованного названия	`code`, `symbol`, `name`	`symbol`
`useGrouping`	Разделять ли цифры на группы	`true`, `false`	`true`
`minimumIntegerDigits`	Минимальное количество цифр целой части	от `1` до `21`	21
`minimumFractionDigits`	Минимальное количество десятичных цифр	от `0` до `20`	для чисел и процентов `0`, для валюты зависит от кода.
`maximumFractionDigits`	Максимальное количество десятичных цифр	от `minimumFractionDigits` до `20`.	для чисел `max(minimumFractionDigits, 3)`, для процентов `0`, для валюты зависит от кода.
`minimumSignificantDigits`	Минимальное количество значимых цифр	от `1` до `21`	`1`
`maximumSignificantDigits`	Максимальное количество значимых цифр	от `minimumSignificantDigits` до `21`	`minimumSignificantDigits`

Пример без опций:

```
var formatter = new Intl.NumberFormat("ru");
alert( formatter.format(1234567890.123) ); // 1 234 567 890,123
```

С ограничением значимых цифр (важны только первые 3):

```
var formatter = new Intl.NumberFormat("ru", {
  maximumSignificantDigits: 3
});
alert( formatter.format(1234567890.123) ); // 1 230 000 000
```

С опциями для валюты:

```
var formatter = new Intl.NumberFormat("ru", {
  style: "currency",
  currency: "GBP"
});

alert( formatter.format(1234.5) ); // 1 234,5 £
```

С двумя цифрами после запятой:

```
var formatter = new Intl.NumberFormat("ru", {
  style: "currency",
  currency: "GBP",
  minimumFractionDigits: 2
});

alert( formatter.format(1234.5) ); // 1 234,50 £
```

Методы в Date, String, Number

Методы форматирования также поддерживаются в обычных строках, датах, числах:

`String.prototype.localeCompare(that [, locales [, options]])`

Сравнивает строку с другой, с учётом локали, например:

```
var str = "ёжик";

alert( str.localeCompare("яблоко", "ru") ); // -1
```

Date.prototype.toLocaleString([locales [, options]])

Форматирует дату в соответствии с локалью, например:

```
var date = new Date(2014, 11, 31, 12, 00);

alert( date.toLocaleString("ru", { year: 'numeric', month: 'long' }) ); // Декабрь 2014
```

Date.prototype.toLocaleDateString([locales [, options]])

То же, что и выше, но опции по умолчанию включают в себя год, месяц, день

Date.prototype.toLocaleTimeString([locales [, options]])

То же, что и выше, но опции по умолчанию включают в себя часы, минуты, секунды

Number.prototype.toLocaleString([locales [, options]])

Форматирует число, используя опции `Intl.NumberFormat` .

Все эти методы при запуске создают соответствующий объект `Intl.*` и передают ему опции, можно рассматривать их как укороченные варианты вызова.

Старые IE

В IE10- рекомендуется использовать полифилл, например библиотеку https://github.com/andyearnshaw/Intl.js.

Задачи

Отсортируйте массив с буквой ё

важность: 5

Используя `Intl.Collate` , отсортируйте массив:

```
var animals = ["тигр", "ёж", "енот", "ехидна", "АИСТ", "ЯК"];

// ... ваш код ...

alert( animals ); // АИСТ,ёж,енот,ехидна,тигр,ЯК
```

В этом примере порядок сортировки не должен зависеть от регистра.

Что касается буквы `"ё"` , то мы следуем обычным правилам сортировки буквы ё, по которым «е» и «ё» считаются одной и той же буквой, за исключением случая, когда два слова отличаются только в позиции буквы «е» / «ё» – тогда слово с «е» ставится первым.

К решению

Особенности регулярных выражений в Javascript

Регулярные выражения в javascript немного странные. Вроде – перловые, обычные, но с подводными камнями, на которые натыкаются даже опытные javascript-разработчики.

Эта статья ставит целью перечислить неожиданные фишки и особенности `RegExp` в краткой и понятной форме.

Точка и перенос строки

Для поиска в многострочном режиме почти все модификации перловых регэкспов используют специальный multiline-флаг.

И javascript здесь не исключение.

Попробуем же сделать поиск и замену многострочного вхождения. Скажем, будем заменять `[u]` … `[/u]` на тэг подчеркивания: `<u>` :

```
function bbtagit(text) {
  text = text.replace(/\[u\](.*?)\[\/u\]/gim, '<u>$1</u>')

  return text
}

var line = "[u]мой\n текст[/u]"
alert(bbtagit(line))
```

Попробуйте запустить. Заменяет? Как бы не так!

Дело в том, что в javascript мультилайн режим (флаг `m`) влияет только на символы ^ и $, которые начинают матчиться с началом и концом строки, а не всего текста.

Точка по-прежнему – любой символ, кроме новой строки. В javascript нет флага, который устанавливает мультилайн-режим для точки. Для того, чтобы заматчить совсем что угодно – используйте `[\s\S]`.

Работающий вариант:

```
function bbtagit(text) {
  text = text.replace(/\[u\]([\s\S]*)\[\/u\]/gim, '<u>$1</u>')

  return text
}

var line = "[u]мой\n текст[/u]"
alert(bbtagit(line))
```

Жадность

Это не совсем особенность, скорее фича, но все же достойная отдельного абзаца.

Все регулярные выражения в javascript – жадные. То есть, выражение старается отхватить как можно больший кусок строки.

Например, мы хотим заменить все открывающие тэги `<a>`. На что и почему – не так важно.

```
var text = '1 <A href="#">...</A> 2'
text = text.replace(/<A(.*)>/, 'TEST')
alert(text)
```

При запуске вы увидите, что заменяется не открывающий тэг, а вся ссылка, выражение матчит ее от начала и до конца.

Это происходит из-за того, что точка-звездочка в «жадном» режиме пытается захватить как можно больше, в нашем случае – это как раз до последнего `>`.

Последний символ `>` точка-звездочка не захватывает, т.к. иначе не будет совпадения.

Как вариант решения используют квадратные скобки: `[^>]`:

```
var text = '1 <A href="#">...</A> 2'
text = text.replace(/<A([^>]*)>/, 'TEST')
alert(text)
```

Это работает. Но самым удобным вариантом является переключение точки-звездочки в нежадный режим. Это осуществляется простым добавлением знака « ? » после звездочки.

В нежадном режиме точка-звездочка пустит поиск дальше сразу, как только нашла совпадение:

```
var text = '1 <A href="#">...</A> 2'
text = text.replace(/<A(.*?)>/, 'TEST')
alert(text)
```

В некоторых языках программирования можно переключить жадность на уровне всего регулярного выражения, флагом.

В javascript это сделать нельзя... Вот такая особенность. А вопросительный знак после звездочки рулит – честное слово.

Backreferences в паттерне и при замене

Иногда нужно в самом паттерне поиска обратиться к предыдущей его части.

Например, при поиске BB-тагов, то есть строк вида `[u]…[/u]`, `[b]…[/b]` и `[s]…[/s]`. Или при поиске атрибутов, которые могут быть в одинарных кавычках или двойных.

Обращение к предыдущей части паттерна в javascript осуществляется как \1, \2 и т.п., бэкслеш + номер скобочной группы:

```
var text = ' [b]a [u]b[/u] c [/b] ';

var reg = /\[([bus])\](.*?)\[\/\1\] /;
text = text.replace(reg, '<$1>$2</$1>');  // <b>a [u]b[/u] c </b>
alert(text);
```

Обращение к скобочной группе в строке замены идет уже через доллар: `$1`. Не знаю, почему, наверное так удобнее...

P.S. Понятно, что при таком способе поиска bb-тагов придется пропустить текст через замену несколько раз – пока результат не перестанет отличаться от оригинала.

Найти все / Заменить все

Эти две задачи решаются в javascript принципиально по-разному.

Начнем с «простого».

Заменить все

Для замены всех вхождений используется метод String#replace. Он интересен тем, что допускает первый аргумент – регэксп или строку.

Если первый аргумент – строка, то будет осуществлен поиск подстроки, без преобразования в регулярное выражение.

Попробуйте:

```
alert("2 ++ 1".replace("+", "*"))
```

Как видите, заменился только один плюс, а не оба.

Чтобы заменить все вхождения, String#replace обязательно нужно использовать с регулярным выражением.

В режиме регулярного выражения плюс придётся экранировать, но зато `replace` заменит все вхождения (при указании флага `g`):

```
alert("2 ++ 1".replace(/\+/g, "*"))
```

Вот такая особенность работы со строкой.

Заменить функцией

Очень полезной особенностью `replace` является возможность работать с функцией вместо строки замены. Такая функция получает первым аргументом – все совпадения, а последующими аргументами – скобочные группы.

Следующий пример произведет операции вычитания:

```
var str = "count 36 - 26, 18 - 9"
str = str.replace(/(\d+) - (\d+)/g, function(a,b,c) { return b-c })
alert(str)
```

Найти всё

В javascript нет одного универсального метода для поиска всех совпадений. Для поиска без запоминания скобочных групп – можно использовать String#match:

```
var str = "count 36-26, 18-9";
var re = /(\d+)-(\d+)/g;
var result = str.match(re);
for (var i = 0; i < result.length; i++) {
  alert(result[i]);
}
```

Как видите, оно исправно ищет все совпадения (флаг „g“ у регулярного выражения обязателен), но при этом не запоминает скобочные группы. Эдакий «облегченный вариант».

Найти всё с учётом скобочных групп

В сколько-нибудь сложных задачах важны не только совпадения, но и скобочные группы. Чтобы их найти, предлагается использовать многократный вызов RegExp#exec.

Для этого регулярное выражение должно использовать флаг „g“. Тогда результат поиска, запомненный в свойстве `lastIndex` объекта `RegExp` используется как точка отсчета для следующего поиска:

```
var str = "count 36-26, 18-9"
var re = /(\d+)-(\d+)/g
var res
while ((res = re.exec(str)) != null) {
  alert("Найдено " + res[0] + ":  (" + res[1] + ") и (" + res[2] + ")")
  alert("Дальше ищу с позиции " + re.lastIndex)
}
```

Проверка `while( (res = re.exec(str)) != null)` нужна т.к. значение `res = 0` является хорошим и означает, что вхождение найдено в самом начале строки (поиск успешен). Поэтому необходимо сравнивать именно с `null`.

Решения

Основы XMLHttpRequest

Выведите телефоны

Код для загрузки и вывода телефонов:

```
function loadPhones() {

  var xhr = new XMLHttpRequest();

  xhr.open('GET', 'phones.json', true);

  xhr.send();

  xhr.onreadystatechange = function() {
    if (xhr.readyState != 4) return;

    button.parentNode.removeChild(button);
```

```
    if (xhr.status != 200) {
      // обработать ошибку
      alert( xhr.status + ': ' + xhr.statusText );
    } else {
      try {
        var phones = JSON.parse(xhr.responseText);
      } catch (e) {
        alert( "Некорректный ответ " + e.message );
      }
      showPhones(phones);
    }

  }

  button.innerHTML = 'Загружаю...';
  button.disabled = true;
}
```

```
function showPhones(phones) {

    phones.forEach(function(phone) {
      var li = list.appendChild(document.createElement('li'));
      li.innerHTML = phone.name;
    });

  }
```

Полное решение с возможнотью скачать:

Обратите внимание – код обрабатывает возможную ошибку при чтении JSON при помощи `try..catch`.

Технически, это такая же ошибка, как и `status != 200`. Ведь сервер обязан присылать корректный JSON. Поэтому если уж обрабатываем ошибки запроса, то и её тоже.

К условию

XMLHttpRequest: кросс-доменные запросы

Зачем нужен Origin?

`Origin` нужен, потому что `Referer` передаётся не всегда. В частности, при запросе с HTTPS на HTTP – нет `Referer`.

Политика Content Security Policy может запрещать пересылку `Referer`.

По стандарту `Referer` является необязательным HTTP-заголовком, в некоторых браузерах есть настройки, которые запрещают его слать.

Именно поэтому, ввиду того, что на `Referer` полагаться нельзя, и придумали заголовок `Origin`, который гарантированно присылается при кросс-доменных запросах.

Что же касается «неправильного» `Referer` – это из области фантастики. Когда-то, много лет назад, в браузерах были ошибки, которые позволяли подменить `Referer` из JavaScript, но они давно исправлены. Никакая «злая страница» не может его подменить.

К условию

CSS-анимации

Анимировать самолёт (CSS)

CSS-код для анимации одновременно `width` и `height`:

```
/* исходный класс */

#flyjet {
  transition: all 3s;
}
/* JS добавляет .growing *.
#flyjet.growing {
  width: 400px;
  height: 240px;
}
```

Небольшая тонкость с окончанием анимации. Соответствующее событие `transitionend` сработает два раза – по одному для каждого свойства. Поэтому, если не предпринять дополнительных шагов, сообщение из обработчика может быть выведено 2 раза.

Открыть решение в песочнице.

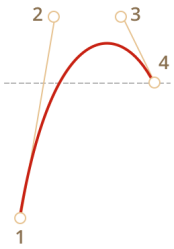
К условию

Анимировать самолёт с перелётом (CSS)

Для такой анимации необходимо подобрать правильную кривую Безье.

Чтобы она выпрыгивала вверх, обе опорные точки можно вынести по `y>1`, например: `cubic-bezier(0.25, 1.5, 0.75, 1.5)` (промежуточные опорные точки имеют `y=1.5`).

Её график:



Открыть решение в песочнице.

К условию

JS-Анимация

Анимируйте мяч

В HTML/CSS, падение мяча можно отобразить изменением свойства `ball.style.top` от 0 и до значения, соответствующего нижнему положению.

Нижняя граница элемента `field`, в котором находится мяч, имеет значение `field.clientHeight`. Но свойство `top` относится к верху мяча, поэтому оно меняется до `field.clientHeight - ball.clientHeight`.

Для создания анимационного эффекта лучше всего подойдет функция `bounce` в режиме `easeOut`.

Следующий код даст нам нужный результат:

```
var to = field.clientHeight - ball.clientHeight;

animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw: function(progress) {
    ball.style.top = to * progress + 'px'
  }
});
```

Открыть решение в песочнице.

К условию

Анимируйте падение мяча с отскоками вправо

В задаче Анимируйте мяч создаётся подпрыгивающий мяч. Нам нужно всего лишь добавить еще одну анимацию для `elem.style.left`.

Горизонтальная координата меняется по другому закону, нежели вертикальная. Она не «подпрыгивает», а постоянно увеличивается, постепенно сдвигая мяч вправо.

Поэтому мы не можем добавить её в тот же `animate`, нужно делать отдельный.

В качестве временной функции для перемещения вправо мы могли бы применить для неё `linear`, но тогда горизонтальное движение будет отставать от скачков мяча. Более красиво будет что-то типа `makeEaseOut(quad)`.

Код:

```
var height = field.clientHeight - ball.clientHeight;
var width = 100;

animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw: function(progress) {
    ball.style.top = height * progress + 'px'
  }
```

```
});

animate({
  duration: 2000,
  timing: makeEaseOut(quad),
  draw: function(progress) {
    ball.style.left = width * progress + "px"
  }
});
```

Открыть решение в песочнице.

К условию

Свойство "float"

Разница inline-block и float

Разница колоссальная.

В первую очередь она в том, что `inline-block` продолжают участвовать в потоке, а `float` – нет.

Чтобы её ощутить, достаточно задать себе следующие вопросы:

1. Что произойдёт, если контейнеру `UL` поставить рамку `border` – в первом и во втором случае?
2. Что будет, если элементы `LI` различаются по размеру? Будут ли они корректно перенесены на новую строку в обоих случаях?
3. Как будут вести себя блоки, находящиеся под галереей?

Попробуйте сами на них ответить.

Затем читайте дальше.

Что будет, если контейнеру **`UL`** поставить рамку **`border`** ?

Контейнер не выделяет пространство под `float` . А больше там ничего нет. В результате он просто сожмётся в одну линию сверху.

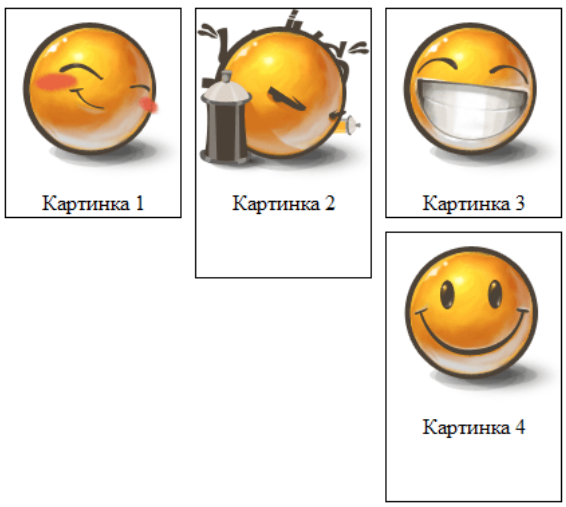
Попробуйте сами, добавьте рамку в песочнице.

А в случае с `inline-block` всё будет хорошо, т.к. элементы остаются в потоке.

Что будет, если элементы **`LI`** различаются по размеру? Будут ли они корректно перенесены на новую строку в обоих случаях?

При `float:left` элементы двигаются направо до тех пор, пока не наткнутся на границу внешнего блока (с учётом `padding`) или на другой `float` -элемент.

Может получиться вот так:



Вы можете увидеть это, открыв демо-галерею в отдельном окне и изменяя его размер:

При использовании `inline-block` таких странностей не будет, блоки перенесутся корректно на новую строку. И, кроме того, можно выровнять элементы по высоте при помощи `li { vertical-align:middle }` :

Как будут вести себя блоки, находящиеся под галереей?

В случае с `float` нужно добавить дополнительную очистку с `clear`, чтобы поведение было идентично обычному блоку.

Иначе блоки, находящиеся под галереей, вполне могут «заехать» по вертикали на территорию галереи.

Открыть решение в песочнице.

К условию

Дерево с многострочными узлами

Для решения можно применить принцип двухколоночной верстки `float + margin`. Иконка будет левой колонкой, а содержимое – правой.

Открыть решение в песочнице.

К условию

Постраничная навигация (CSS)

HTML-структура:

```
<div class="nav">
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <img src="arrow-right.jpg" class="right" width="40" height="40">
  <ul class="pages">
    <li>...</li>
  </ul>
</div>
```

Стили:

```
.nav {
  height: 40px;
  width: 80%;
  margin: auto;
}

.nav .left {
  float: left;
  cursor: pointer;
}

.nav .right {
  float: right;
  cursor: pointer;
}

.nav .pages {
  list-style: none;
  text-align: center;
  margin: 0;
  padding: 0;
}

.nav .pages li {
  display: inline;
```

```
  margin: 0 3px;
  line-height: 40px;
  cursor: pointer;
}
```

Основные моменты:

- Сначала идёт левая кнопка, затем правая, а лишь затем – текст. Почему так, а не лево – центр – право?

  Дело в том, что `float` смещает элемент вправо относительно обычного места. А какое обычное место будет у правого `IMG` без `float`?

  Оно будет под списком, так как список – блочный элемент, а `IMG` – инлайн-элемент. При добавлении `float:right` элемент `IMG` сдвинется вправо, оставшись под списком.

  Код в порядке лево-центр-право (неправильный):

```
<div...>
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <ul class="pages"> (li) 1 2 3 4 5 6 7 8 9</ul>
  <img src="arrow-right.jpg" class="right" width="40" height="40">
</div>
```

  Его демо:

Текст Сверху

1 2 3 4 5 6 7 8 9

Текст Снизу

  Правильный порядок: лево-право-центр, тогда `float` останется на верхней строке.

  Код, который даёт правильное отображение:

```
<div ...>
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <img src="arrow-right.jpg" class="right" width="40" height="40">
  <ul class="pages"> .. список .. </ul>
</div>
```

  Также можно расположить стрелки при помощи `position: absolute`. Тогда, чтобы текст при уменьшении размеров окна не налез на стрелки – нужно добавить в контейнер левый и правый `padding`:

  Выглядеть будет примерно так:

```
<div style="position:relative; padding: 0 40px;">
  <img style="position:absolute;left:0" src="..left.." width="40" height="40">
  <ul> (li) 1 2 3 4 5 6 7 8 9 </ul>
  <img style="position:absolute;right:0" srr="..right.." width="40" height="40">
</div>
```

- Центрирование одной строки по вертикали осуществляется указанием **`line-height`**, равной высоте.

  Это красиво лишь для одной строки: если окно становится слишком узким, и строка вдруг разбивается на две – получается некрасиво, хотя и читаемо.

  Если хочется сделать красивее для двух строк, то можно использовать другой способ центрирования.

Открыть решение в песочнице.

К условию

Добавить рамку, сохранив ширину

Подсказка

Решение

К условию

Свойство "position"

Модальное окно

Если использовать `position: absolute`, то `DIV` не растянется на всю высоту документа, т.к. координаты вычисляются *относительно окна*.

Можно, конечно, узнать эту высоту при помощи JavaScript, но CSS даёт более удобный способ. Будем использовать `position:fixed`:

Стиль:

```
#box {
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  z-index: 999;
}
```

Свойство `z-index` должно превосходить все другие элементы управления, чтобы они перекрывались.

Открыть решение в песочнице.

К условию

Центрирование горизонтальное и вертикальное

Поместите мяч в центр поля (CSS)

Сместим мяч в центр при помощи `left/top=50%`, а затем приподымем его указанием `margin`:

```
#ball {
  position: absolute;
  left: 50%;
  top: 50%;
  margin-left: -20px;
  margin-top: -20px;
}
```

Открыть решение в песочнице.

К условию

Форма + модальное окно

Структура решения

Решение

К условию

vertical-align + table-cell + position = ?

Подсказка

Решение

К условию

Свойство "margin"

Нерабочие margin?

Ошибка заключается в том, что `margin` при задании в процентах высчитывается *относительно ширины*. Так написано в стандарте.

При этом не важно, какой отступ: левый, правый. верхний или нижний. Все они в процентах отсчитываются от ширины. Из-за этого и ошибка.

Ситуацию можно исправить, например, заданием `margin-top/margin-bottom` в пикселях, если это возможно или, в качестве альтернативы, использовать другие средства, в частности, `position` или `padding-top/padding-bottom` на родителе.

К условию

Расположить текст внутри INPUT

Подсказка

Решение

К условию

Знаете ли вы селекторы?

Выберите элементы селектором

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>

  <input type="checkbox">
  <input type="checkbox" checked>
  <input type="text" id="message">

  <h3 id="widget-title">Сообщения:</h3>
  <ul id="messages">
    <li id="message-1">Сообщение 1</li>
    <li id="message-2">Сообщение 2</li>
    <li id="message-3" data-action="delete">Сообщение 3</li>
    <li id="message-4" data-action="edit do-not-delete">Сообщение 4</li>
    <li id="message-5" data-action="edit delete">Сообщение 5</li>
    <li><a href="#">...</a></li>
  </ul>


  <a href="http://site.com/list.zip">Ссылка на архив</a>
  <a href="http://site.com/list.pdf">..И на PDF</a>


  <script>
    // тестовая функция для селекторов
    // проверяет, чтобы элементов по селектору selector было ровно count
    function test(selector, count) {
      var elems = document.querySelectorAll(selector);
      var ok = (elems.length == count);

      if (!ok) alert(selector + ": " + elems.length + " != " + count);
    }

    // ------------- селекторы --------------

    // Выбрать input типа checkbox
    test('input[type="checkbox"]', 1);

    // Выбрать input типа checkbox, НЕ отмеченный
    test('input[type="checkbox"]:not(:checked)', 1);

    // Найти все элементы с id=message или message-*
    test('[id|="message"]', 6);

    // Найти все элементы с id=message-*
    test('[id^="message-"]', 5);

    // Найти все ссылки с расширением href="...zip"
    test('a[href$=".zip"]', 1);

    // Найти все элементы с data-action, содержащим delete в списке (через пробел)
    test('[data-action~="delete"]', 2);

    // Найти все элементы, у которых ЕСТЬ атрибут data-action,
    // но он НЕ содержащит delete в списке (через пробел)
    test('[data-action]:not([data-action~="delete"])', 1);

    // Выбрать все чётные элементы списка #messages
    test('#messages li:nth-child(2n)', 3);

    // Выбрать один элемент сразу за заголовком h3#widget-title
    // на том же уровне вложенности
    test('h3#widget-title + *', 1);

    // Выбрать все ссылки, следующие за заголовком h3#widget-title
    // на том же уровне вложенности
    test('h3#widget-title ~ a', 2);

    // Выбрать ссылку внутри последнего элемента списка #messages
    test('#messages li:last-child a', 1);
```

```
  </script>
</body>

</html>
```

К условию

Отступ между элементами, размер одна строка

Выбор элементов

Решение

Ещё решение

К условию

Отступ между парами, размером со строку

Селектор

Правило

К условию

Классы и спецсимволы

Найдите время

Ответ: `\d\d:\d\d` .

```
alert( "Завтрак в 09:00.".match( /\d\d:\d\d/ ) ); // 09:00
```

К условию

Наборы и диапазоны [...]

Java[^script]

Ответы: нет, да.

- В строке `Java` он ничего не найдёт, так как исключающие квадратные скобки в `Java[^...]` означают «один символ, кроме указанных». А после «Java» – конец строки, символов больше нет.

  ```
  alert( "Java".match(/Java[^script]/) ); // нет совпадений
  ```

- Да, найдёт. Поскольку регэксп регистрозависим, то под `[^script]` вполне подходит символ `"S"` .

  ```
  alert( "JavaScript".match(/Java[^script]/) ); // "JavaS"
  ```

К условию

Найдите время в одном из форматов

Ответ: `\d\d[-:]\d\d` .

```
var re = /\d\d[-:]\d\d/g;
alert( "Завтрак в 09:00. Обед - в 21-30".match(re) );
```

Обратим внимание, что дефис `'-'` не экранирован, поскольку в начале скобок он не может иметь специального смысла.

К условию

Квантификаторы +, *, ? и {n}

Как найти многоточие... ?

Решение:

```
var reg = /\.{3,}/g;
alert( "Привет!... Как дела?.....".match(reg) ); // ..., .....
```

Заметим, что символ `.` является специальным, значит его надо экранировать, то есть вставлять как `\.`.

К условию

Регулярное выражение для цвета

Итак, нужно написать выражение для описания цвета, который начинается с «#», за которым следуют 6 шестнадцатеричных символов.

Шестнадцатеричный символ можно описать с помощью `[0-9a-fA-F]`. Мы можем сократить выражение, используя не чувствительный к регистру шаблон `[0-9a-f]`.

Для его шестикратного повторения мы будем использовать квантификатор `{6}`.

В итоге, получаем выражение вида `/#[a-f0-9]{6}/gi`.

```
var re = /#[a-f0-9]{6}/gi;

var str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2";

alert( str.match(re) );  // #121212,#AA00ef
```

Проблема этого выражения в том, что оно находит цвет и в более длинных последовательностях:

```
alert( "#12345678".match( /#[a-f0-9]{6}/gi ) ) // #12345678
```

Чтобы такого не было, можно добавить в конец `\b`:

```
// цвет
alert( "#123456".match( /#[a-f0-9]{6}\b/gi ) ); // #123456

// не цвет
alert( "#12345678".match( /#[a-f0-9]{6}\b/gi ) ); // null
```

К условию

Найдите положительные числа

Целое число – это `\d+`.

Десятичная точка с дробной частью – `\.\d+`.

Она не обязательна, так что обернём её в скобки с квантификатором `'?'`.

Итого, получилось регулярное выражение `\d+(\.\d+)?`:

```
var re = /\d+(\.\d+)?/g

var str = "1.5 0 12. 123.4.";

alert( str.match(re) );   // 1.5, 0, 12, 123.4
```

К условию

Найдите десятичные числа

Целое число с необязательной дробной частью – это `\d+(\.\d+)?`.

К этому нужно добавить необязательный `-` в начале:

```
var re = /-?\d+(\.\d+)?/g
var str = "-1.5 0 2 -123.4.";
alert( str.match(re) );   // -1.5, 0, 2, -123.4
```

К условию

Жадные и ленивые квантификаторы

Совпадение для /d+? d+/

Результат: `123 456`.

Ленивый `\d+?` будет брать цифры до пробела, то есть `123`. После каждой цифры он будет останавливаться, проверять – не пробел ли дальше? Если нет – брать ещё цифру, в итоге возьмёт `123`.

Затем в дело вступит `\d+`, который по-максимуму возьмёт дальнейшие цифры, то есть `456`.

К условию

Различие между "[^"]*" и ".*?"

Они очень похожи и, да, *почти* одинаковы. Оба ищут от одной кавычки до другой.

Различие здесь в символе точка `'.'`. Как мы помним, точка `'.'` обозначает *любой символ, кроме перевода строки*.

А `[^"]` – это *любой символ, кроме кавычки `'"'`.

Получается, что первый регэксп `"[^"]*"` найдёт закавыченные строки с `\n` внутри, а второй регэксп `".*?"` – нет.

Вот пример:

```
alert( '"многострочный \n текст"'.match(/"[^"]*"/) ); // найдёт
alert( '"многострочный \n текст"'.match(/".*?"/) ); // null (нет совпадений)
```

К условию

Найти HTML-комментарии

Нужно найти начало комментария `<!--`, затем всё до конца `-->`.

С первого взгляда кажется, что это сделает регулярное выражение `<!--.*?-->` – квантификатор сделан ленивым, чтобы остановился, достигнув `-->`.

Однако, точка в JavaScript – любой символ, *кроме* конца строки. Поэтому такой регэксп не найдёт многострочный комментарий.

Всё получится, если вместо точки использовать полностью «всеядный» `[\s\S]`.

Итого:

```
var re = /<!--[\s\S]*?-->/g;
var str = '.. <!-- Мой -- комментарий \n тест --> ..  <!----> .. ';
alert( str.match(re) ); // '<!-- Мой -- комментарий \n тест -->', '<!---->'
```

К условию

Найти HTML-теги

Начнём поиск с `<`, затем один или более произвольный символ, но до закрывающего «уголка»: `.+?>`.

Проверим, как работает этот регэксп:

```
var re = /<.+?>/g;
```

```
var str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(re) ); // <> <a href="/">, <input type="radio" checked>, <b>
```

Результат неверен! В качестве первого тега регэксп нашёл подстроку `<> <a href="/">`, но это явно не тег.

Всё потому, что `.+?` – это «любой символ (кроме `\n`), повторяющийся один и более раз до того, как оставшаяся часть шаблона совпадёт (ленивость)».

Поэтому он находит первый `<`, затем есть «всё подряд» до следующего `>`.

Первое совпадение получается как раз таким:

```
<.............>
<> <a href="/"> <input type="radio" checked> <b>
```

Правильным решением будет использовать `<[^>]+>`:

```
var re = /<[^>]+>/g

var str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(re) ); // <a href="/">, <input type="radio" checked>, <b>
```

Это же решение автоматически позволяет находится внутри тегу символу `\n`, который в класс точка `.` не входит.

К условию

Скобочные группы

Найдите цвет в формате #abc или #abcdef

Регулярное выражение для поиска 3-значного цвета вида `#abc`: `/#[a-f0-9]{3}/i`.

Нужно добавить ещё три символа, причём нужны именно три, четыре или семь символов не нужны. Эти три символа либо есть, либо нет.

Самый простой способ добавить – просто дописать в конец регэкспа: `/#[a-f0-9]{3}([a-f0-9]{3})?/i`

Можно поступить и хитрее: `/#([a-f0-9]{3}){1,2}/i`.

Здесь регэксп `[a-f0-9]{3}` заключён в скобки, чтобы квантификатор `{1,2}` применялся целиком ко всей этой структуре.

В действии:

```
var re = /#([a-f0-9]{3}){1,2}/gi;

var str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(re) ); // #3f3 #AA0ef #abc
```

В последнем выражении `#abcd` было найдено совпадение `#abc`. Чтобы этого не происходило, добавим в конец `\b`:

```
var re = /#([a-f0-9]{3}){1,2}\b/gi;

var str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(re) ); // #3f3 #AA0ef
```

К условию

Разобрать выражение

Регулярное выражение для числа, возможно, дробного и отрицательного: `-?\d+(\.\d+)?`. Мы уже разбирали его в предыдущих задачах.

Оператор – это `[-+*/]`. Заметим, что дефис `-` идёт в списке первым, так как на любой позиции, кроме первой и последней, он имеет специальный смысл внутри `[...]`, и его понадобилось бы экранировать.

Кроме того, когда мы оформим это в JavaScript-синтаксис `/.../` – понадобится заэкранировать слэш `/`.

Нам нужно число, затем оператор, затем число, и необязательные пробелы между ними.

Полное регулярное выражение будет таким: `-?\d+(\.\d+)?\s*[-+*/]\s*-?\d+(\.\d+)?`.

Чтобы получить результат в виде массива, добавим скобки вокруг тех данных, которые нам интересны, то есть – вокруг чисел и оператора: `(-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?)`.

Посмотрим в действии:

```
var re = /(-?\d+(\.\d+)?)\s*([-+*\/])\s*(-?\d+(\.\d+)?)/;

alert( "1.2 + 12".match(re) );
```

Итоговый массив будет включать в себя компоненты:

- `result[0] == "1.2 + 12"` (вначале всегда полное совпадение)
- `result[1] == "1"` (первая скобка)
- `result[2] == "2"` (вторая скобка – дробная часть `(\.\d+)?`)
- `result[3] == "+"` (...)
- `result[4] == "12"` (...)
- `result[5] == undefined` (последняя скобка, но у второго числа дробная часть отсутствует)

Нам из этого массива нужны только числа и оператор. А, скажем, дробная часть сама по себе – не нужна.

Уберём её из запоминания, добавив в начало скобки `?:`, то есть: `(?:\.\d+)?`.

Итого, решение:

```
function parse(expr) {
  var re = /(-?\d+(?:\.\d+)?)\s*([-+*\/])\s*(-?\d+(?:\.\d+)?)/;

  var result = expr.match(re);

  if (!result) return;
  result.shift();

  return result;
}

alert( parse("-1.23 * 3.45") );  // -1.23, *, 3.45
```

К условию

Обратные ссылки: \n и $n

Найдите пары тегов

Открывающий тег – это `\[(b|url|quote)\]`.

Для того, чтобы найти всё до закрывающего – используем ленивый поиск `[\s\S]*?` и обратную ссылку на открывающий тег.

Итого, получится: `\[(b|url|quote)\][\s\S]*?\[/\1\]`.

В действии:

```
var re = /\[(b|url|quote)\][\s\S]*?\[\/\1\]/g;

var str1 = "..[url]http://ya.ru[/url]..";
var str2 = "..[url][b]http://ya.ru[/b][/url]..";

alert( str1.match(re) ); // [url]http://ya.ru[/url]
alert( str2.match(re) ); // [url][b]http://ya.ru[/b][/url]
```

Для закрывающего тега `[/1]` понадобилось дополнительно экранировать слеш: `\[\/1\]`.

К условию

Альтернация (или) |

Найдите языки программирования

Сначала неправильный способ.

Если перечислить языки один за другим через `|`, то получится совсем не то:

```
var reg = /Java|JavaScript|PHP|C|C\+\+/g;

var str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,Java,PHP,C,C
```

Как видно, движок регулярных выражений ищет альтернации в порядке их перечисления. То есть, он сначала смотрит, есть ли `Java`, а если нет – ищет `JavaScript`.

Естественно, при этом `JavaScript` не будет найдено никогда.

То же самое – с языками `C` и `C++`.

Есть два решения проблемы:

1. Поменять порядок, чтобы более длинное совпадение проверялось первым: `JavaScript|Java|C\+\+|C|PHP`.
2. Соединить длинный вариант с коротким: `Java(Script)?|C(\+\+)?|PHP`.

В действии:

```
var reg = /Java(Script)?|C(\+\+)?|PHP/g;

var str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,JavaScript,PHP,C,C++
```

К условию

Найдите строки в кавычках

Решение задачи: `/"(\\.|[^"\\])*"/g`.

То есть:

- Сначала ищем кавычку `"`
- Затем, если далее слэш `\\` (удвоение слэша – техническое, для вставки в регэксп, на самом деле там один слэш), то после него также подойдёт любой символ (точка).
- Если не слэш, то берём любой символ, кроме кавычек (которые будут означать конец строки) и слэша (чтобы предотвратить одинокие слэши, сам по себе единственный слэш не нужен, он должен экранировать какой-то символ) `[^"\\]`
- ...И так жадно, до закрывающей кавычки.

В действии:

```
var re = /"(\\.|[^"\\])*"/g;
var str = '.. "test me" .. "Скажи \\"Привет\\"!" .. "\\r\\n\\\\" ..';

alert( str.match(re) ); // "test me","Скажи \"Привет\"!","\r\n\\"
```

К условию

Найдите тег style

Начало шаблона очевидно: `<style`.

А вот дальше... Мы не можем написать просто `<style.*?>`, так как `<styler>` удовлетворяет этому регэкспу.

Нужно уточнить его. После `<style` должен быть либо пробел, после которого может быть что-то ещё, либо закрытие тега.

На языке регэкспов: `<style(>|\s.*?>)`.

В действии:

```
var re = /<style(>|\s.*?>)/g;

alert( "<style> <styler> <style test>".match(re) ); // <style>, <style test>
```

К условию

Начало строки ^ и конец $

Регэксп ^$

Нам нужна строка, которая начинается – и тут же кончается. То есть, пустая.

Или, если быть ближе к механике регэкспов, то движок сначала будет искать в тексте начальную позицию `^`, а как только найдёт её – будет ожидать конечной `$`.

Заметим, что и `^` и `$` не требуют наличия символов. Это – проверки. В пустой строке движок сначала проверит первую, а потом – вторую – и зафиксирует совпадение.

К условию

Проверьте MAC-адрес

Двузначное шестнадцатиричное число – это `[0-9a-f]{2}` (с учётом флага `/i`).

Нам нужно одно такое число, и за ним ещё 5 с двоеточиями перед ними: `[0-9a-f]{2}(:[0-9a-f]{2}){5}`

И, наконец, совпадение должно начинаться в начале строки и заканчиваться – в конце. То есть, строка целиком должна подходить под шаблон. Для этого обернём шаблон в `^...$`.

Итого, в действии:

```
var re = /^[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}$/i;

alert( re.test('01:32:54:67:89:AB') ); // true

alert( re.test('0132546789AB') ); // false (нет двоеточий)

alert( re.test('01:32:54:67:89') ); // false (5 чисел, а не 6)

alert( re.test('01:32:54:67:89:ZZ') ) // false (ZZ в конце)
```

К условию

Intl: интернационализация в JavaScript

Отсортируйте массив с буквой ё

Здесь подойдут стандартные параметры сравнения:

```
var animals = ["тигр", "ёж", "енот", "ехидна", "АИСТ", "ЯК"];

var collator = new Intl.Collator();
animals.sort(function(a, b) {
  return collator.compare(a, b);
});

alert( animals ); // АИСТ,ёж,енот,ехидна,тигр,ЯК
```

А вот, что было бы при обычном вызове `sort()`:

```
var animals = ["тигр", "ёж", "енот", "ехидна", "АИСТ", "ЯК"];

alert( animals.sort() ); // АИСТ,ЯК,енот,ехидна,тигр,ёж
```

К условию