Кирилл Сухов

NODE.JS
Путеводитель по технологии

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

К. К. Сухов

Node.js. Путеводитель по технологии

Москва, 2015

УДК 004.738.5:004.4Node.js
ББК 32.973.202-018.2
С91

Сухов К. К.
С91 Node.js. Путеводитель по технологии. – М.: ДМК Пресс, 2015. – 416 с.: ил.

ISBN 978-5-97060-164-8

За последние несколько лет платформа Node.js стремительно повысила свой статус от экспериментальной технологии до основы для серьезных промышленных проектов. Тысячи программистов оценили возможность построения достаточно сложных, высоко нагруженных приложений на простом, элегантном и, самое главное, легковесном механизме.

Все эти скучные слова правдивы, но на самом деле не это главное. Прежде всего Node.js – это совершенно увлекательная и захватывающая вещь, с которой по-настоящему интересно работать!

Есть одна проблема – невозможно рассказывать про использование Node.js в отрыве от остальных технологий современной веб-разработки (и Highload-разработки). Я и не стал этого делать, дав обзор инструментов, без которых сегодня трудно обойтись. Прежде всего это чудесный язык JavaScript, и в книге рассказано о новинках в его последней и будущей спецификациях (EcmaScript 5 и 6). Кроме того, дается краткое введение в большинство связанных веб-технологий – от NoSQL-хранилищ данных (Memcached, MongoDB, Redis) до CSS-препроцессоров и MVC JavaScript-фреймворков. Конечно, эту книгу нельзя рассматривать как полноценный учебник по MongoDB, LESS или EcmaScript 6, Dart или CoffeScript, но в ней дано основное представление об этих довольно интересных вещах, вполне достаточное для начала работы.

УДК 004.738.5:004.4Node.js
ББК 32.973.202-018.2

ISBN 978-5-97060-164-8

Содержание

Благодарности

В первую очередь огромное спасибо Райану Далу, TJ Holowaychuk и другим разработчикам, подарившим мне отличный инструмент, благодаря которому я могу на своем любимом языке писать полноценные серверные приложения.

Огромное спасибо Дмитрию Мовчану, который буквально вынудил меня взяться за этот труд и поддерживал в процессе. Спасибо издательству «ДМК Пресс» и отдельно Галине Синяевой.

Спасибо редакции журнала «Системный администратор», Галине Положенец, Полине Гвоздь – без вас мне было бы очень трудно.

Особая благодарность тем, кто помогал мне в работе над книгой, – Александу Календареву, Валентину Синицину, Александру Слесареву, Алексею Вторникову.

Спасибо тем специалистам, советы и рекомендации которых помогли в работе над материалом. Это Дмитрий Бородин, Андрей Шетухин, Олег Царев, Илья Кантор, Евгений Зиндер, Борис Богданов, а также Константин Полянский, Александр Байрак, Артем Демин «and last but not least» – Андрей Бешков. Огромное спасибо Александру Смирнову и сообществу phpclub.ru.

Спасибо моему сыну Роману и моим друзьям за поддержку.

И самое главное спасибо – моей музе и невесте Лиде. Любимая, без твоего терпения и поддержки вообще бы ничего не состоялось.

Вступление

О Node.js

Node.js – это серверная JavaScript-платформа, предназначенная для создания масштабируемых распределенных сетевых приложений, использующая событийно-ориентированную архитектуру и неблокирующее асинхронное взаимодействие. Она основана на JavaScript-движке V8 и использует этот же JavaScript для создания приложений. Node.js хоть и достигла в своем развитии только цифр 0.10.30 в номере версии, но уже активно используется в реальных проектах.

Помимо эффектной асинхронной модели работы, неблокирующих процессов, высокой производительности, Node.js делает то, что считалось принципиально невыполнимым, – дает возможность разработчику создавать как server-side/backend-, так и frontend-приложения, пользуясь единой технологией! Да-да, теперь на JavaScript можно написать обработчик http-запросов, да что там, настоящий, полнофункциональный веб-сервер! Можно работать с SQL- (и NoSQL-) базами данных, сетью, файловой системой. Еще недавно все это казалось трудно достижимым.

На самом деле, чего там скрывать, когда 5 лет назад новая технология только появилась, автору этих строк и многим его коллегам она казалась забавной игрушкой – интересной, но без шансов промышленного применения. Я рад, мы ошибались – Node.js доказала свою состоятельность, и сейчас её «боевое» использование – не экзотика, а нормальная практика, особенно в пресловутых высоконагруженных проектах. Node.js сейчас тем или иным образом используют такие известные участники IT-рынка, как Groupon, SAP, LinkedIn, Microsoft, Yahoo!, Walmart, PayPal. По-моему, достойная компания, к которой не грех присоединиться.

Об этой книге

Мне всегда нравились рассказы о новых технологиях, тесно завязанные на практические примеры, на реальный, работающий код. И сам я попытался создать нечто подобное. В книге не слишком много общих фраз, но очень много кода и пояснений к нему.

Сначала, после небольшого вводного обзора, мы установим Node.js, начнем работу и даже, с места в карьер, напишем первый Node.js-сайт (это не займет много времени). Далее мы подробно познакомимся

с ядром Node.js, освоим его основные компоненты – событийную модель, процессы, понятие буфера, таймеры.

Далее мы познакомимся с понятием модуля, освоим менеджер пакетов Node.js – Node Packaged Manager и даже напишем собственный Node.js-модуль (а то и пару).

Затем, изучив работу с файлами и потоками (Stream), примемся за сетевую ипостась платформы. Мы узнаем, как на Node.js можно создавать TCP/UDP/HTTP-серверы и организовать работу сети. В следующем разделе мы будем изучать работу Node.js с хранилищами данных, как реляционными (mysql), так и NoSql – Memcached, Redis, MongoDB. Если вы до этого мало имели дело с NoSQL, не беда, мы постараемся более или менее подробно разобрать работу каждого хранилища.

Далее рассказывается о различных реализациях WebSocket-сервера и инструментах Node.js для работы с протоколом Websocket – ws, socket.io. Веб-сокеты – это уже стандарт современного веба, и платформа Node.js имеет все средства для их воплощения.

Следующий раздел посвящен клиентской стороне веб-разработки – изучаем шаблонизаторы и CSS-препроцессоры и выбираем лучший. Выбрать есть из чего – рассмотрена работа Mustache, EJS, Jade, LESS, Stylus.

Вторая половина книги возвращает нас на сторону сервера, но уже на новом уровне – будут рассмотрены midware-фреймворк Connect, платформа Express, создано первое RESTful API Node.js-приложение. В разделе «Практика разработки приложений Node.js» будут рассмотрены средства отладки, профилирования, сборки и развертывания Node.js-приложений. Отдельная глава посвящена инструментам тестирования – модулям assert, chai, should, фреймворкам Mocha, Jasmine.

Последний раздел посвящен будущему основного инструмента платформы Node.js – языка JavaScript. Рассмотрены такие его модификации/заменители, как CoffeeScript, TypeScript и Dart, а также подробно рассмотрены настоящие и будущие нововведения языка, привнесенные стандартами EcmaScript5 и EcmaScript6 (Harmony), – на платформе Node.js сейчас они доступны почти все.

Для нормального восприятия книги достаточно начальных знаний языка JavaScript, общего представления об устройстве Всемирной сети и желания разобраться в самых современных веб-технологиях.

Предыстория

За что я люблю веб-разработку? За то, что веб развивается прямо у нас на глазах. Это живая среда, которая растет бурно и зачастую непредсказуемо, меняясь и меняя нашу жизнь, увлекая ее за собой. Я отлично помню то время, когда получить на свой компьютер простую HTML-страничку с какого-нибудь заморского сервера было маленьким чудом, а сейчас... даже описывать что-либо не имеет смысла. Сегодня www – это значительная часть нашей рабочей, личной и социальной сферы, и похоже, это только начало.

Впрочем, это все лирика, нас, суровых технарей, больше интересует именно техническая сторона этой непрерывной веб-революции.

Веб-страница, веб-сервис, веб-приложение

Не будем сейчас особо вдаваться в детали, но если кратко: всё развивалось в несколько этапов. Сначала были плоские HTML-странички, и задачей веб-сервера было только исправно отдавать их клиентам. Затем появился cgi и аналогичные механизмы, страницы стали генериться с помощью различных интерпретаторов (perl, php и т. д.), затем они (интерпретаторы) стали стандартом, как модули к веб-серверу. Появилась возможность использовать базы данных, было написано множество библиотек и фреймворков, выводящих веб-разработку на новый уровень.

Параллельно развивалась и клиентская часть www. HTML-страницы с помощью JavaScript и DOM перестали быть статичными, они «ожили», превратив с появлением технологии Ajax сайты из набора связанных страниц в полноценные приложения.

А потом... а потом наступила эра highload, и многих возможностей старого веба стало не хватать, причем нехватка эта носила принципиальный, архитектурный характер. Требовались новые подходы, такие как использование NoSQL-хранилищ данных, очередей сообщений, балансировки нагрузки и асинхронного доступа.

Асинхронность как необходимость

Что такое асинхронно событийная модель и зачем она нужна? Строго говоря, второй вопрос не требует ответа. Если у вас, при разработке вашего проекта, не возникла потребность в неблокирующем асинхронном взаимодействии, то, скорее всего, это вам просто не нужно.

Пока. Но если вы создаете систему, которая поддерживает взаимодействие по множеству одновременных подключений, причем взаимодействие двунаправленное, то рано ли поздно вы придёте к необходимости организации асинхронного доступа.

Да, именно доступа – под асинхронностью и понимают асинхронный доступ к ресурсам – файлам, сокетам, данным. Как он происходит по классической схеме работы веб-приложения? Все просто и состоит из нескольких этапов:

- подключиться к ресурсу (источнику данных);
- считать из него первую порцию данных;
- ждать, пока на него не будет готова вторая порция данных;
- считать вторую порцию данных;
- ждать третью порцию данных;
- ...; завершить считывание данных;
- разорвать соединение и продолжить работу.

При этом при исполнении программы возникает «блокировка», вызванная тем, что установка соединения с ресурсом и (особенно) чтение из него данных требуют времени. Во время этой блокировки поток исполнения простаивает, и это, безусловно не самое рациональное использование ресурсов. Для преодоления данной проблемы была разработана хорошо всем известная многопоточная модель. Суть ее работы – в том, что приложение создает некоторое количество потоков, передавая каждому из них задачу и данные для обработки. Все задачи выполняются параллельно, и если они не имеют общих данных и не нуждаются в синхронизации, их обработка происходит достаточно быстро. В такую модель работы почти идеально укладывается обработка веб-сервером запросов от многочисленных клиентов (браузеров, запрашивающих веб-страницы).

Собственно, все еще самый популярный в Интернете веб-браузер – Apache – так и работает. На каждый http-запрос создаётся отдельный поток (ну, если точнее, может просто забираться свободный поток из пула, но это уже детали реализации), в котором работает, например, php-скрипт. Все хорошо, и такая модель способна обеспечить стабильную работу при относительном множестве запросов. Но... но вот Hihgload ведь. Что, если одновременных запросов будет не просто множество, а тысяча? Десять тысяч? Сто тысяч? Да в общем ничего особенного – просто создаваемые потоки, в конце концов, заполнят всю оперативную память и уронят север. С этим, конечно, можно бороться, наращивая объем памяти, вводя в бой дополнительные ресурсы, но есть и другие способы справляться с высокими нагрузками.

Асинхронная модель решает проблемы доступа принципиально по-другому. Она построена на зацикленной очереди событий (event-loop). При возникновении некоторого события (пришел запрос, ответ от базы данных, считалась порция данных из файла) оно помещается в конец очереди. Поток исполнения, обрабатывающий этот цикл, выполняет код, связанный со следующим событием, и, в свою очередь, помещает его в конец. Это происходит до тех пор, пока очередь не опустеет.

Реализации такой схемы работы написаны для различных языков программирования. Это фреймворки Perl AnyEvent, Ruby EventMachine, Python Twisted и веб-сервер Tornado. По такой же схеме работает и Node.js, имея в своем воплощении event loop ряд особенностей, которые, возможно, позволят этой среде получить ключевое конкурентное преимущество. Все их мы со временем постараемся рассмотреть.

Решения – Twisted, Tornado и другие

Twisted – событийно-ориентированный сетевой фреймворк, написанный на Python. Он поддерживает все основные сетевые протоколы (HTTP, TCP, UDP, SSL/TLS, IP Multicast, Unix domain sockets и т. д.). Основной модуль фреймворка, twisted.internet.reactor, представляет реализованный собой тот самый цикл событий (event loop), который занимается выполнением обработчиков событий и возможных ошибок.

По умолчанию реактор веб-сервера (как и фреймворк в целом) использует механизм распределения событий для неблокирующих сокетов. **Tornado** – это расширяемый, неблокирующий серверный веб-фреймворк, написанный тоже на Python. Он разработан для использования в проекте FriendFeed, который был приобретен Facebook в 2009 году, после чего исходные коды Tornado были открыты. Tornado был создан для обеспечения высокой производительности, и он действительно отлично справляется с той задачей на легких запросах, но вот на сложных проектах требует дополнительных компонентов – прежде всего это веб-сервер.

Ключевые различия между этими двумя платформами следующие: Twisted – это набор библиотек для асинхронного программирования на питоне, сложный и многофункциональный; Tornado – это веб-сервер, который может запускать приложения (как асинхронные, так и самые обычные).

Ну, хватит с Python, есть что-нибудь на других языках? Конечно! **EventMachine** – легкий фреймворк, использующий асинхронный, событийно-ориентированный механизм обработки сетевых соединений, предназначенный для сетевого взаимодействия. EventMachine написан на языке Ruby и имеет в своем составе довольно много средств – веб-сервер, механизмы подписки и отложенного выполнения, набор стандартных классов. Это не единственная платформа на Ruby, но самая известная.

Впрочем, и языки – ветераны веб-разработки имеют собственные решения для асинхроной модели. **AnyEvent** – Perl-модуль, популярный фреймворк, реализующий так называемую концепцию событийно-ориентированного программирования (СОП) в Perl. Близок к рассматриваемой нами асинхронной событийной модели и является наиболее оптимальным выбором для программирования современных сетевых приложений в Perl.

И, надо сказать, это почти все. Нельзя сказать, что перечисленные решения не получили признания, но все же не сильно распространены, однако они достаточно нишевые, и это естественным образом ограничивает их применение.

Странный язык – JavaScript

Этот странный, рожденный «на коленке» язык программирования собрал огромное количество критики и нареканий. И это естественно – язык никогда не проектировался для такого применения и вообще развивался «от практики». Поначалу и применение его на веб-страницах (и только на них!) ограничивалось простыми эффектами вроде подсветки кнопок при наведении мыши и тому подобными мелкими «красивостями». Но постепенно JavaScript брал на себя все большую роль по взаимодействию www и человека, а с появлением суммы технологий, обозначенных красивым термином Ajax, эта роль стала вообще ключевой. Еще интенсивней javaScript стал использоваться с распространением библиотек/фреймворков, делающих разработку сложных сценариев, приложений (да-да, речь идет уже о JavaScript-приложениях!) и веб-интерфейсов простым и приятным занятием. Я говорю прежде всего о библиотеке Prototype, заложившей основы нескольких JavaScript-фреймворков, о jQuery, ставшей практически стандартом разработки, о Ext.js, задающем новый уровень веб-интерфейсов, наконец, о MVC-среде – Backbone.js, предоставляющей каркас для создания полноценных REST-приложений.

Но этого было мало. Новый стандарт языка разметки веб-страниц – HTML5 – подразумевает наличие множества JavaScript API, которые, не являясь частью языка, творят настоящие чудеса, как на веб-странице, так и за ее пределами. Я говорю про Geolocation API, WebRTC, WebGL и подобные технологии, находящиеся сейчас на переднем крае развития www.

Обилие возможностей породило Javascript-безумие, которое длится и по сей день, – на этом языке стали писать все – 3D-игры, компиляторы, DE-окружения и даже операционные системы! Правда, пока эти творения скорее более ориентированы на демонстрацию возможностей, чем на практическое использование, но зато демонстрация получается ну очень впечатляющая.

И вот появилась платформа Node.js – JavaScript проник на сервер! Похоже, скоро в Интернете ничего, кроме JavaScript, и не останется.

И тут я бы хотел рассказать про этот удивительный язык, но, во-первых, это точно займет много времени, а во-вторых, рядовой веб-разработчик JavaScript, как правило, знает. Ну или думает, что знает.

В любом случае, я хочу прояснить всего два аспекта этого языка. Они очень важны для дальнейшего изложения материала, но если замыкания и ООП-практика JavaScript вам хорошо знакомы, смело пропускайте следующие два раздела.

Волшебные замыкания

Замыкание (*closure*) – это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции, причем в качестве её параметров, то есть функция, которая ссылается на некие сущности в своём контексте.

Что такое замыкание?

Ничего не понятно? Хм. Я бы тоже не понял. Можно сказать, что замыкание – это особый вид функции, которая определена в теле другой функции и создаётся каждый раз во время её выполнения. При этом вложенная внутренняя функция может содержать ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра замыкания, с новыми ссылками на переменные внешней функции.

Замыкания поддерживаются в таких языках, как Ruby, Python, Scheme, Haskell, Java (с помощью анонимных классов). Не так давно их реализация стала возможна в C++ (стандарт C++11) и PHP (с версии 5.3). Конечно, многие программисты при написании JavaScript-сценариев обходятся без замыканий вовсе, но это не значит, что мы должны следовать их примеру. Во-первых, замыкания здорово упрощают жизнь, во-вторых, некоторые вещи без их применения просто невозможно реализовать. В современном JavaScript замыкания – важный элемент языка, без которого нам не обойтись, поэтому давайте разберемся, что это и как с этими конструкциями обращаться.

Начнем с примера:

```
function localise(greeting) {
    return function(name) {
        console.log(greeting + ' ' + name);
    };
}

var english = localise('Hello');
var russian = localise('Привет');

english('closure');
russian('closure');
```

Такой код выведет в консоль:

```
>Hello closure
>Привет closure
```

Фактически мы создали две разные функции, в которых фраза приветствия «замкнута» вмещающей функцией. Но это еще не все. Как известно, в JavaScript областью видимости локальной переменной является тело функции, внутри которой она определена. Если вы объявляете функцию внутри другой функции, первая получает доступ к переменным и аргументам последней и сохраняет их даже после того, когда внешняя функция отработает:

```
function counter(n) {
    var count = n;
    return function() {
        console.log(count++);
    };
}
var ct = counter(5);
ct(); //5
ct(); //6
ct(); //7
ct(); //8
ct(); //9
```

Функция ct() возвращает количество собственных вызовов, причем начальное значение задается при ее создании.

Магия? Да ничего подобного. Вернее, да, магия, но она заложена в самой концепции JavaScript. Возьмем, например, такой код:

```
var i = 5
function plusOne(){
     i++;
     return i;
}
console.log(plusOne()); //6
console.log(plusOne()); //7
console.log(i); //7
```

Тут, наверное, никого не смущает, что функция plusOne() имеет доступ к глобальной переменной i. А ведь это тоже замыкание, только по глобальной области видимости. Переменные же, объявленные внутри функций, имеют область видимости, ограниченную рамками этой функции, и замыкание происходит именно по ней – пусть даже и сама внешняя функция завершена. Это делает возможными следующие конструкции:

```
var outerVar = 4;
var handle;
function outer() {
    var innetVar = 5;
    function inner() {
        console.log(outerVar);
         console.log(innetVar);
    };
    handle = inner;
}
outer();
handle(); //4
          //5
```

(Разумеется, если **handle()** вызывать до **outer()**, скрипт завершится с ошибкой, ведь handle еще не ссылается на функцию.)

Применение замыканий

Самым очевидным применением замыканий будет конструкция вроде «фабрики функций» (не буду называть это Abstact Factory, дабы не вызвать гнев приверженцев чистоты терминологии):

```
function createFunction(n) {
   return function(x) {
      return x*n;
   }
}

var twice = createFunction(2);
var threeTimes = createFunction(3);
var fourTimes = createFunction(4);
console.log(twice(4)+" "+fourTimes(5)); //8 20
```

Это, конечно, очень простой пример, но оцените саму возможность. Практически мета-программирование!

Следующее важное применение – сохранение состояния функции между вызовами. Мы уже рассматривали это на примере функции, возвращающей количество вызовов. Ее можно сделать еще проще:

```
var ct = (function counter() {
    var count = 5;
    return function() {
        console.log(count++);
    };
})();
ct(); //5
ct(); //6
ct(); //7
```

Так у нас нет необходимости отдельно вызывать внешнюю функцию.

Как известно, в JavaScript отсутствуют модификаторы доступа, а оградить переменную от нежелательного влияния иногда ой как необходимо. Опять же использование замыкания нам может организовать это не хуже, чем модификатор private:

```
function counter() {
    var count = 0;
    this.getCount = function(){
      return count;
    }
    this.setCount = function(n){
      count = n;
    }
    this.incrCount = function(){
      count++;
    }
}

var ct = new counter();
ct.setCount(5);
ct.incrCount();
console.log(ct.getCount()); //6
console.log(ct.count); //undefined
```

Как видите, цель достигнута – прямой доступ к «приватной» переменной count невозможен. Её можно получить, только используя методы counter, которые даже могут быть приватными!

```
function counter() {
   var count = 0;
   function setCount(n) {
     count = n;
   }
   return {
      safeSetCount: function(n) {
      if(n!=13){
         setCount(n);
      } else {
         console.log("Bad number!");
      }
      },
    getCount: function(){
      return count;
    },
    incrCount: function(){
      count ++ ;
    }
```

```
  }
}
ct = new counter();
ct.safeSetCount(5);
ct.safeSetCount(13); //Bad number!
ct.incrCount(13);
console.log(ct.getCount()); //6
```

Тут попытка вызвать метод **ct.SetCount()** снаружи немедленно приведет к ошибке.

ООП в JavaScript

Поддерживает ли JavaScript парадигму объектно-ориентированного программирования? С одной стороны, да, безусловно, с другой – при изучении реализации ООП в JavaScript большинство специалистов, привыкших к разработке на C++ или Java, в лучшем случае отмечают своеобразие этой самой реализации. Ну действительно, есть объекты, но где классы? как наследовать? А инкапсулировать? Как с этим вообще можно работать?

На самом деле работать можно, и даже очень эффективно, просто надо помнить, что ООП в JavaScript прототипное. То есть в нем отсутствует понятие класса, а наследование производится путём клонирования существующего экземпляра объекта – прототипа. С этим обстоятельством не надо бороться, его надо использовать.

Объекты в JavaScript представляют собой в первую очередь ассоциативный массив. То есть набору строковых ключей соответствует одно любое значение. Значением массива может быть и функция, в таком случае она является методом этого объекта.

Как мы обычно создаем «с нуля» объект в JavaScrit? Например, так:

```
var user_vasya = { name: "Vasya",
        id: 51,
        sayHello: function(){
              console.log("Hello "+ this.name);
              }
        };
user_vasya.sayHello(); // Hello Vasya
```

Или так:

```
var user_masha = {};
user_masha.name = "Masha";
user_masha.uid = 5;
user_masha.sayHello = function(){ console.log("Hello "+ this.name);}
user_masha.sayHello();  // Hello Masha
```

Все просто и понятно, но для полноценной ООП-разработки явно чего-то не хватает. Представьте, что нам нужно получить полсотни однотипных объектов. Ну, или добавить к объекту некоторую инициализацию – установку начальных параметров. Иными словами, я говорю о необходимости конструктора. Его тут нет.

Попробуем создать объект другим способом. Мы знаем, что в JavaScript функция – это всегда объект. Причем объект первого класса (first-class object), сущности, которые могут быть переданы как параметр, возвращены из функции, присвоены переменной. Итак, объект User:

```
var User = function(name, id){
            this.name = name;
            this.id = id;
              this.sayHello = function (){ console.log("Hello "+ this.name);}
        }
```

Это объект? Проверим:

```
console.log(typeof User); // function

var John =  User("John",51);
console.log(typeof John); // undefined

John.sayHello() // John is undefine
```

Собственно, ничего удивительного – чтобы создать объект, нужно воспользоваться специальным методом, хорошо нам знакомым по другим ООП-языкам:

```
var John =  new User("John",51);
console.log(typeof John); // object

John.sayHello() // Hello John
```

Теперь все в порядке. Чтобы это стало совсем очевидно, мы можем в JavaScript-консоли браузера Google Chrome посмотреть доступные свойства объекта John (рис. 1), и надо же – среди них метод consructor, значение которого – тело функции User (рис. 2).

В первом приближении наша цель достигнута, но хотелось бы еще и расширения функционала базового э... нет, не класса, в данном случае конструктора. Это делается с помощью объекта prototype, который есть у каждого JavaScript-объекта и принадлежит полю consructor:

```
var User = function(name, id){
            this.name = name;
            this.id = id;
              this.sayHello = function (){ console.log("Hello "+ this.name);}
        }
User.prototype.sayHi = function(){console.log("Hi "+ this.name);}
John.sayHi(); // Hi John
```

Рис. 1 ❖ Просматриваем свойства объекта

Новый метод создан и работает. Обратите внимание: мы создали его уже после создания нового объекта – он все равно в этом объекте появится. Важно понимать, что объект prototype является дополнением к конструктору. При обращении к некоторому свойству или методу объекта сначала он будет искаться в конструкторе, потом в объекте prototype:

```
var User = function(name, id){
            this.name = name;
            this.id = id;
              this.sayHello = function (){ console.log("Hello "+ this.name);}
            }
User.prototype.sayHello() = function(){console.log("Hi "+ this.name);}
var John =  new User("John",51);
John.sayHello(); // Hello John
```

Еще один важный момент – метод, созданный посредством объекта Prototype, принадлежит конструктору и является общим для всех

Рис. 2 ❖ Тело конструктора

объектов. Если нам надо переопределить метод (поле) созданного объекта или добавить ему собственный, то следует обращаться непосредственно к нему:

```
var John =  new User("John",51);
John.sayHello =  function (){ console.log("Preved "+ this.name);}
John.sayHello(); // Preved  John
```

А с помощью поля constructor можно создавать новые экземпляры объекта, не обращаясь напрямую к функции-конструктору, в том числе и за пределами его видимости:

```
User.prototype.sayHello = function(){console.log("Hi "+ this.name);}
var John =  new User("John",51);
……..……
var Ian =  new John.constructor('Ian', 52);
Ian.sayHello(); // Hi Ian
```

Напомним: сам prototype представляет собой объект, а это, в частности, обозначает, что у него тоже есть свой prototype, а у того – свой. Эту цепочку прототипов можно использовать для реализации наследования объектов.

Вообще, проблема наследования в JavaScript всегда стояла остро, особенно для разработчиков на языках с классической моделью ООП. Достаточно часто можно наблюдать попытки решить ее «в лоб», копируя методы объекта prototype или сам объект целиком:

```
var User = function(name, id){
            this.name = name;
            this.id = id;
        }
User.prototype.sayHello = function (){ console.log("Hello!");}

var Admin = function () {};
Admin.prototype = User.prototype;
admin = new Admin();
admin.sayHello(); //Hello!
```

Но это и подобные ему решения неудовлетворительны. В частности, в результате выполнения вышеприведенного кода объекты будут просто иметь общий объект prototype, и добавление новых методов Admin автоматически добавит их в User. Можно, конечно, делать все аккуратнее, не копируя prototype целиком, но тогда возникнет другая проблема – новые методы User придется каждый раз добавлять в Admin. Принципиальную порочность такого пути можно выразить в двух строчках кода:

```
Admin.prototype.sayHello = User.prototype.sayHello;
admin = new Admin();
admin.sayHello();

console.log(admin instanceof Admin); // true
console.log(admin instanceof User); // false
```

Настоящее же наследование следует организовывать, учитывая структуру объекта JavaScript. Условием «правильного» наследования должно быть то, что прототип дочернего класса является экземпляром родительского класса. Примерно так:

```
var User = function(name, id){
            this.name = name;
            this.id = id;
        }
User.prototype.sayHello = function (){ console.log("Hello "+this.name);}

var Admin = function () {};
```

```
Admin.prototype = new User();
var admin = new Admin();
admin.name = "Ian";
admin.sayHello(); // Hello Ian

console.log(admin instanceof Admin); // true
console.log(admin instanceof User); // true

Admin.prototype.sayHi = function (){ console.log("Hi "+this.name);}
var user = new User("John", 52);
user.sayHi(); //TypeError: Object [object Object] has no method 'sayHi'
```

Ну что же, на этом, наверное, хватит про тонкости JavaScript. Я надеюсь, что мне удалось рассеять чье-то непонимание. Ну или побыть Капитаном Очевидность, на худой конец.

Явление Node

Платформа Node.js была создана в 2009 году Райном Далом (*Ryan Dahl*) в ходе исследований по созданию событийно-ориентированных серверных систем. Асинхронная модель была по причине низких накладных расходов (по сравнению с многопоточной моделью) и высокого быстродействия. Node была (и остается) построена на основе JavaScript-движка V8 с открытым исходным кодом, разработанного компанией Google в процессе работы над своим браузером Google Chrome. Это была не первая реализация V8 на стороне сервера, но технология оказалась так удачно спроектирована, что сразу же обрела большое число сторонников и энтузиастов и, как следствие, множество модулей, реализующих самый разнообразный функционал. В настоящее время разработка Node.js спонсируется основанной Райном компанией Joyent.

Хватит теории! Начало работы с Node.js

Установка Node

Инсталляция Node на компьютер – дело, на сегодняшний момент очень простое. Нужно получить необходимый вам вариант дистрибутива (в зависимости от используемой операционной системы) на странице загрузки Node.js – http://nodejs.org/download/ (рис. 3). Как видим, у нас есть и инсталляторы, и бинарные сборки под Windows, macOs X, бинарники для Linux и SunOS(!) и, разумеется, исходные тексты, к сборке из которых вам вряд ли придется прибегнуть. На MacOS или Windows установка состоит из привычных шагов мастера ("next"->"next"->...."next"->"finish", рис. 4); что касается операционной системы Linux, то Node.js входит во все распространенные дистрибутивы и устанавливается штатными средствами.

Рис. 3 ❖ Сайт проекта Node.js

После установки, для проверки работоспособности платформы, откроем консоль и наберем простую команду:

```
$ node -v
v0.10.30
```

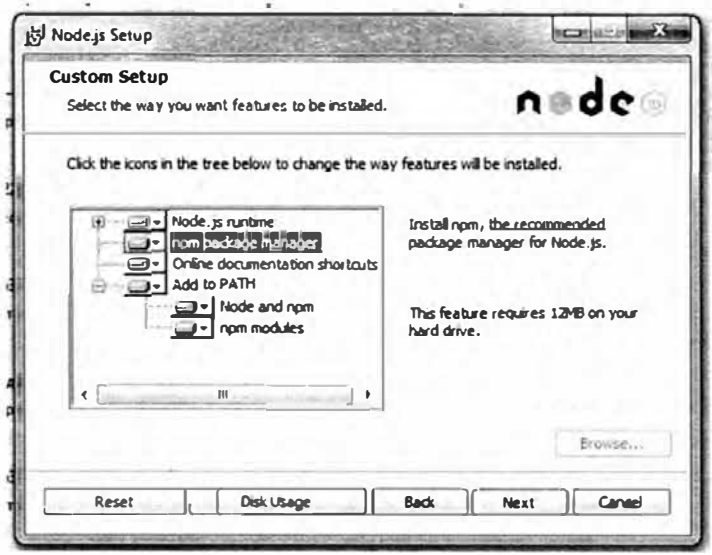

Рис. 4 ❖ Установка на Windows – муки выбора

Как видите, самая последняя версия Node.js установлена и работает.

Да, текущей версии еще очень далеко до единицы, но уже существует соглашение, по которому четная вторая цифра номера соответствует стабильным релизам Node.js. Все последующие примеры (кроме оговоренных особо) будут выполняться именно на этом релизе платформы. Надо сказать, что еще 2 года назад текст, посвященный установке Node.js, был бы больше, по крайней мере, раз в пять. С тех пор эта процедура стала достаточно тривиальной и не требует описания. Что не может не радовать.

Чтобы окончательно убедиться в работоспособности свежеинсталлированной платформы, наберем в консоли команду node без параметров:

```
$ node
>
```

Угловая скобка традиционно означает приглашение к вводу команд. Мы оказались в интерактивном режиме Node.js – REPL (Read-Eval-Print-Loop). Аналоги этой интерактивной среды можно найти в языках Lisp, Smalltalk, Perl, Python, Ruby, Haskell, Scala и др. Работать с ним просто:

```
$ node
> 1
1
> 1+2
3
> console.log("test");
test
undefined
>
```

В этом режиме в консоль просто выводится результат набранного выражения. Строчка undefined в последнем примере появляется только потому, что метод console.log() не возвращает никакого значения. Но нам в данный момент это не очень интересно, мы же собрались писать Node.js-приложения, не так ли? Для того чтобы запустить на платформе JavaScript-код, сохраненный в файле, нужно просто поставить имя этого файла в качестве параметра перед вызовом интерпретатора Node.js. И давайте уже начнем программировать. Для первого знакомства мы напишем не привычный «Hello Word», а....

Веб-сервер из пяти строк

Да, эта среда изначально предназначена для серверной веб-разработки, и веб-сервер на Node.js действительно укладывается в несколько строк кода. Вот они (файл **start.js**):

```
var http = require('http');
http.createServer(function (request, response) {
    console.log("HTTP works!");
});
http.listen(8080);
```

Собственно, веб-сервер в среде Node – это часть практически любого приложения. Понять вышеприведённый код несложно. Сначала запрашивается необходимый модуль, затем создаётся сервер и запускается на заданном порту (выбран порт 8080, так как на стандартном, 80-м обычно работает Apache или nginx). Метод **createServer()** объекта **http** принимает в качестве аргумента анонимную функцию обратного вызова, аргументами которой, в свою очередь, служат объекты **request** и **response**. Они соответствуют, как нетрудно догадаться, поступавшему HTTP-запросу и отдаваемому HTTP-ответу. Про реализацию HTTP мы поговорим подробно позже (так же, как и про структуру модулей Node.js). Пока давайте попробуем запустить то, что у нас уже есть:

```
$ node start.js
```

Если все прошло благополучно (то есть никакой реакции в консоли не появилось), откроем браузер и наберём адрес http://127.0.0.1:8080. В консоли должно появиться жизнеутверждающее сообщение:

```
$ node start.js
HTTP works!
```

говорящее о том, что обращение к веб-серверу зафиксировано. Веб-сервер работает, мы это сделали!

Вот, правда, браузеру в этом случае ничего, кроме 200-го ответа в заголовке, от сервера не придет. Но это нетрудно исправить:

```
http.createServer(function (request, response) {
   console.log("HTTP works!");
   response.writeHead(200, {'Content-Type': 'text/html'});
   response.end('<h1>Hello!</h1>');
}).listen(8080);
```

Теперь работу сервера видно и в браузере (рис. 5).

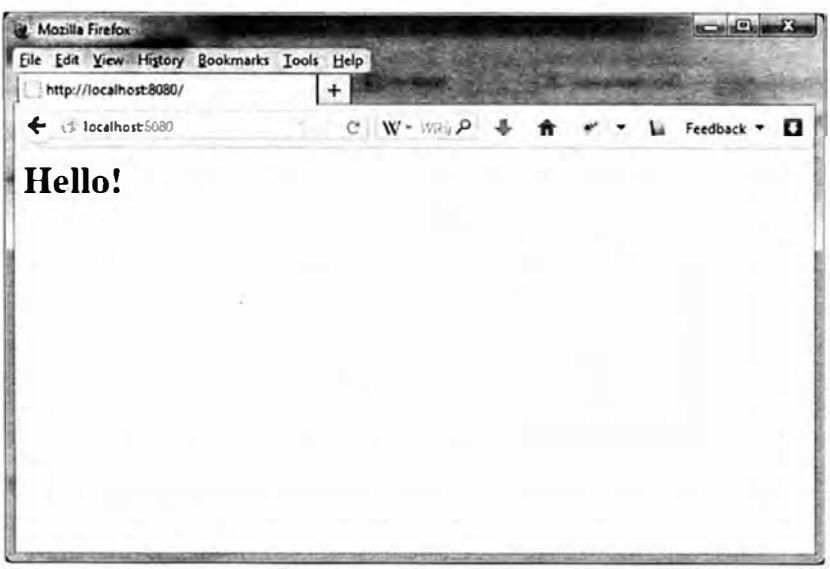

Рис. 5 ❖ Hello Node!

На этом небольшом примере видны основные приёмы программирования под node.js, основанные на её асинхронной событийной модели. В отличие от привычных cgi- или perl/php_mode-сценариев,

HTTP-запрос тут не является инициатором запуска всей программы. Напротив – JavaScript-объект создается и ждет запроса, при поступлении которого отрабатывает связанная с этим событием функция обратного вызова. В целом JavaScript поведет себя так же, как и в сценарии на HTML-странице:

```
<div onclick = "...">
```

Что нам еще нужно от веб-сервера? А, да, чтобы он отдавал веб-страницы! Ну, это тоже можно. Приготовим простую веб-страницу **page.html**:

```
<html>
   <head>
       <title>Node-page</title>
       <link rel=stylesheet href=styles.css type=text/css>
   </head>
   <body>
       <h1>Просто страница</h1>
   </body>
</html>
```

Теперь немного модифицируем северный скрипт:

```
var fs = require('fs');
var fileName = "page.html";

http.createServer(function (req, res) {
    fs.readFile(fileName, 'utf8', function(err, data) {
         if (err){
            console.log('Could not find or open file for reading\n');
         } else {
            res.writeHead(200, {'Content-Type': 'text/html'});
            res.end(data);
           }
   })
}).listen(8080);
console.log('Server running on 8124');
```

Тут мы сначала запрашиваем еще один важный Node-модуль, отвечающий за работу с файловой системой. Затем при обработке запроса читаем файл **page.html** в нужной кодировке и записываем его в ответ сервера. Обратите внимание, что две последние операции мы выполняем опять в теле анонимной функции обратного вызова – чтение файла методом **readFile()** также проходит в асинхронном режиме.

Теперь при открытии браузером адреса http://127.0.0.1:8080 мы можем любоваться нашей веб-страницей (рис. 6), но боюсь, данный способ отдавать веб-контент – это не совсем то, чего вы ожидали.

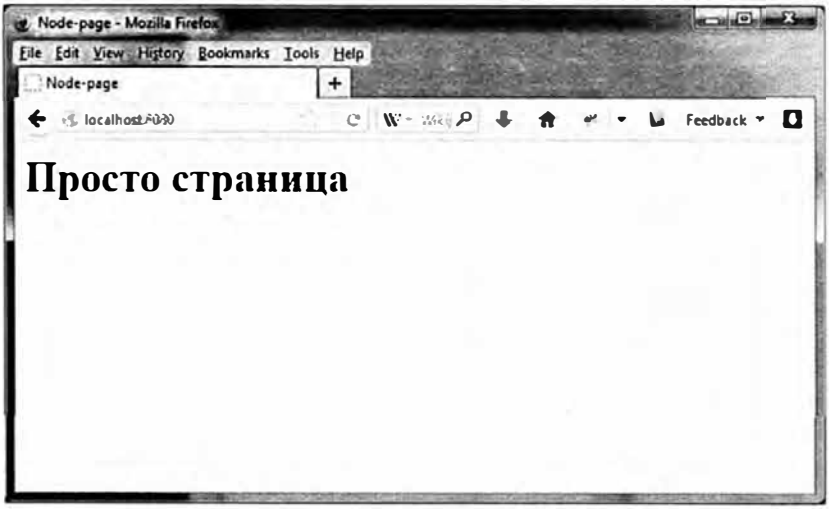

Рис. 6 ❖ Страница. Просто страница

Вообще, один из первых вопросов, которые обычно возникают у веб-разработчика, приступающего к изучению Node.js, – это: можно ли на этой среде построить обычный «плоский» сайт из статических страниц и как, черт возьми, это сделать? Оставим в стороне вопрос, предназначен ли Node.js для подобных задач, а просто попытаемся реализовать эту конструкцию.

Сайт на Node.js

Представим сайт, структура которого изображена на рис. 7. Ничего сложного, правда? При использовании в качестве веб-сервера Apache или Nginx нам бы потребовалось только разместить файлы в нужной папке и настроить права доступа. Здесь же все не так просто – нам надо самостоятельно написать механизм отдачи различного веб-контента, разбирая url и заботясь об отображении различных MIME-типов. Но нас этой задачей не испугаешь, к тому же в арсенале Node.js средств для её решения более чем достаточно.

Итак, приступим.

Сначала нам потребуется подключить еще один модуль:

```
var http = require("http");
var fs = require('fs');
var url = require("url");
```

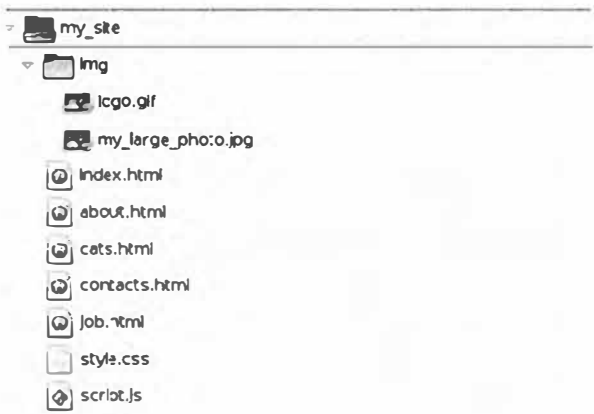

Рис. 7 ❖ Структура сайта

Он реализует методы, работающие с различными составляющими URL. Посмотрим на его работу:

```
http.createServer(function onRequest(request, response) {
  var pathname = url.parse(request.url).pathname;
  console.log("Получен запрос " + pathname);
  ..............
```

Теперь при запросе браузером адреса вида http://localhost:8080/about.html мы можем наблюдать в консоли информацию о запрашиваемом пути. Файл **index.html** содержит следующую разметку:

```
<html>
<head>
    <title>Node-page</title>
    <link rel=stylesheet href=styles.css type=text/css>
      <script src=script.js type=text/javascript></script>
</head>
<body>
    <h1><img src="img/logo.gif" />Сайт на Node.js</h1>
      <div class = "main_menu">
      <p><a href="about.html">О нас</a></p>
      <p><a href="cats.html">Кошечки</a></p>
      <p><a href="about.html">Контакты</a></p>
      <p><a href="about.html">Вакансии</a></p>
      </div>
      <img src= "my_large_photo.jpg" />
</body>
</html>
```

Попробуем его отобразить уже опробованным нами способом:

```
var pathname = url.parse(request.url).pathname;
console.log("Получен запрос " + pathname);
fs.readFile('index.html', 'utf8', function(error, data) {
  if(error){
    console.log('Could not find or open file for reading\n');
  } else {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end(data);
  }
})
```

Теперь при запуске браузера с адресом http://localhost:8080 мы будем наблюдать в консоли такую картину:

```
$ node start1.js
Server running on 8124
Получен запрос /
Получен запрос /my_large_photo.jpg
Получен запрос /styles.css
Получен запрос /script.js
Получен запрос /img/logo.gif
Получен запрос /img/my_large_photo.jpg
Получен запрос /favicon.ico
```

Тут представлено все разнообразие контента, которое нам надо в нужном формате отдать браузеру. Если бы этот контент состоял только из html-страниц, можно было бы ограничиться следующим кодом:

```
http.createServer(function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).path;
    if(pathname =='/')
       pathname = '/index.html';
     // чтобы убрать начальный слэш
    pathname = pathname.substring(1, pathname.length);

    fs.readFile(pathname, 'utf8', function(err, data) {
         if (err){
            console.log('Could not find or open file '+pathname +' for reading\n');
         } else {
               response.writeHead(200, {'Content-Type': 'text/html'});
            response.end(data);
            }
   })
}).listen(8080);
```

К сожалению (или к счастью, я за разнообразие), тут у нас присутствуют не только JavaScript- и CSS-файлы, которые современный браузер, поморщившись, примет и как 'text/html', но и изображения,

для которых правильный Content-type обязателен. Поэтому дополним наш код – сначала добавим еще один, небесполезный модуль:

```
var path = require('path');
```

Как можно понять из названия, он отвечает за параличные операции с путями в файловой системе. Далее создадим объект с mime-типами:

```
var mimeTypes = {
'.js' : 'text/javascript',
'.html': 'text/html',
'.css' : 'text/css',
'.jpg' :'image/jpeg',
'.gif' : 'image/gif'
};
```

Ну а теперь немного изменим код отдачи контента:

```
fs.readFile(pathname, 'utf8', function(err, data) {
         if (err){
           console.log('Could not find or open file '+pathname +' for reading\n');
         }  else {
              response.writeHead(200, {'Content-Type': mimeTypes[path.
extname(pathname)]});
            response.end(data);
            }
   })
```

Уже лучше, но картинок мы все равно не увидим. Причина проста – это бинарные данные, и читаются они другим способом. Тут я не буду подробно останавливаться на деталях (мы все подробно изучим потом), а просто покажу модифицированный код:

```
http.createServer(function onRequest(request, response) {
  var pathname = url.parse(request.url).path;
  if(pathname =='/'){
     pathname = '/index.html';
  }
  var extname = path.extname(pathname);
  console.log(extname);
  var mimeType = mimeTypes[path.extname(pathname)];
  pathname = pathname.substring(1, pathname.length);
  if((extname == ".gif") || (extname==".jpg")){
    var img = fs.readFileSync('./'+pathname);
    response.writeHead(200, {'Content-Type': mimeType });
    response.end(img, 'binary');
  } else {
    fs.readFile(pathname, 'utf8', function(err, data) {
    if (err){
```

```
      console.log('Could not find or open file '+pathname +' for reading\n');
    } else {
      console.log(pathname+" "+mimeType);
      response.writeHead(200, {'Content-Type': mimeType});
        response.end(data);
    }
  })
 }
}).listen(8080);
```

На этом все. Поставленную задачу мы выполнили, и теперь можно с чистой совестью приниматься за изучение Node.js по-серьёзному!

Node Core

Давайте рассмотрим, что собой представляет Node как «Software system». Из примеров, представленных в предыдущем разделе, можно сделать вывод о модульном характере её архитектуры, и он будет совершенно верен. Даже в простом движке плоского сайта мы использовали почти десяток модулей, каждый из которых отвечает за что-то свое. Это нормальный принцип построения Node-приложений. Но, разумеется, этим и многим другим модулям просто не с чем было бы работать, если бы не основа системы – ядро Node.js, содержащее объекты и методы, доступные всем модулям, в глобальном пространстве имен. Именно с этого мы и начнем изучение Node, попутно освоив базовые понятия и элементы системы.

Как все работает? Event loop в Node.js

В основе Node.js лежит библиотека libev, реализующая цикл событий (event loop). Libev – это написанная на C библиотека *событийно-ориентированной обработки данных, предназначенная для упрощения асинхронного неблокирующего ввода/вывода.*

При каждой итерации цикла происходят следующие события (причем именно в таком порядке):

- выполняются функции, установленные на предыдущей итерации цикла с помощью особого метода – **process.nextTick()**, обрабатывающего события libev, в том числе таймеров;
- выполняется опрос libeio (библиотеки для создания пула потоков – thread pool) для завершения операций ввода/вывода и выполнения установленных для них кэллбеков.

Если ни одно из вышеперечисленных действий не потребовалось (то есть все очереди, таймеры и т. д. оказались пусты), Node.js завершает работу.

Глобальные объекты (Globals)

С «Globals», то есть объектами-методами, доступными из любого метода, любого модуля, мы уже успели столкнуться. Это, например, метод requer или объект console. Давайте теперь разберем глобальные объекты и методы Node.js подробнее.

Global

Самый главный в иерархии глобальных объектов так и называется – **Global**.

Если вы имеете опыт программирования на JavaScript, то должны знать одну особенность его реализации в браузере. При инициализации переменной на верхнем уровне вложенности она автоматически становится глобальной в смысле области видимости. В Node.js переменная, объявленная в любом месте модуля, так будет определена только в этом модуле (и это замечательно!); чтобы она стала глобальной, необходимо объявить её как свойство объекта Global:

```
> global.foo = 2
2
> console.log(global)
{ ArrayBuffer: [Function: ArrayBuffer],
  Int8Array: { [Function: Int8Array] BYTES_PER_ELEMENT: 1 },
  Uint8Array: { [Function: Uint8Array] BYTES_PER_ELEMENT: 1 },
.....................................................
    registerExtension: [Function],
    cache: {} },
  _: 2,
  foo: 2 }
```

(Надо сказать, что команда consolo.log(global) всегда позволяет получить массу протезной и интересной информации.)

Фактический Global – это аналог объекта window в DOM-модели браузера. Он доступен отовсюду и не требует специального вызова, когда вызываются его методы.

Объект Console

Здесь все понятно – этот объект используется для стандартного вывода и вывода ошибок (**stdout** и **stderr**). Основной его метод **console.log()** просто выводит заданную строку (или объект, приведенный к строке) на стандартный вывод. **console.log()** может принимать два аргумента, аналогично функции языка Си **printf( )**:

```
7
> console.log('Price: %d', bar);
Price: 7
```

Для **stderr** (стандартного потока вывода ошибок) существует метод **console.error()**, в остальном полностью аналогичный **console.log()**.

Еще два метода – **console**, **console.time()** и **console.timeEnd()** – позволяют отслеживать время исполнения программы. Первый из

них устанавливает (запоминает) текущее время, а второй завершает отсчёт времени и выводит результат:

```
console.time('items');
for (var i = 0; i < 1000000000; i++) {
  // что-нибудь делаем
}
console.timeEnd('1000000000-items');
1000000000-items: 1492ms
```

Для отладки также полезен метод **console.trace()**, выводящий в консоль стек вызовов для текущей инструкции:

```
for (var i = 0; i < 100; i++) {
  console.trace()
}

Trace
    at Object.<anonymous> (C:\Users\Geol\time.js:5:10)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
Trace
    at Object.<anonymous> (C:\Users\Geol\time.js:5:10)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
.....
```

Require и псевдоглобальные объекты

Метод require, служащий для подключения модулей, на самом деле не является глобальным – он локален для каждого модуля. Это же относится к методу **require.resolve()**, возвращающему имя файла, содержащего модуль, и свойствам **require.cache** и **require.paths**, первое из которых отвечает за кэширование модулей, а второе – за пути их загрузки.Еще к global относят свойства, на самом деле также являющиеся локальными для каждого модуля:

- **module** – ссылка на текущий модуль;
- **exports** – объект, является общим для всех экземпляров текущего модуля, доступным при использовании **require(). exports.**
- **__filename** – имя исполняемого скрипта (абсолютный путь);

- **__dirname** – имя директории исполняемого скрипта.

Кроме того, в секцию Globals входят еще три важных элемента, о которых поговорим ниже:

- Process – объект процесса. Большая часть данных процесса находится именно здесь;
- Class: Buffer – объект используется для операций с бинарными данными; таймеры – группа методов для контроля времени исполнения.

Процессы

Свойства и методы объекта Process

Каждое Node.js-приложение – это экземпляр объекта **Process** и наследует его свойства. Это свойства и методы, несущие информацию о приложении и контексте его исполнения. Пример работы некоторых из них (назначение свойств процесса очевидно из названий):

```
console.log(process.execPath);
console.log(process.version);
console.log(process.platform);
console.log(process.arch);
console.log(process.title);
console.log(process.pid);
```

Результат может быть, например, таким (операционная система Windos 7):

```
C:\Users\Geol>node process.js
C:\Program Files\nodejs\node.exe
v0.10.7
win32
x64
Command Prompt - node  process.js
5896
```

Свойство **process.moduleLoadList** показывает информацию о загруженных модулях, а **process.argv** содержит массив аргументов командной строки:

```
process.argv.forEach(function(val, index, array) {
  console.log(index + ' - ' + val);
});
```

Запустим этот сценарий с несколькими аргументами:

```
C:\Users\Geol>node process.js foo bar 1

0 - node
```

```
1 - C:\Users\Geol\process.js
2 - foo
3 - bar
4 - 1
```

Тут можно понять, что первым аргументом считается имя исполняемого приложения, вторым – сам исполняемый сценарий (точнее, файл, его содержащий).

Метод **process.env** возвращает объект, хранящий пользовательское окружение процесса, и практически незаменим при отладке приложений. Ниже показана часть вывода команды console.log(process.env):

```
{ ALLUSERSPROFILE: 'C:\\ProgramData',
  APPDATA: 'C:\\Users\\Geol\\AppData\\Roaming',
  'asl.log': 'Destination=file',
  CommonProgramFiles: 'C:\\Program Files\\Common Files',
  COMPUTERNAME: 'GEOL-PC',
  ComSpec: 'C:\\Windows\\system32\\cmd.exe',
  FP_NO_HOST_CHECK: 'NO',
  HOMEDRIVE: 'C:',
  HOMEPATH: '\\Users\\Geol',
  ...
  PATHEXT: '.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC',
  PROCESSOR_ARCHITECTURE: 'AMD64',
  PROCESSOR_IDENTIFIER: 'Intel64 Family 6 Model 37 Stepping 5, GenuineIntel',
  PROCESSOR_LEVEL: '6',
  PROCESSOR_REVISION: '2505',
...
  PUBLIC: 'C:\\Users\\Public',
  SESSIONNAME: 'Console',
  SystemDrive: 'C:',
  SystemRoot: 'C:\\Windows',
  USERDOMAIN: 'Geol-PC',
  USERNAME: 'Geol',
  USERPROFILE: 'C:\\Users\\Geol',
  VBOX_INSTALL_PATH: 'C:\\Program Files\\Oracle\\VirtualBox\\',
  windir: 'C:\\Windows' }
```

Команда **process.exit()** завершает процесс с указанным в качестве аргумента кодом (по умолчанию со стандартным успешным кодом 0). Чтобы выйти с ошибкой, следует выполнить **process.exit(1)**. Код выхода можно отследить:

```
process.on('exit', function () {
setTimeout(function(code) {
  // Этот код никогда не запустится!
   console.log('This will not run');
```

```
  },0);
  console.log('Exit with code:' + code);
});
```

Метод **process.kill()**, аргументами которого служат идентификатор процесса и команда, делает то же, что и одноименная команда операционной системы, то есть посылает сигнал процессу (как в случае posix kill, это не обязательно сигнал завершения).

Еще один информативный метод, **process.memoryUsage()**, возвращает объект, описывающий потребление памяти процессом Node (в байтах):

```
console.log(process.memoryUsage());
process.nextTick(function () {
    console.log('Test');
});

console.log(process.memoryUsage());
```

Вывод:

```
C:\Users\Geol>node process.js
{ rss: 11096064, heapTotal: 4083456, heapUsed: 2149792 }
{ rss: 11526144, heapTotal: 5115392, heapUsed: 2493656 }
Test
```

Тут следует пояснить, что heapTotal и heapUsed характеризуют потребление памяти JavaScript движком V8.

Метод process.nextTick()

Как вы могли заметить, в этом примере мы использовали еще один метод объекта process – **process.nextTick()**, который напрямую отсылает нас к циклу событий (event loop), с описания которого мы начали эту главу. **NextTick()** назначает функцию, служащую ему аргументом, к исполнению при следующей итерации цикла событий. Это такой своеобразный event handler, специфичный для node.js, который, правда, в этом примере служит всего лишь элегантной заменой **setTimeout()**. Работу **NextTick()** руководство поясняет на более простом примере:

```
process.nextTick(function() {
  console.log('nextTick callback');
});
```

Напомним, что на каждом витке event loop в первую очередь идёт выполнение функций, установленных на предыдущем витке цикла

с помощью **process.nextTick()**. Далее идёт обработка событий libev, в частности событий таймеров. В последнюю очередь идёт опрос libeio для завершения операций ввода/вывода и выполнения установленных для них функций обратного вызова. В случае если ни одной операции не назначено, нет работающих таймеров и очереди запросов в libev и libeio пусты, node завершает работу.

Процессы ввода/вывода

Стандартные процессы ввода/вывода операционной системы – **stdin**, **stdout** и **stderror** – также имеют свое воплощение в объекте process. Ниже представлен небольшой сервер эхо-печати, просто возвращающий в консоль вводимый текст:

```
process.stdin.setEncoding('utf8');
process.stdin.resume();

process.stdin.on('data', function(chunk) {
  if (chunk !== "end\n") {
    process.stdout.write('data: ' + chunk);
  } else {
    process.kill();
  }
});
```

Тут мы сначала устанавливаем кодировку вывода. Затем делаем очень важный метод **process.stdin.resume()**, возобновляющий работу потока **process.stdin** (он по умолчанию приостановлен). Далее к запущенному процессу привязывается обработчик на событие поступления данных. В функции обратного вызова **process.stdout** просто пишет поступившие данные в консоль (до того, пока не будет встречено слово «end»» и перевод строки). Запустим этот код и напечатаем что-нибудь в консоли:

```
sukhov@geol-System-Product-Name:~/node$ node process
test
data: test
Hello!
data: Hello!
Bay!
data: Bay!

sukhov@geol-System-Product-Name:~/node$
```

Поток **process.stderr** представляет стандартный поток ошибок **stderr**. Следует принимать во внимание важное отличие его от предыдущих потоков, операция записи в него всегда является блокирующей.

Signal Events

У объекта **Process**, как и у всех порядочных javaScript-объектов, есть свои события, с которыми можно связать необходимые обработчики, и некоторыми из них мы уже успели воспользоваться. Нас сейчас интересуют события, специфичные для этого объекта, и прежде всего это так называемые сигнальные события (Signal Events), генерирующиеся при получении процессом сигнала (собственно POSIX-функция sigaction). Сигналы могут быть стандартные, POSIX-ов – SIGINT, SIGUSR1, SIGTSTP и др. Ниже приведен пример кода, при запуске которого функция обратного вызова будет ждать сигнала, соответствующего нажатию **Ctrl+C** на клавиатуре:

```
process.stdin.resume();

process.on('SIGTSTP', function() {
   console.log('Interrupted');
   process.exit(1);
});
```

Еще один пример:

```
process.on('SIGHUP', function() {
  console.log('Got a SIGHUP');
});

setInterval(function() {
    console.log('Running');
}, 10000);

console.log('PID:', process.pid);
```

После запуска этого кода в консоли мы увидим следующую картину:

```
PID: 5772
Running
Running
Running
Running
Running
Running
Running
Running
```

Теперь в другой консоли, зная PID запущенного процесса, можно послать ему требуемый сигнал:

```
kill -s SIGHUP 5772
```

Результат:

```
Got a SIGHUP
```

Child Process – параллелизм в Node.js

Модуль, название которого вынесено в заголовок, требует процедуры require, но это стоит того. При его применении работа с процессами в Node.js становится полноценной. Вот некоторые возможности child_process:

- позволяет запускать команды shell;
- дает возможность запускать дочерние процессы, исполняемые параллельно;
- позволяет процессам обмениваться сообщениями.

Простой, но действенный метод этого модуля, **child_process.exec()**, позволяет запустить shell-команду на выполнение и сохранить результат в буфер. Вот пример его работы:

```
var exec = require('child_process').exec;

var child = exec('cat *.js not_ecxist_file | wc -l', {cwd: '/'},
    function (error, stdout, stderr) {
        console.log('stdout: ' + stdout);
        console.log('stderr: ' + stderr);
        if (error !== null) {
          console.log('exec error: ' + error);
        }
});
```

Тут мы с помощью метода **child_process.exec()** запускаем shell-команду cat, выводящую содержимое нескольких файлов и через пайп передающую их команде wc, считывающей количество строк. Вторым аргументом передаётся массив параметров (в данном случае мы уточняем рабочую директорию). Функция обратного вызова, служащая третьим аргументом метода, принимает три параметра, кроме объекта ошибки, это еще и потоки **stdout**, **stderr**.

Результат будет следующим:

```
sukhov1@sukhov-System-Product-Name:~/node$  node pr2
stdout: 263

stderr: cat: not_ecxist_file: No such file or directory
```

Ценность команды состоит в возможности дальнейших действий с полученными результатами.

Следующий метод еще интереснее. **Child_process.swan()** запускает команду в новом процессе, дескриптор которого становится доступным:

```
var spawn = require('child_process').spawn;
var ls    = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

ls.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

ls.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

Результат:

```
$ node  spawn .js
stdout: total 128K

drwxr-xr-x   2 root root  44K авг.  22 17:52 bin

drwxr-xr-x   2 root root 4,0K окт.  16  2013 games

drwxr-xr-x  36 root root 4,0K авг.  20 18:35 include

drwxr-xr-x 171 root root  36K авг.  21 11:50 lib

drwxr-xr-x  13 root root 4,0K авг.  20 17:26 local

drwxr-xr-x   2 root root  12K авг.  21 11:50 sbin

drwxr-xr-x 305 root root  12K авг.  22 17:52 share

drwxr-xr-x  38 root root 4,0K авг.  20 19:33 src


child process exited with code 0
```

Тут все просто – любой вывод исполняемой команды передаётся в поток **stdout** дочернего процесса (или в **stderr**, если случилась ошибка), и мы связали обработчики именно с этими объектами. Мы видим, что поток исполнялся параллельно с основным процессом и завершился с кодом 0, что свидетельствует об успехе. Чтобы проде-

монстрировать использование еще и потока **stdout**, трудно не удержаться и не привести красивый пример из руководства Node.js:

```
var spawn = require('child_process').spawn;
var ps    = spawn('ps', ['ax']),
var grep  = spawn('grep', ['ssh']);

ps.stdout.on('data', function (data) {
  grep.stdin.write(data);
});

ps.stderr.on('data', function (data) {
  console.log('ps stderr: ' + data);
});

ps.on('close', function (code) {
  if (code !== 0) {
    console.log('ps process exited with code ' + code);
  }
  grep.stdin.end();
});

grep.stdout.on('data', function (data) {
  console.log('' + data);
});

grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

grep.on('close', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

В этом коде запускаются два дочерних процесса, исполняющих две unix-команды: ps – выводящую отчёт о работающих процессах и grep – находящую соответствие заданному регулярному выражению подаваемой на ввод строки. Суть программы заключается в том, чтобы данные, полученные в поток stdout ps, направить в stdin grep. Как видите, все получается:

```
$ node  spawn
  737 pts/1    S+     0:00 grep ssh

 1311 ?        Ss     0:00 ssh-agent

 5172 pts/10   S+     0:00 ssh sukhov@91.142.84.103 -p 4251 -i sukhov.rsa
```

```
5873 pts/14   S+     0:00 sudo -i ssh sukhov@192.168.1.201

5877 pts/14   S+     0:00 ssh sukhov@192.168.1.201
```

(Аналогичного результата можно достичь, набрав в консоли 'ps ax | grep ssh'.)

Следующий метод **child_process.fork()** запускает дочерним процессом процесс, порожденный самой Node.js. Продемонстрировать его работу можно следующим простым кодом (файл **main.js**):

```
var cp = require('child_process');
var child1 = cp.fork(__dirname + '/sub1.js');
var child2 = cp.fork(__dirname + '/sub2.js');
while(cp){
  console.log("running main");
}
```

Тут мы запускаем два дочерних процесса, исполняющих код, хранящийся в файлах **sub1.js** и **sub2.js**, и создаём бесконечный цикл вывода сообщений. Содержимое исходного кода дочерних процессов тоже сильно не отличается:

- **sub1.js**:

```
while(1){
  console.log("running: process1");
}
```

- **sub2.js:**

```
while(1){
  console.log("running: process2");
}
```

Теперь запускаем основной процесс:

```
$ node main.js
running main

running: process1

running main

running: process2

running: process1

running main

running main

running: process1
running: process2
```

```
running main

.....................................................
```

Все параллельно и все работает!

Еще один метод – **child.send()** – самый интересный. В соответствии со своим названием он отсылает сообщения. Отсылает сообщения от процесса к процессу (**main.js**):

```
var cp = require('child_process');

var child = cp.fork(__dirname + '/sub.js');
child.on('message', function(data) {
  console.log('Main got message:', data);
});

child1.send({ hello: 'child' });
```

sub.js:

```
process.on('message', function(m) {
  console.log('Child got message:', m);
});
process.send({ foo: 'bar' });
```

Результат:

```
$ node process

Main got message: { foo: 'bar' }

Child got message: { hello: 'child' }
```

Разумеется, в виде сообщений можно посылать команды управления.

Понятие буфера

В любой системе, претендующей на роль серверной платформы в вебсреде, необходимы средства для полноценной работы с потоками двоичных данных – одним plain-текстом сыт не будешь. В классическом JavaScript подобные средства отсутствовали (если не считать недавно появившихся типов File API, ArrayBuffer, относящихся не к самому языку, а к объектной модели браузера). В Node.js для решения подобных задач существует объект Buffer. Бинарные данные хранятся в экземплярах этого класса, с ним ассоциирована область памяти, выделенная вне стандартной кучи V8.

Еще одна проблема, которую решает данный объект, является частным случаем первой. Я имею в виду работу со строками unicode.

При преобразовании между буферами и строками JavaScript требуется явно указывать метод кодирования символов. Node поддерживает следующие кодировки для строк:

- **'ascii'** – только для 7-битных ASCII-строк. Этот метод кодирования очень быстрый, он сбрасывает старший бит символа;
- **'utf8'** – Unicode-символы UTF-8;
- **'binary'** – хранит двоичные данные в строке, используя младшие 8 бит каждого символа. Это кодирование не будет поддерживаться в будущих версиях;
- **'base64'** – строка, закодированная в системе Base64;
- **'hex'** – кодирует каждый байт как два шестнадцатеричных символа.

Работа с буфером происходит следующим образом:

```
buf = new Buffer(256, 'utf8');
text = '\u00bd + \u00bc = \u0D75';
len = buf.write(text , 0, text.length);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
```

Этот чуть измененный пример из руководства выводит в консоль следующую строку (при наличии соответствующей кодировки):

```
$ node ts1.js
13 bytes: ½ + ¼ = ¾
```

Конструктор объекта получает в качестве аргумента желаемый размер буфера в байтах (его потом невозможно изменить). Тут же вторым аргументом указываем кодировку. Можно этого не делать: требуемый utf8 – значение по умолчанию.

В данном примере мы использовали метод **buffer.write()**, помещающий данные в буфер. Он возвращает количество байт. Аргументами его служат (кроме самой строки) начальная и конечная позиции записи данных в пространство буфера. Оба они не обязательны. Важная особенность, призванная немного сохранить психику разработчиков, заключается в том, что, несмотря на то что запись данных прекращается при превышении размера буфера, символы юникода не будут записаны «частично», по первому байту, или целиком, или никак. Думаю, что Perl-, php- да и Си-разработчики могут по достоинству оценить эту мелочь.

Метод **buffer.toString()**, как несложно догадаться, превращает содержимое буфера в виде строки. При необходимости (см. первый

аргумент в примере) может и перекодировать. Для полного счастья программистов предусмотрен еще метод **buffer.toJSON()**:

```
buf = new Buffer('test');
console.log(buf.toJSON());
```

Вывод:

```
$ node buffer
[ 116, 101, 115, 116 ]
```

Метод **buffer.length()**, возвращающий размер данных в буфере, имеет одну особенность – это общий объем зарезервированного пространства под данные, он может не совпадать с объёмом самих данных.

Еще один способ задания буфера – непосредственная передача конструктору массива байтов (то есть восьмибитных данных):

```
buf = new Buffer([01,02,03,04,05]);

console.log(buf);
console.log(buf[4]);
```

Вывод:

```
$ node buffer
<Buffer 01 02 03 04 05>
5
```

В последней строчке этого примера мы обращаемся к содержимому буфера по индексу – собственно, это основной приём работы с данным объектом.

Что еще можно творить с буфером? Ну, например, копировать один буфер в другой, задав границы областей вставки и замещения:

```
buffer1 = new Buffer(24);
buffer2 = new Buffer(16);
for (var i = 0 ; i < 24 ; i +=4) {
  buffer1[i] = 78;
  buffer1[i+1] = 111;
  buffer1[i+2] = 100;
  buffer1[i+3] = 101;
}
for (var i = 0 ; i < 16 ; i++) {
  buffer2[i] = 42; // ASCII *
}

buffer1.copy(buffer2, 6, 16, 20);
console.log(buffer2.toString('ascii'));
```

Вывод:

```
$ node buffer
******Node******
```

Метод **buffer.slice()** возвращает новый буфер, представляющий собой срез старого. При этом надо понимать, что вновь сознанный объект указывает на ту же область памяти, что и предыдущий, соответственно, любые изменения нового буфера коснутся и буфера источника:

```
var buffer2 = buffer1.slice(0, 3);
console.log(buffer2.toString('ascii', 0, buffer2.length));
buffer2[0] = 33;
console.log(buffer1.toString('ascii', 0, buffer1.length));
```

Вывод:

```
$ node buffer
***
!*********
[TODO SlowBuffer]
```

Таймеры

Таймеры Node.js представлены несколькими жизненно необходимыми глобальными функциями, хорошо знакомыми по классическому JavaScript. Прежде всего это **setTimeout()**, позволяющая выполнить переданный ей в качестве аргумента код через заданное количество миллисекунд. Функция возвращает ID тайм-аута.

clearTimeout() обнуляет счетчики по заданному идентификатору. Работу этих двух функций можно продемонстрировать на простом рекурсивном примере:

```
var tid;
function toConsole(n) {
    console.log(n);
     tid = setTimeout(toConsole, 1000, n+1);
     if(n  >  5){
         clearTimeout(tid);
     }
}
toConsole(0);
```

Вывод:

```
$ node buffer
0
```

```
1
2
3
4
5
6
```

Тут следует обратить внимание на то, что все дополнительные аргументы **setTimeout()** превращаются в аргументы функции обратного вызова. То обстоятельство, что в консоль проникла цифра 6, объясняется тем, что таймер успевает отработать перед уничтожением. Впрочем, аналогичная задача решается без всякой рекурсии двумя другими таймер-функциями **setInterval()** и **clearInterval()**, устанавливающими и сбрасывающими (соответственно) так называемые интервальные таймеры, то есть таймеры, срабатывающие периодически, через заданный интервал:

```
var tid;
function toConsole() {
    console.log(n);
     n++;
     if(n > 5){
         clearInterval(tid);
     }
}
var n = 0;
tid = setInterval(toConsole, 1000);
```

Да, кстати, и это все! То есть после получения вышеизложенных сведений (соглашусь, местами нудноватых) мы можем творить с Node.js практически все! Не верите? Напрасно. То, что сейчас написано и реализовано для node, – это модули, созданные на данной основе. Но!

Работа с Node.js практически невозможна без её продвинутой событийной модели, модуля events и класса **EventEmitter**.

События

Обработка событий – основа работы с Node.js. События генерируют практически все объекты. В явном и неявном виде мы уже использовали их (например, событие **'exit'** для объекта process), теперь пришло время разобраться с этим механизмом более подробно.

Слушаем!

За события в Node.js отвечает специальный модуль – events.

Назначать объекту обработчик события следует методом **addListener(event, listener)**, аналогичным имеющемуся в обычном «браузерном» JavaScript. Аргументами для него служат имя события (строка, обычно в camelCase-стиле: connect, messages, messageBegin) и функция обратного вызова – обработчик события. Для особо ленивых разработчиков, привыкших к удобствам jQuery, для этого метода существует синоним – просто **on()**:

```
server.on('connection', function () {
  console.log('connected!');
});
```

По-моему, это прекрасно, давайте и далее останемся ленивыми. У метода **on()** есть чрезвычайно полезная модификация – **once()**, назначающая однократный обработчик события. То есть код

```
server.once('connection', function () {
  console.log('first connection!');
});
```

сработает только при первом соединении с сервером.

Теоретически с одним объектом можно связать сколько угодно обработчиков, но по умолчанию их количество ограничено 10. Это сделано для предотвращения утечек памяти. Ограничение преодолевается методом **setMaxListeners(n)**, где **n** – требуемое максимально допустимое количество обработчиков.

Посмотреть все обработчики объекта, связанные с конкретным событием, можно методом **listeners**, возвращающим массив обработчиков:

```
var http = require('http');
var server = http.createServer(function (request, response) {
    }).listen(8080);

function serverClose () {
    server.close();
}
server.on('connection', function () {
```

```
    console.log('Connected!');
});

server.on('connection', serverClose );
console.log(server.listeners('connection'));
```

Вывод:

```
$ node ev.js
[ [Function: connectionListener],
  [Function],
  [Function: serverClose] ]
```

Тут может возникнуть вопрос, откуда взялся первый обработчик, ведь мы ничего подобного не назначали? Всё просто, **connectionListener** автоматически создается методом **http.createServer()**, это «родной» обработчик объекта.

Удалить обработчик можно можно методом **removeListener(event, listener)**. Как видно из сигнатуры метода, желательно, чтобы функция-обработчик была именована или присвоена именованной переменной:

```
var callback = function() {
  console.log('Connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

Наконец, метод **emit(event, [args])** позволяет назначенным обработчикам срабатывать, как если бы связанное событие случилось. Причем событие, переданное **emit()**, не обязательно должно вообще существовать:

```
server.on('someevent', function (arg) {
  console.log('event '+arg);
});
server.emit('someevent', '!!!');
console.log(util.inspect(server.listeners('someevent')));
```

Вывод:

```
event !!!
[ [Function] ]
```

Кроме имени обработчика, метод в качестве необязательных параметров принимает список его аргументов.

Как видите, работать в Node.js с событиями просто и удобно. Но на самом деле все вышеизложенное – лишь внешние, инкапсулированные методы объекта, стоящего за всеми обработками событий Node.js.

Объект EventEmitter

EventEmitter – это основной объект, реализующий работу обработчиков событий в Node.js. Любой объект, являющийся источником событий, наследует от класса **EventEmitter**, и, повторюсь, все методы, о которых мы говорили, принадлежат этому классу.

Оперировать событиями посредством **EventEmiter** можно и напрямую, явным образом, создав объект этого класса:

```
var EventEmitter = require('events').EventEmitter;
var emiter = new EventEmitter();

emiter.on('myEvent', function(ray) {
    console.log(ray);
});
setInterval(function() {
    emiter.emit('myEvent', 'YES!');
}, 1000);
```

Для того чтобы добавить методы **EvetntEmiter** к произвольному (например, созданному нами) объекту, достаточно унаследовать **EvetntEmiter** с помощью метода **inherits** из модуля **utils** (речь о котором еще впереди). Давайте попробуем это сделать. Сначала создадим объект:

```
var VideoPlayer = function(movie) {
    var self = this;
    setTimeout(function() {
        self.emit('start', movie);
    }, 0);

    setTimeout(function() {
        self.emit('finish', movie);
    }, 5000);

    this.on('newListener', function(listener) {
        console.log('Event Listener: ' + listener);
    });
}
```

Если прямо сейчас мы попробуем использовать его, связывая с ним обработчики событий:

```
myPlayer.on('start', function() {
    console.log(' movie started');
});
```

то результат будет немного предсказуем:

```
/home/geol/node/newNodel.js:16
    this.on('newListener', function(listener) {
         ^
TypeError: Object [object Object] has no method 'on'
    at new VideoPlayer (/home/geol/node/newNodel.js:16:10)
    at Object.<anonymous> (C:\Users\Geol\node\newNodel.js:25:16)
```

```
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

Все верно, ведь **методу on()** пока просто неоткуда взяться. Исправим это положение вещей:

```
var util = require("util");
var EventEmitter = require('events').EventEmitter;
var VideoPlayer = function(movie) {
    var self = this;
.....
};
util.inherits(VideoPlayer, EventEmitter);
```

И теперь можем смело обращаться к объекту:

```
};
util.inherits(VideoPlayer, EventEmitter);
var movie = {
    name: 'My cat'
};
var myPlayer = new VideoPlayer(movie);

myPlayer.on('start', function(movie) {
    console.log('"%s" started', movie.name);('? d?');
});

myPlayer.on('finish', function(movie) {
    console.log('"%s" finished', movie.name);
});
```

Результат:

```
C:\Users\Geol\node>node newNodel.js
Event Listener: start
Event Listener: finish
"My cat" started
"My cat" finished
```

Возможно, метод **inherits()** немного сбивает с толку, особенно если JavaScript – не ваш основной язык разработки. Тут важно понимать, что это не классическое ООП-наследование, не имплементация, не агрегация. Это прототипирование.

Вот теперь действительно все. То есть вся мощь Node.js теперь в наших руках. И теперь мы можем реализовывать на этой платформе что угодно. Правда, у меня сюрприз. Наверное, приятный. Это все в основном уже реализовано.

Модули

Да, все (весьма впечатляюще) возможности платформы Node.js реализованы (и продолжают реализовываться) в модулях. Мы уже неоднократно применяли этот механизм расширений – модули **util**, **http**, **fs**, **event** и др. Кроме набора модулей, входящего в стандартную поставку Node.js, существует великое множество расширений, реализующих самую разную функциональность. Это могут быть модули для работы с разными базами данных, с протоколом WebSockets, с Apache Hadoop или хэш-таблицами memcached. Замечательный MVC-фреймворк Express, фреймворк Connect, шаблонный движок Jade, все остальные инструменты Node.js – все это тоже модули.

Если те модули, которые мы уже успели попробовать, входят в ядро Node.js и не требуют установки (достаточно вызова метода **requere()** с названием модуля), то про огромное множество модулей, созданных сторонними (и не очень) разработчиками, такого сказать нельзя. Их необходимо разыскивать и устанавливать. «Водятся» модули на просторах репозитория GitHub, а также в специально отведенных «заказниках».

Прежде всего это wiki для Node-модулей на GitHub:

```
https://github.com/joyent/node/wiki/modules
```

и Nipster!:

```
http://eirikb.github.io/nipster.
```

Кроме того, существует и основное, специализированное хранилище модулей, но о нем чуть позже.

Установить модуль можно, просто скачав его исходные коды и связав с сосуществующим приложением. Давайте попробуем совершить это на примере простенького, но симпатичного модуля Colors, реализующего абсолютно необходимый функционал – вывод текста на консоль с форматированием и в различном цвете.

Сначала скачиваем исходный текст модуля с его страницы https://github.com/Marak/colors.js. Соберем все необходимые файлы в папку /colors. Теперь напишем небольшой сценарий, демонстрирующий возможности модуля (файл **color.js**):

```
var colors = require('./colors');

console.log('rainbows rasing!'.rainbow);
console.log('background color!'.grey.blueBG)
console.log(colors.bold(colors.red('Chains are also cool...')));
```

```
colors.addSequencer("rastafari", function(letter, i, exploded) {
  if(letter === " ") return letter;
  switch(i%3) {
    case 0: return letter.red;
    case 1: return letter.yellow;
    case 2: return letter.green;
  }
});
console.log("Bob Marley YEAH!".rastafari );
```

Я не собираюсь разбирать приемы работы с **Colors** – они описаны в документации. Тут нам интересна только первая строчка. В ней мы подключаем свежескачанный модуль, причем здесь мы должны указать физический путь к нему (в данном случае относительный, но можно указать и полный, корня файловой системы).

Запустим :

```
$ node color.js
```

И сможем полюбоваться картинкой, приведенной на рис. 8. Правда, здорово? Вот и я говорю – ужас! Посему идем дальше.

Рис. 8 ❖ Гламурная консоль

Пакетный менеджер npm

Модуль, который мы установили, прямо скажем, не балует богатством функционала, впрочем, со своими задачами, при всей простоте, он вполне справляется. Справляется именно из-за простоты – модуль не использует никаких ресурсов и, самое главное, не отягощён связями с другими внешними модулями. На самом деле это достаточно уникальная ситуация, любое более или менее сложное расширение, как правило, тянет за собой пучок зависимостей (а чего вы хотели? Где модульная архитектура, там и «dependencies»!). Устанавливать все необходимое вручную – дело очень муторное и неблагодарное. Естественно, в подобного рода работе давно уже нет необходимости. Средство для поиска, установки и обновления модулей Node.js теперь входит в стандартную поставку платформы. Это npm – Node Package Manager, диспетчер пакетов Node.

Наличие npm в системе нетрудно проверить, выполнив в консоли следующую команду:

```
$ npm
```

В случае если все в порядке, будет выведен список (не очень большой) команд диспетчера:

```
Usage: npm <command>

where <command> is one of:
    add-user, adduser, apihelp, author, bin, bugs, c, cache,
    completion, config, ddp, dedupe, deprecate, docs, edit,
    explore, faq, find, find-dupes, get, help, help-search,
    home, i, info, init, install, isntall, issues, la, link,
    list, ll, ln, login, ls, outdated, owner, pack, prefix,
    prune, publish, r, rb, rebuild, remove, restart, rm, root,
    run-script, s, se, search, set, show, shrinkwrap, star,
    stars, start, stop, submodule, tag, test, tst, un,
    uninstall, unlink, unpublish, unstar, up, update, version,
    view, whoami

npm <cmd> -h     quick help on <cmd>
npm -l           display full usage info
npm faq          commonly asked questions
npm help <term>  search for help on <term>
npm help npm     involved overview

Specify configs in the ini-formatted file:
    C:\Users\Geol\.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config
```

Собственно, про работу с npm после этого можно ничего больше не писать – тут все понятно. Просто поясним на примере.

Допустим, нам понадобилось (а нам в дальнейшем это непременно понадобится) работать из Node с документоориентированным NoSQL-хранилищем данных MongoDB. В ядре Node.js такого модуля нет, поэтому воспользуемся npm.

Прежде всего нелишним будет посмотреть, какие модули у нас уже установлены. Сделать это можно командой npm ls:

```
$ npm ls

/home/geol/
├── colors@0.6.2
├── websocket@1.0.8
└── ws@0.4.27
    ├── commander@0.6.1
    ├── options@0.0.5
    └── tinycolor@0.0.1
```

Как видите, ничего похожего на требуемое у нас не обнаружено.

Теперь ищем нужные модули. Проще всего это делать на веб-странице npm, https://npmjs.org/, где все дополнения структурированы и каталогизированы, с возможностью поиска в едином реестре (рис. 9), но сейчас попробуем выполнить поиск из консоли:

```
$ npm search mongodb
```

Рис. 9 ❖ Сайт репозитория npm

Результат будет довольно впечатляющим:

```
npm http GET https://registry.npmjs.org/-/all/since?stale=update_a
1379573866851
npm http 200 https://registry.npmjs.org/-/all/since?stale=update_a
1379573866851
NAME                    DESCRIPTION
bass-mongodb            MongoDB Adapter for Bass.js
bitcoinjs-mongoose      Mongoose MongoDB ORM (temporary BitcoinJS fo
caminte                 ORM for every database: redis, mysql, neo4j
modm                    A MongoDB Object Document Mapper (ODM)
monastery               Light modeling for MongoDB
mongo-col               mongoDB collection wrapper
mongo-lite              Simple and minimal Driver for MongoDB
mongo-model             Model for MongoDB
mongo-relation          Model relationships plugin for Mongoose
mongo_model             Model for MongoDB
mongodb-collection-dump Dumps a mongodb collection to a stream.
mongodb-uri             A parser and formatter for MongoDB URIs.
mongol                  Light modeling for MongoDB
```

```
mongolite              Simple and minimal Driver for MongoDB
mongomodel             Model for MongoDB
mongoose               Elegant MongoDB object modeling for Node.jsST
```

Здесь есть из чего выбрать, и мы делаем выбор – модуль **mongoose** (почему, будет рассказано при описании работы Node.js с MongoDb).

Теперь произведем установку:

```
npm install mongoose

npm http GET https://registry.npmjs.org/mongoose
npm http 200 https://registry.npmjs.org/mongoose
npm http GET https://registry.npmjs.org/mongoose/-/mongoose-3.6.19.tgz
npm http 200 https://registry.npmjs.org/mongoose/-/mongoose-3.6.19.tgz
.....................................................................
npm http 304 https://registry.npmjs.org/bson/0.2.2
npm http 304 https://registry.npmjs.org/kerberos/0.0.3

> kerberos@0.0.3 install /home/Geol/node_modules/mongoose/node_modules/mongodb/
node_modules/kerberos
> (node-gyp rebuild 2> builderror.log) || (exit 0)
.....................................................................
/home/Geol/node_modules/mongoose/node_modules/mongodb/node_modules\bson
>node: "\Program Files\nodejs\node_modules\npm\bin\node-gyp-bin\\..\..\node_
modules\n
ode-gyp\bin\node-gyp.js" rebuild
npm WARN package.json policyfile@0.0.4 No repository field.
npm WARN package.json policyfile@0.0.4 'repositories' (plural) Not supported.
npm WARN package.json Please pick one as the 'repository' field
mongoose@3.6.19 ..\node_modules\mongoose
├── regexp-clone@0.0.1
├── hooks@0.2.1
├── ms@0.1.0
├── muri@0.3.1
├── sliced@0.0.5
├── mpath@0.1.1
├── mpromise@0.2.1 (sliced@0.0.4)
└── mongodb@1.3.19 (kerberos@0.0.3, bson@0.2.2)
```

Полного ввода я не привожу, но строк, приведенных выше, хватает для понимания сути процесса. Сначала npm запрашивает и скачивает последнею версию требуемого модуля (впрочем, версию можно указать точно), потом другие модули, необходимые для соблюдения зависимостей. Все это распаковывается и устанавливается локально. В конце npm вводит небольшой отчет – подсказывает дерево предустановленных модулей с номерами версий.

Теперь в нашей домашней папке должна появиться (если ее там до сих пор не было) папка **/node_modules**, в которой будут храниться

локально уставленные Node.js-расширения. Все! Модуль можно использовать.

Я не случайно несколько раз упомянул слово «локально», потому что существует и глобальная установка. Если бы мы инсталлировали mongoose с соответствующим ключом:

```
npm -g install mongoose
```

то код модуля был бы установлен в системную папку и стал доступнум для всех пользователей. Как правило, в этом нет необходимости.

Создаем собственный модуль

Чтобы понять, как устроены дополнения в Node.js, лучше всего не мелочиться и написать собственный модуль. Давайте так и поступим, тем более что модули в Node.js – это не только способ добавления нового функционала. Их нельзя сравнивать, например, с расширениями PHP или с модулями Apache. Модули – это нормальный способ организации приложения Node.js. То есть, создавая свои объекты со сколько-нибудь значимым функционалом, крайне желательно оформлять их в виде модулей, что, помимо прочего, решает проблему повторного использования кода, проблему структурированности приложения, проблему подпространств имен, наконец.

Попробуем продемонстрировать такую организацию кода, написав небольшой пример. Сначала сделаем все в одном файле (**main.js**):

```
var band = function(name) {
    this.name = name;
};
band.prototype.getName = function(){
   console.log(this.name)
}
var myBand1 = new band("The Beatles");
var myBand2 = new band("The Rolling Stones");
myBand1.getName();
myBand2.getName();
```

Тут все очень просто – мы создаем объект Band, задаем его единственное свойство и с помощью прототипа описываем его, пока единственный, метод. Затем последовательно создаем два экземпляра этого объекта и вызываем данный метод у каждого из них (прошу прощения, если я объясняю очевидные вещи, но некоторые особенности языка JavaScript не всегда понятны сразу). Результат:

```
$ node bands/main.js
The Beatles
The Rolling Stones
```

Теперь выделим наш объект в отдельный файл (**band.js**):

```
var band = function(name) {
    this.name = name;
};
band.prototype.getName = function(){
   console.log(this.name)
}
```

И подключим его с помощью уже хорошо знакомого нам метода require:

```
require('./band.js');
var myBand1 = new band("The Beatles");
var myBand2 = new band("The Rolling Stones");
myBand1.getName();
myBand2.getName();
```

Все? Нет, не все. Такая конструкция работать не будет. Мы уже упоминали, что, в отличие от подключаемых скриптов на веб-странице, код на Node.js не имеет глобальных объектов (кроме встроенных или явным образом определенных в глобальном пространстве имен). Переменные, определенные внутри одного модуля, «действительны» только для него, и это на самом деле очень хорошо!

Но что же нам делать для полноценной работы с подключаемым модулем? Можно использовать уже знакомый нам объект global, переведя band в данное пространство имен:

```
band.prototype.getName = function(){
   console.log(this.name)
}global.band = band;
```

Так все будет работать, но мы лишаем себя одного из преимуществ модульности – использования собственного пространства имен. При использовании немного более сложных объектов мы легко можем перезаписать их свойства. Чего, естественно, надо всячески избегать.

К счастью, Node.js предлагает наилучший способ – метод **exports()** глобального объекта module, наследником которого автоматически стал наш модуль. Добавим в него этот метод:

```
band.prototype.getName = function(){
   console.log(this.name)
}
```

```
exports.Band = band;

main.js теперь будет выглядеть так:
var item = require('./band.js');
var myBand1 = new item.Band("The Beatles");
var myBand2 = new item.Band("The Rolling Stones");
myBand1.getName();
myBand2.getName();
```

Как видите, пришлось немного пожертвовать простотой, но зато теперь мы можем пользоваться всеми преимуществами изолированного пространства имен.

Пока все хорошо, но служебные модули и тем более полнофункциональные приложения редко бывают настолько простыми, чтобы умещаться в одном файле. Вернее, конечно, можно разместить все и в одном, но тогда будет очень трудно разбираться с кодом. На счастье, require() вполне справляется с подключением отдельной папки, содержащей модуль. Для этого надо выполнить всего насколько требований, которые мы сейчас и воплотим в жизнь применительно к создаваемому нами приложению.

(Да, если кто не заметил, мы уже начали писать приложение. Что за приложение? Ну, пока действуем по принципу Портоса: сначала ввяжемся в бой, а потом – как получится.)

Для начала создадим папку band и поместим туда наш прототип модуля – файл **band.js**, переименовав его в **index.js**. Зачем – объясню чуть позже. Код этого, теперь главного файла нашего модуля немного изменим. В предметной области, как вы наверняка догадались, речь идет о рок-группах, и нам наверняка понадобится еще одна сущность – музыканты, входящие в их состав. Поэтому добавим эту сущность, описанную отдельным объектом, в отдельном файле (пусть он пока ничего не делает, сейчас его задача – просто не вызвать ошибок). Итак, файл **bands/index.js**:

```
var artist = require('./artist');
var band = function(name) {
    this.name = name;
};
band.prototype.getName = function(){
   console.log(this.name)
}
exports.Band = band;
```

Код в файле **bands/artist.js**:

```
var artist = function(name) {
    this.name = name;
```

```
};
artist.prototype.getName = function(){
   console.log(this.name)
}
exports.Artist = artist;
```

И немного изменим главный файл (**main.js**):

```
var item = require('./bands');
var myBand1 = new item.Band("The Beatles");
```

Теперь самое время пояснить, что при подключении модуля метод **require()** сначала просто ищет одноименный файл, затем, в случае неудачи, ищет файлы с тем же именем и расширением .js, .json, .node (о двух последних форматах речь впереди) и наконец, снова потерпев неудачу, ищет папку с таким именем. Затем механизм приключения модулей находит в ней файл **index.js** и подключает его. Подразумевается, что последний тянет за собой все остальные компоненты модуля.

Чисто технически для нашего приложения теперь встает проблема хранения данных (хотя бы составы групп!). Пока забудем о СУБД и других хранилищах, у нас задача немного проще, и для ее решения отлично подходит другая форма модулей Node.js – JSON-модули. JSON (*JavaScript Object Notation*) – это текстовый формат обмена данными, основанный на JavaScript и, соответственно, «родной» для него. Преимущество этого формата перед, например, XML состоит в его легкой читаемости, не только JavaScript, но и человеком.

Запишем составы этих двух великих групп в нотации JSON (файл **members.json**):

```
{
    "The Beatles": ["John Lennon","Paul McCartney","George Harrison","Ringo Starr"],
    "The Rolling Stones":["Mick Jagger","Keith Richards","Charlie Watts","Ronnie
Wood"]
}
```

(Про Брайана Джонса или про Питера Беста я в курсе, но пока мы не можем себе позволить излишне усложнять данные.)

И переработаем код **index.js**, дополнив его новым методом, не забыв подключить .json-файл с данными:

```
var artist = require('./artist');
var members = require('./members');
var band = function(name) {
    this.name = name;
};
band.prototype.getName = function(){
```

```
  return this.name;
}

band.prototype.getMembers = function(name){
   band = members[name];
   console.log(band);
}
exports.Band = band;
```

Как видите, теперь метод **getName()** ничего не выводит в консоль, а, что более естественно, возвращает значение. Соответственно, придется сделать некоторые изменения в основном файле:

```
var band = require('./bands');

var myBand1 = new band.Band("The Beatles");
var myBand2 = new band.Band("The Rolling Stones");

var bandName = myBand1.getName();
console.log(bandName);
myBand1.getMembers(bandName);

var bandName = myBand2.getName();
console.log(bandName);
myBand2.getMembers(bandName);
```

Результат:

```
$ node main.js
The Beatles
[ 'John Lennon',
  'Paul McCartney',
  'George Harrison',
  'Ringo Starr' ]
The Rolling Stones
[ 'Mick Jagger',
  'Keith Richards',
  'Charlie Watts',
  'Ronnie Wood' ]
```

Тут, конечно, еще есть над чем поработать, наверное, есть необходимость написать методы для более человеческого, форматированного вывода, но это уже детали. Гораздо важнее сделать наш модуль совсем полноценным и понятным для npm.

Зачем это нам нужно, ведь распространять его мы не собираемся? Ну, во-первых, не будем зарекаться, а во-вторых, скорее всего, с расширением функционала нам потребуется связать его зависимостями с какими-то дополнительными модулями, например с тем же **Colors** и с шаблонизатором **Jade** (о последнем мы еще как-нибудь поговорим).

Да и что значит «не собираемся распространять»? Не планируем же мы использовать модуль именно на том же самом компьютере, на котором его и написали? А развертывание можно и нужно автоматизировать.

Для того чтобы npm получил информацию о нашем модуле, достаточно создать в его корневой папке специальный конфигурационный файл **package.js**. Посмотрим, как он выглядит у mongoose (в сокращенном варианте):

```
{
  "name": "mongoose",
  "description": "Elegant MongoDB object modeling for Node.js",
  "version": "3.6.19",
  "author": {
    "name": "Guillermo Rauch",
    "email": "guillermo@learnboost.com"
  },
  "keywords": [
    "mongodb",
    "document",
    ...
  ],
  "dependencies": {
    "hooks": "0.2.1",
    "mongodb": "1.3.19",
    ........
  },
  "directories": {
    "lib": "./lib/mongoose"
  },
  "main": "./index.js",
  "engines": {
    "node": ">=0.6.19"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/LearnBoost/mongoose.git"
  },
  "homepage": "http://mongoosejs.com",
}
```

Основные, обязательные поля тут – имя, описание и версия модуля. Также очень важны **dependencies** – перечисление зависимостей и **main** – точка входа в модуль (да, index.js – это всего лишь установка по умолчанию, ее можно изменить).

Писать package.js «от руки» в настоящее время необходимости нет, его можно создать с помощью самого npm. Зайдем в папку приложения и выполним следующую команду:

```
$ npm init
```

Далее в консоли последует интерактивный диалог, в котором будут запрошены параметры оформляемого модуля. В случае обязательности поля в скобках будут указаны значения по умолчанию (достаточно очевидные):

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (bands) Bands
version: (0.0.0) 0.0.1
description: Test Aplication
entry point: (index.js)
test command:
git repository:
keywords: test
author: k.sukhov
license: (BSD)

Is this ok? (yes) yes
npm WARN package.json Bands@0.0.1 No repository field.
npm WARN package.json Bands@0.0.1 No readme data.
```

Как видите, все прошло успешно. В последних строчках npm посетовал на отсутствие указания на дистрибутив и пояснительные сведения (readme), но тем не менее в папке модуля у нас появился файл **package.js**:

```
{
  "name": "Bands",
  "version": "0.0.1",
  "description": "Test Aplication",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": "",
  "keywords": [
    "test"
  ],
  "author": "k.sukhov",
```

```
  "license": "BSD"
}
```

Добавим сюда следующие строчки:

```
  "author": "k.sukhov",
  "license": "BSD",
  "dependencies": {
    "colors": "*",
    "jade": "*",
  }
}
```

продекларировав зависимость нашего проекта от модулей Colors и Jade. Теперь дело за малым – инсталлировать модуль, пока на собственный компьютер, попутно проверив работоспособность.

```
$ npm install
```

Если все сделано правильно, то в папке нашего модуля должна появиться подпапка **/node_modules** с необходимыми для его работы инструментами.

А теперь по-взрослому – пишем Си++ Addons

Как и для всякого расширяемого продукта, для Node.js существует API, с помощью которого можно сделать расширение базовых функциональных возможностей. Процесс создания такого модуля (C/C++ Addons), в принципе, не очень прост и требует основательного знания нескольких библиотек:

- **V8 JavaScript C++ library** – используется для интерфейса с JavaScript: создания объектов, вызовов функций и т. д.;
- **libuv** – библиотека, написанная на языке C, реализующая event loop;
- внутренние библиотеки Node.js (прежде всего **node::ObjectWrap**).

Для сборки такого модуля необходим процесс создания Makefайла, связанный с множеством зависимостей сборки движка V8 и Node.js.

На наше счастье, все можно упростить – существует вспомогательный инструмент, который автоматизирует этот процесс, – node-gyp. Это кросс-платформенный command-line-инструмент для компиляции «нативных» addon-модулей для Node.js. Проект node-gyp был

инициирован Google, а также поддержан компаниями Bloomberg Finance и Yandex. Для работы node-gyp требуется наличие python.

Сам node-gyp – это модуль Node.js, и устанавливается он обычным способом:

```
$ npm install -g node-gyp
```

Флаг **-g** тут обязателен по очевидным причинам.

Теперь пишем код. Задача у нас не очень сложная. Нужно создать модуль с методом **hello()**, выводящим строку приветствия. На JavaScript с этим бы справился следующий код:

```
module.exports.hello = function() {
    return 'Node.js addon!';
};
```

Но мы уже отказались от легких путей, поэтому пишем на C++:

```
#include <node.h>
#include <v8.h>

using namespace v8;

Handle<Value> MyMethod(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("Node.js addon!"));
}

void init(Handle<Object> exports) {
  exports->Set(String::NewSymbol("hello"),
      FunctionTemplate::New(MyMethod)->GetFunction());
}


NODE_MODULE(hello, init)
```

Сначала мы подключаем заголовочные файлы Node.js и V8. Затем объявляем пространство имен. Далее объявляем обработчик – **MyMethod()**, в котором определяем V8 контейнер (scope) и назначаем ему вернуть требуемую строку текста.

Это, собственно, реализация интерфейса модуля, в данном случае его единственного метода.

Работа любого модуля для node.js начинается с выполнения макроса **NODE_MODULE** (последняя строчка нашей программы), в который передаются имя модуля, которое потом будет вызываться в **require()** (hello), и имя функции (init), которая выполняется в момент подключения модуля.

Init() – это функция регистрации модуля, в которой описывается его интерфейс. Она вызывается один раз при подключении модуля.

Каждый новый метод модуля задается методом **Set()** объекта интерфейса (target):

```
target->Set(String::NewSymbol("test"), FunctionTemplate::New(MethodTest)->
GetFunction());
```

Но мы пока ограничимся одним методом.

Теперь необходимо создать сценарий сборки. Он описывается в файле **binging.gyp**:

```
{
  "targets": [
    {
      "target_name": "hello",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

Тут все просто: target_name – это название модуля, sources – массив исходных файлов (у нас один). Теперь выполним команду **node-gyp configure**:

```
:
$ node-gyp configure

gyp info it worked if it ends with ok

gyp info using node-gyp@1.0.1

gyp info using node@0.10.26 | linux | x64

gyp info spawn python

gyp info spawn args [ '/usr/local/lib/node_modules/node-gyp/gyp/gyp_main.py',

gyp info spawn args   'binding.gyp',

gyp info spawn args   '-f',

gyp info spawn args   'make',

gyp info spawn args   '-I',

gyp info spawn args   '/var/node/build/config.gypi',

gyp info spawn args   '-I',
```

```
gyp info spawn args   '/usr/local/lib/node_modules/node-gyp/addon.gypi',
gyp info spawn args   '-I',
gyp info spawn args   '/home/ref-deploy/.node-gyp/0.10.26/common.gypi',
gyp info spawn args   '-Dlibrary=shared_library',
gyp info spawn args   '-Dvisibility=default',
gyp info spawn args   '-Dnode_root_dir=/home/ref-deploy/.node-gyp/0.10.26',
gyp info spawn args   '-Dmodule_root_dir=/var/node',
gyp info spawn args   '--depth=.',
gyp info spawn args   '--no-parallel',
gyp info spawn args   '--generator-output',
gyp info spawn args   'build',
gyp info spawn args   '-Goutput_dir=.' ]
gyp info ok
```

Если не вникать в тонкости, то для нас важна только последняя строчка вывода – у нас все получилось. После выполнения команды мы получим в корне проекта новую папку build, в которой содержатся следующие файлы:

```
build/
    binding.Makefile
    config.gypi
    hello.target.mk
    Makefile
```

Теперь для компиляции модуля достаточно выполнить Makefile, но тут есть риск, что придется самостоятельно разбираться с зависимостями. Лучше воспользоваться специальной командой node-gyp build:

```
$ node-gyp build
gyp info it worked if it ends with ok
gyp info using node-gyp@1.0.1
gyp info using node@0.10.26 | linux | x64
gyp info spawn make
```

```
gyp info spawn args [ 'BUILDTYPE=Release', '-C', 'build' ]
make: Entering directory `/var/node/build'
  CXX(target) Release/obj.target/hello/hello.o
  SOLINK_MODULE(target) Release/obj.target/hello.node
  SOLINK_MODULE(target) Release/obj.target/hello.node: Finished
  COPY Release/hello.node
make: Leaving directory `/var/node/build'
gyp info ok
```

Опять же, последняя строчка нам сообщает о полном успехе задуманного. В папке build/Release теперь должен появиться скомпилированный модуль **hello.node** (а также зависимости, объектные файлы и т. д.). Для проверки созданного модуля напишем и запустим простенький сценарий, в котором просто подключим модуль и выполним единственный метод **(test_addon.js)**:

```
var addon = require('./build/Release/hello');

console.log(addon.hello());
```

Результат:

```
$ node  test_addon.js
Node.js addon!
```

Как видите, наш модуль совершенно рабочий.

На этом мы остановимся: создание полноценных addon-модулей – дело не очень простое, требующее изучения многих нюансов. Кроме соответствующего раздела руководства, для старта очень рекомендую статью программиста из Санкт-Петербурга Александра Календарева «Создание addon-модулей для Node.js» [4] (спасибо ему за помощь при написании этого раздела).

Теперь модуль полностью готов, и его даже можно опубликовать, но мне кажется, что, прежде чем радовать мир своими Node.js-шедеврами, мы должны разобраться еще с несколькими вещами. Например, с TCP- и веб-сокетами, сетью, хранилищами данных, работой с файловой системой... словом, со всем остальным.

Работа с файлами

Именно в этом месте изучения Node.js следует окончательно забыть, что JavaScript – это язык, предназначенный для исполнения клиентских сценариев в браузере. Тут мы вторгаемся в ту область, в которую подобные клиент-сайт-украшательства вторгаться не должны по определению. А мы будем! В путь.

В дебри файловой системы

Модуль FileSystem входит в дистрибутив Node.js, и, честно говоря, мы его уже использовали – для чтения html-контента. Естественно, этим его функции не ограничиваются; лучше всего показать его работу на примере конкретной задачи. Причем важной. Скажем, составить список всех композиций в формате mp3 у меня на винчестере с их местоположением.

Наверняка вам нечто подобное уже приходилось писать на C или, скажем, на Perl. Теперь очередь JavaScript, и мы сейчас убедимся, что этот язык справится с данной задачей ничуть не хуже:

```
var fs = require('fs');
var base = 'G:\muzic'; // тут хранится музыка
function readDir(base){
fs.readdir(base, function(err, files) {
    files.forEach( function(item){
        fs.stat(base+'/'+item, function(err,state){
        if(state.isDirectory()){
            console.log(item);
            localBase = base+'/'+item;
            readDir(localBase);
        }else{
            console.log("    "+item);
          }
        });
    })
});
}
readDir(base);
```

Тут мы написали функцию, осуществляющую рекурсивный обход каталогов и выводящую консоль названия их и содержащихся в них файлов. При этом сначала используем метод **fs.readdir()**, имеющий аналоги во многих языках программирования. Особенность использования **readdir()** в node традиционна для этой платформы – резуль-

тат (массив файлов) передается в функцию обратного вызова. Для определения, является ли полученный файл директорией, используется объект **fs.Stats**, возвращаемый методом **fs.stat()**. Это объект, содержащий различную информацию о найденном файле. Если мы выведем его значение в консоль, получим что-то вроде следующего:

```
{ dev: 0,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  ino: 0,
  size: 7854984,
  atime: Fri Mar 21 2014 00:10:45 GMT+0400 (Russian Standard Time),
  mtime: Wed May 09 2012 12:18:53 GMT+0400 (Russian Standard Time),
  ctime: Fri Mar 21 2014 00:10:45 GMT+0400 (Russian Standard Time) }
```

Что обозначают эти данные, я думаю, догадаться несложно. Они, конечно, часто бывают нужны, но в нашей задаче практически бесполезны. Чего не скажешь о методах, которые предоставляет этот объект:

- **stats.isFile()** – проверяет, является ли объект файлом;
- **stats.isDirectory()** – проверяет, является ли объект директорией;
- **stats.isBlockDevice()** – проверяет, является ли объект файлом устройства блочного ввода/вывода;
- **stats.isCharacterDevice()** – проверяет, является ли объект файлом устройства посимвольного ввода/вывода;
- **stats.isSymbolicLink()** – проверяет, является ли объект символической ссылкой (при этом для получения stat должен быть использован специальный метод – **fs.lstat()**);
- **stats.isFIFO()** – проверяет, является ли объект *FIFO-файлом* (именованным каналом);
- **stats.isSocket()** – проверяет, является ли объект сокетом.

Этого арсенала должно хватить, чтобы получить информацию для любых традиционных манипуляций с файловой системой. Все это замечательно, но вот вывод данной программы нас может не устроить. Я не буду его приводить, верьте мне на слово, это слабоупорядоченная смесь названий файлов и директорий, ориентироваться в которой просто нельзя. Почему это случилось? Дело в том, что методы **fs.readdir()** и **fs.stat()** асинхронны и совсем не обязаны выдавать результат в строгой очередности. Для целого ряда задач (например,

нам бы понадобилось массово переименовать файлы или просканировать их содержимое) такой подход не только уместен, но и наиболее эффективен. Но вот нам он не подходит. На счастье, многие ключевые методы модуля **fs** имеют свои синхронные аналоги. В том числе **fs.readdir()** и **fs.stat()**. Перепишем нашу программу в синхронном стиле:

```
var fs = require('fs');
var base = 'G:\muzic';
String.prototype.repeat = function( num )
{
    return new Array( num + 1 ).join( this );
}
function readDir(base,level){
files = fs.readdirSync(base);
    files.forEach( function(item){
        state = fs.statSync(base+'/'+item);
        if(state.isDirectory()){
            console.log("\n"+" ".repeat(level*2)+item+"\n");
            localBase = base+'/'+item;
            readDir(localBase, level+1);
         }else{
            console.log(" ".repeat(level*2)+item);
          }
});
    //console.log(files);
}
readDir(base, 0);
```

Нам пришлось предпринять некоторые меры для форматирования вывода (обозначить вложенность папок отступами), но это не принципиально. Главное, что названия папок и файлы выводятся в правильном порядке:

```
Popol Vuh

  1995 - City Raga

    01. Wanted Maya.mp3
    02. Tears of Concrete.mp3
    03. Last Village.mp3
    04. City Raga.mp3
    05. Morning Raga.mp3
    06. Running Deep.mp3
    07. City Raga (Mystic House Mix).mp3
    Back.jpg
    Front.jpg

  1997 - Shepherd's Symphony - Hirtensymphonie [2004 SPV reissue]
```

```
    01. Shepherds Of The Future.mp3
    02. Short Visit To The Great Sorcerer.mp3
    03. Wild Vine.mp3
    04. Shepherd's Dream.mp3
    05. Eternal Love.mp3
    06. Dance Of The Menads.mp3
    07. Yes.mp3

    covers

      CD.jpg
      Digipack - credits.jpg
      Digipack.jpg
      front.jpg
      [1997] Shepherd's Symphony - Hirtensymphonie - 1024x1024.jpg
      [1997] Shepherd's Symphony - Hirtensymphonie - 200x200.jpg
      [1997] Shepherd's Symphony - Hirtensymphonie - 600x600.jpg

Comus

  Comus - 1971 - Diana (Maxi Single)

    01. Diana.mp3
    02. In the Lost Queen's Eyes.mp3
    03. Winter Is A Coloured Bird.mp3
    Cover.jpg
.................................................
```

Так гораздо лучше.

Теперь для разминки можно сделать нечто осмысленное – разобрать нашу музыкальную коллекцию, расположив всех исполнителей по папкам в алфавитном порядке. Для этого придется освоить несколько новых методов:

```
var fs = require('fs');
var path = require('path');
var base = 'G:\muzic'; // тут хранится музыка
var collection = 'G:\collection';
                //тут будет упорядоченная коллекция
function collect(base){
  fs.readdir(base, function(err, files) {
    files.forEach( function(item){
      console.log(item.charAt(0));
      var fileName = collection+'/'+item.charAt(0);
      if(fs.existsSync(fileName)){
        copyRecursive(base+"/"+item,fileName+"/"+item);
      } else {
        fs.mkdir(fileName, function(){
          copyRecursive(base+"/"+item, fileName+"/"+item);
        });
```

```
      }
    });
  });
}
```

Тут мы пользуемся синхронной версией метода (**fs.existsSync()**), проверяющего существование объекта файловой системы (есть и асинхронный). Метод **fs.mkdir()** предсказуемо создает директорию, а вот с методом **copyRecursive()** сложнее. Такого в документации нет, это самодеятельность. В составе **fs** вообще нет аналога posix команды **copy()**, и, если подумать, это закономерно. Есть, правда, несколько отдельных модулей, решающих синхронное рекурсивное копирование каталогов и их содержимого, но мы поступим проще:

```
var copyRecursive = function(src, dest) {
  var exists = fs.existsSync(src);
  var stats = exists && fs.statSync(src);
  var isDirectory = exists && stats.isDirectory();
  if (exists && isDirectory) {
    fs.mkdirSync(dest);
    fs.readdirSync(src).forEach(function(childItemName) {
      copyRecursive(path.join(src, childItemName),
                    path.join(dest, childItemName));
    });
  } else {
    fs.linkSync(src, dest);
  }
};

collect(base)
```

Что тут нового? Ну, в самом приеме рекусивного обхода ресурсов файловой системы точно нет никаких инноваций. А вот на что стоит обратить внимание, так это на метод создания директории (**fs.mkdirSync()**, он тоже имеет синхронную форму) и метод **fs.linkSync()**, создающий, по идее, жесткую ссылку на файл, но в данной ситуации это соответствует процедуре копирования. Естественно, последний метод тоже имеет синхронный аналог, а есть еще методы **fs.symlink** и **fs.symlinkSync**, создающие символические ссылки.

Но забудем о ссылках. У нас файлы музыкальных треков уже упакованы в коллекцию, и нам можно уничтожить свалку, которую мы разобрали, – потереть исходные файлы и директории. Попытка сделать это средствами Node.js «в лоб» (файл **delete.js**):

```
var fs = require("fs");
var path = 'G:\muzic';

fs.rmdir(path, function(error) {
```

```
  if (error) {
    console.error(error.message);
  }
});
```

ожидаемо потерпит неудачу:

```
$ node delelete.js
rmdir error:  ENOTEMPTY, rmdir 'G:\muzic'
```

Все правильно, директория не пуста. Тут тоже придется прибегнуть к рекурсии:

```
var fs = require("fs");
var path = 'G:\muzic';

function removeDir(path) {
  if (fs.existsSync(path)) {
    fs.readdirSync(path).forEach(function(file) {
      var f = path + "/" + file;
      var stats = fs.statSync(f);
       if (stats.isDirectory()) {
        removeDir(f);
      } else {
        fs.unlinkSync(f);
        console.log( f+ " is removed");
      }
    });
    fs.rmdirSync(path);
    console.log( path+ " is removed");
  }
}

removeDir(path);
```

Теперь другое дело:

```
$ node delelete.js
G:muzic/Comus/Comus - 1971 - First Utterance/Art/Tray.jpg is removed
G:muzic/Comus/Comus - 1971 - First Utterance/Art is removed
G:muzic/Comus/Comus - 1971 - First Utterance is removed
G:muzic/Comus/Comus - 1971 - First Utterance/01. Diana.mp3 is removed
.....
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie/06. Dance Of The
Menads.mp3 is removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie/07. Yes.mp3 is
removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie/covers/CD.jpg is
removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie /covers/Digipack -
credits.jpg is removed
```

```
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie \/covers/Digipack.
jpg is removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie /covers/front.jpg
is removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie - 1024x1024.jpg is
removed
G:muzic/Popol Vuh/1997 - Shepherd's Symphony - Hirtensymphonie
/covers is removed
G:muzic/Popol Vuh is removed
G:muzic is removed
```

При операции с файловой системой мы опять не обошлись без модуля path, и, собственно, самое время остановиться на нем чуть подробнее.

Маленький полезный модуль – Path

Этот модуль по-настоящему делает только одно действие – обработку и преобразование путей к файлам. Причем он не обращается к файловой системе, мудуль производит только семантические операции. Но делает их замечательно!

Самый, наверное, интересный метод из его небольшого арсенала – это **path.resolve()**, разрешающий (преобразующий) заданный путь в абсолютный:

```
var path = require('path');
var resosolve = path.resolve('./baz');
console.log(resosolve);
```

Результат:

```
$ node path.js
/home/geol/node/file/baz
```

Метод имеет еще необязательные аргументы (причем стоящие перед основным), задающие старт для определения пути. Если заданный путь не является абсолютным, то к нему добавляют пути справа налево из предшествующих аргументов – до тех пор, пока полученный путь не будет абсолютным. Если в итоге путь останется относительным, он будет разрешён относительно рабочей директории.

Полученный путь нормализуется, и у него удаляется завершающий слэш, если, конечно, это не корневая директория в Unix-like-системах:

```
var resosolve = path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
console.log(resosolve);
```

Результат:

```
$ node path.js
/home/geol/node/file/wwwroot/static_files/gif/image.gif
```

Документация предлагает такой прием для понимания механизма работы этого метода: считать, что он последовательно выполняет команду **cd** и возвращает конечный путь, то есть

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

возвращает то же самое, что и последовательность команд:

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

Метод **path.relative()**, преобразующий заданный путь в относительный, неплохо дополняет предыдущий:

```
var myPath = require('path');
var resosolve = path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb');
console.log(resosolve);
```

Результат:

```
C:\Users\Geol\node\node path.js
..\..\impl\bbb
```

Это, как вы понимаете, была операционная система семейства Windows. А вот пример для Unix-like:

```
var myPath = path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
console.log(resosolve);
```

Результат:

```
$ node path.js
./../impl/bbb
```

В общем, работу этих двух методов можно показать так:

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

Метод **path.normalize()** «приводит пути в порядок», то есть удаляет из них все, что там быть не должно, но появилось, например из-за специфичного формата ввода (сочетания символов .., ., или //):

```
var path = require('path');
var myPath = '/foo/bar//baz/asdf/quux/..';
myPath = path.normalize(myPath);
console.log(myPath);
```

Результат:

```
$ node path.js
\foo\bar\baz\asdf
```

Метод **path.join()** позволяет соединять пути в файловой системе:

```
var path = require('path');
var myPath = path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
console.log(myPath);
```

Результат:

```
$ node path.js
/foo/bar/baz/asdf
```

Ничего особенного? Зато все по делу! Ну и еще разные полезные мелочи:

- **path.extname()** – определяем расширение файла:

  ```
  var path = require('path');
  console.log(path.extname('/home/geol/work/listOfVendors.json'));
  console.log(path.extname('http://store.com/listOfVendors.jsp'));
  ```

  Результат:

  ```
  $ node path.js
  .json
  .jsp
  ```

- **path.sep()** – определяем специфичный для платформы разделитель в пути к файлу:

  ```
  var path = require('path');
  console.log(path.sep);
  ```

  Результат:

  ```
  $ node path.js
  /
  ```

- **path.delimiter()** – определяем специфичный для платформы разделитель путей:

  ```
  var path = require('path');
  console.log(path.delimiter);
  console.log(process.env.PATH);
  console.log(process.env.PATH.split(path.delimiter));
  ```

  Результат (тут для разнообразия опять ОС Windows):

  ```
  C:\Users\Geol\node\file>node path.js
  ;
  C:\Windows\system32;C:\Windows;C:\W
  ```

```
indows\System32;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program
Files\nodejs\;C:\Program Files (x86)\Git\cmd;C:\Program Files (x86)\Git\bin
[  'C:\\Windows\\system32',
  'C:\\Windows',
  'C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\',
  'C:\\Program Files\\nodejs\\',
  'C:\\Program Files (x86)\\Git\\cmd',
  'C:\\Program Files (x86)\\Git\\bin' ]
```

- **path.dirname()** – определяем имя директории, содержащей файл:

```
var path = require('path');
console.log(path.dirname('/home/geol/work/listOfVendors.json'));
console.log(path.dirname('listOfVendors.json'));
```

Результат:

```
$ node path.js
/home/geol/work
.
```

- **path.basename()** – определяем базовое имя файла:

```
var path = require('path');
var base = path.basename('/foo/bar/baz/asdf/index.html');
console.log(base);
base = path.basename('/foo/bar/baz/asdf/index.html', '.html')
console.log(base);
```

Результат:

```
$ node path.js
index.html
index
```

Бродим по папкам

Прежде чем начать свой путь по файловой системе, нужно немного задержаться – изучить две глобальные переменные платформы Node.js – **__dirname** и **__filename**. Первая хранит имя текущей директории, вторая – текущего файла.

```
console.log(__dirname);
console.log(__filename);
```

Результат:

```
C:\Users\Geol\node\file
C:\Users\Geol\node\file\fs3.js
```

Теперь мы точно не заблудимся. Правда, спешу немного разочаровать. Перемещаться по файловой системе (то есть менять рабочую папку) модуль **fs** не поможет. Это операция «ядерного уровня», она доступна через процессы. Например, так можно узнать текущую директорию:

```
console.log("The current working directory is " + process.cwd());
```

Результат:

```
$ node pr.js
The current directory is C:\Users\Geol\node\file
```

А так её можно поменять:

```
console.log("The current directory is " + process.cwd());
try {
  process.chdir("../img");
  console.log("The new current directory is " + process.cwd());
} catch (exception) {
  console.error("chdir error:  " + exception.message);
}
```

Результат:

```
$ node pr.js
The current directory is C:\Users\Geol\node\file
The new current directory is C:\Users\Geol\node\img
```

И еще один метод модуля process, который может быть нам полезен:

```
console.log(process.execPath);
```

Так мы получаем путь к текущему исполняемому процессу Node.js:

```
$ node pr.js
C:\Program Files\nodejs\node.exe
```

Работа с файлами – вот она, асинхронность

Читаем файлы

Чтение и запись в файл производятся также в двух режимах, синхронном и асинхронном. Впрочем, все по порядку. Доступ к файлу осуществляется следующим образом:

```
var fs = require("fs");
var path = "text.txt";
```

```
fs.open(path, "r+", function(error, fd) {
    if (error) {
        console.error("open error: " + error.message);
    } else {
        console.log("Successfully opened " + path);
        fs.close(fd, function(error) {
            if (error) {
                console.error("close error: " + error.message);
            }
            else {
                console.log("Successfully closed " + path);
            }
        });
    }
});
```

Метод **fs.open()** в качестве первого параметра принимает имя файла, последним служит функция обратного вызова, а вторым – флаг режима открытия, который в Node.js имеет свои особенности. Ниже приведены его возможные значения:

- **r** – открыть для чтения. Генерирует исключение при отсутствии файла;
- **r+** – открыть для чтения и записи. Генерирует исключение при отсутствии файла;
- **rs** – открыть для чтения в синхронном режиме;
- **rs+** – открыть для чтения и записи в синхронном режиме;
- **w** – открыть для записи. Если файл не существует, он будет создан. Если файл существует, его содержимое будет очищено;
- **w+** – открыть для чтения и записи. Если файл не существует, он будет создан. Если файл существует, его содержимое будет очищено;
- **a** – открыть для записи в конец файла. Если файл не существует, он будет создан;
- **a+** – открыть для чтения и записи в конец файла. Если файл не существует, он будет создан.

Существует еще один, третий, необязательный параметр – модификатор доступа. По умолчанию это вполне демократичные 666.

Запись и чтение на низком уровне происходят следующим образом:

```
var fs = require("fs");
var path = "text.txt";
fs.open(path, "r+", function(error, fd) {
    if (error) {
        console.error("open error: " + error.message);
```

```
    } else {
        console.log("Successfully opened " + path);
          fs.stat(path, function(error, stats) {
              var buffer = new Buffer(stats.size);
              fs.read(fd, buffer, 0, buffer.length, null,
                  function(error, bytesRead, buffer) {
                    var data = buffer.toString("utf8");
                    console.log(data);
                  });
           });
    }
});
```

Метод **fs.read()**, получая в качестве аргумента дескриптор файла, читает данные из него, «как есть», то сеть, в общем случае, в виде бинарных данных. Для того чтобы их получить, мы сначала создаем буфер (для того чтобы определиться с его размером, нам опять потребовался объект **fs.stat()**), читаем в него данные и преобразуем их в строковой формат перед выводом в консоль. Второй аргумент функции обратного вызова метода **fs.read()** – это количество прочитанных байтов.

Запись в файл происходит по той же схеме (давайте сымитируем полезную деятельность и заставим программу писать в файл логи обращения к ней):

```
console.log(data);
var logItem = "Note created "+Date.now()+"\n";
buffer = new Buffer(logItem);
fs.write(fd, buffer, 0, buffer.length, null,
    function(error, bytesWritten, buffer) {
        if(error){
            console.error(error.message);
        } else {
            console.log("Wrote "+bytesWritten+" bytes.");
        }
     });
```

Тут мы создаем свой буфер из заданной строки и пишем его в файл. Все очень просто и универсально, но, честно говоря, не совсем удобно. По крайней мере, для текстовых данных. На счастье, модуль **fs** располагает более высокоуровневыми методами:

```
var fs = require("fs");
var path = "text.txt";

fs.readFile(path, "utf8", function(error, data) {
    if (error) {
        console.error(error.message);
```

```
    } else {
        console.log(data);
    }
});
```

Это все – не надо заботиться о получении файлового дескриптора и подготовке буфера – все это уже инкапсулировано в метод **readFile()**. Есть метод и для записи:

```
        console.log(data);
        var logItem = "Note created "+Date.now()+"\n";
        fs.writeFile(path, logItem, function(error) {
            if (error) {
                console.error(error.message);
            } else {
                console.log("Successfully wrote " + path);
            }
        });
```

Так гораздо удобнее, правда? Единственный недостаток последнего фрагмента кода заключается в том, что все предыдущие записи в файл будут затерты. **fs.writeFile()** тоже позволяет указывать флаги открытия, но если мы действительно пишем логи, лучше воспользоваться специальным методом **appendFile()**:

```
        var logItem = "Note created "+Date.now()+"\n";
        fs.appendFile(path, logItem, function(error) {
            if (error) {
                console.error(error.message);
            } else {
                console.log("Successfully wrote " + path);
            }
```

В полном соответствии с логикой построения модуля **fs** все эти методы имеют свои синхронные аналоги – **readFileSync()**, **writeFileSync()** и **appendFileSync()**.

Watching Files

Это, наверное, самая интересная возможность модуля **fs**. С помощью метода **fs.watch()** мы можем отслеживать состояние файлов. Например, нашего файла логов. Создадим следующий код (файл **fileWatch.js**):

```
var fs = require("fs");
var path = "text.txt";

fs.watch(path, {
            persistent: true
```

```
    },
    function(event, filename) {
        console.log(event);
      if (event === "rename") {
          console.log("The file was renamed/deleted.");
      } else if (event === "change") {
          console.log("The file was changed.");
      }
});
```

Второй (не обязательный) аргумент метода **fs.watch()** указывает опции наблюдения. В данный момент на 100% доступен только один параметр – persistent, определяющий, будет ли наблюдение постоянным (true – значение по умолчанию). Запустим код на исполнение:

```
$ node fileWatch.js
```

А теперь попробуем внести в log.txt изменения, выполнив в другой консоли предыдущий пример, вот что мы можем наблюдать:

```
change
The file was changed.
change
The file was changed.
```

Далее из файлового менеджера попробуем переименовать файл, а затем вернуть прежнее название. **fs.watch()** все видит!

```
rename
The file was renamed
rename
The file was renamed
change
The file was changed.
```

Доступность той или иной информации о файле зависит от используемой операционной системы – Node.js для получения данных использует именно системные средства. В операционной системе Linux это подсистема ядра inotify, в BSD и OS X – интерфейс уведомления о событиях **kqueue**, в семействе Windows применяется вызов функции **ReadDirectoryChangesW**.

Метод **fs.watch()** возвращает особый объект – **fs.FSWatcher**, и код из последнего примера удобнее переписать так:

```
var fs = require("fs");
var path = "text.txt";
var watcher;
watcher = fs.watch(path);
watcher.on("change", function(event, filename) {
```

```
    if (event === "rename") {
        console.log("The file was renamed/deleted.");
    } else if (event === "change") {
        console.log("The file was changed.");
    }
    watcher.close();
});
```

В таком виде мы гораздо легче можем управлять наблюдением. Хорошо. А сейчас пойдем дальше.

Потоки – унифицируем работу с источниками данных

Поток – это также понятие, непривычное для обычного клиентского JavaScript. Между тем использование потоков (streams) – это универсальный способ работы с различными источниками данных.

ReadStream/WriteStream

С чтением потока и записью в поток мы уже сталкивались, правда, не явным образом. Например, при работе с модулем HTTP мы осуществляли чтение входного потока **http.request** и запись в исходящий поток **http.response**, вне зависимости от направления мы использовали при этом объекты модуля **stream** – **stream.ReadStream** и **stream.WriteSream**. К этим же объектам может обращаться модуль **fs** при чтении/записи файла как потока:

```
var fs = require("fs");
var stream = fs.createReadStream("steam.mp3");
    stream.on("open", function(fd) {
    fs.fstat(fd, function(error, stats) {
        if (error) {
            console.error("fstat error: " + error.message);
        } else {
            console.log(stats);
        }
    });
});
```

Результат:

```
$ node stream.js
{ dev: 0,
  mode: 33206,
  nlink: 1,
  uid: 0,
```

```
  gid: 0,
  rdev: 0,
  ino: 0,
  size: 54401148,
  atime: Sat Mar 29 2014 09:44:21 GMT+0400 (Russian Standard Time),
  mtime: Sat Nov 30 2013 14:11:12 GMT+0400 (Russian Standard Time),
  ctime: Sat Mar 29 2014 09:44:21 GMT+0400 (Russian Standard Time) }
```

Пока особой разницы не наблюдается. Но если мы немного модифицируем код, можно будет понять, что сейчас мы имеем дело именно с потоком:

```
stream = fs.createReadStream("teststream.mp3");
    stream.on('readable', function() {
        console.log('read');
     });
     stream.on('end', function() {
        console.log('end');
     });
     stream.on('data', function(chunk) {
      console.log('got %d bytes of data', chunk.length);
    })
```

Результат:

```
$ node stream.js
read
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
.............................. . . . .
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 65536 bytes of data
read
got 6268 bytes of data
end
```

Чтение потока начинается с создания объекта и подключения к источнику данных. Когда порция данных прочитана в буфер созданного потока, наступает событие **readable**. Мы могли бы (если бы это имело смысл) в этот момент вывести данные, примерно таким образом:

```
stream.on('readable', function() {
    console.log('read');
    console.log(stream.read());
});
```

Вместо этого мы воспользуемся другим событием – **date**, наступающим при получении порции данных, и выведем сведения об их размере.

Оба этих действия освобождают буфер потока, и снова происходит получение данных до очередного события **readable**.

Событие **end**, как нетрудно догадаться, наступает при окончании получения данных из потока.

Поток можно в любой момент закрыть, вызвав метод **stream.close()**:

```
stream.on('data', function(chunk) {
 if( chunk.length <10){
   stream.close();
 }
})
```

А еще для более гибкой работы с потоком присутствуют методы **stream.pause()** и **stream.resume()**:

```
var readable = getReadableStreamSomehow();
stream.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
  stream.pause();
  console.log('there will be no more data for 10 second');
  setTimeout(function() {
    console.log('now data will start flowing again');
    stream.resume();
  }, 10000);
})
```

На примере чтения файла с использованием интерфейса потока хорошо видно основное преимущество использования этого механизма. Чтобы получить данные, нам не надо сразу загружать в оперативную память огромных размеров файл, можно получать и обрабатывать данные по частям, приемлемого размера и прекратить их передачу в любой нужный момент.

Веб-сервер на потоках

Освоив потоки, мы теперь можем более рационально переписать наш веб-сервер, описанный в начале этой книги. В чем его нерациональность? Ну хотя бы в том, что при запросе браузером очень больших файлов (а такая ситуация вполне обычна) мы вынуждены до отдачи данных целиком считывать его в память, что недопустимо для сколько-либо серьезно нагруженного веб-сервера:

```
} else {
 /* fs.readFile(pathname, 'utf8', function(err, data) {
        if (err){
            console.log('Could not find or open file '+
                        pathname +' for reading\n');
        } else {
            console.log(pathname+" "+mimeType);
            response.write(data);
            response.end();
        }
    }) */
    var stream = fs.createReadStream(pathname,
                                     {encoding: 'utf8'}
                                     );
    stream.on('readable', function() {
        data = stream.read();
        response.write(data);
    });
    stream.on('end', function() {
        response.end();
     });

}
```

В таком виде все работает, причем корректно, но на самом деле это только полдела. Даже меньше. Смотрите, мы действительно читаем данные из входящего потока, но затем перед записью в исходящий поток сохраняем их в переменную, что, строго говоря, сводит на нет все преимущество использования интерфейса потоков. Конечно, можно писать напрямую в исходящий поток, но размер его буфера – величина конечная, и при его заполнении надо ждать освобождения (например, с отсылкой очередной порции данных, событие **drain**). С этой и некоторыми другими проблемами, конечно, можно справиться, более того, для программиста это по-настоящему «вкусная» и интересная задача, но тут я вынужден разочаровать – она уже давно решена. Смотрим на модифицированный код ниже:

```
} else {
      var stream = fs.createReadStream(pathname, {encoding: 'utf8'});
```

```
        stream.on('readable', function() {
            // data = stream.read();
            // response.write(data);
            stream.pipe(response);
        });
          stream.on('end', function() {
            response.end();
        });
}
```

Мы видим, что у нас появился новый метод – **stream.pipe()** (если точнее, **readable.pipe()**). Его задача – создание канала, соединяющего входящий поток чтения (**ReadStream**) с исходящим потоком записи (**WritableStream**), переданным в качестве параметра.

То есть в данном случае все, что пришло на входящий поток, будет записано в поток исходящий (**response**).

Теперь наш сервер работает как надо. Правда, для надежной работы необходимо предусмотреть обработку обрыва соединения. Действительно – тогда поток останется незакрытым, что на реальном вебсервере совершенно недопустимо! Поэтому добавим следующий код:

```
    response.on('close', function(
       stream.destroy();
});
```

(За это уточнение – спасибо Илье Кантору и его учебным материалам [2].)

На этом пока про потоки закончим, но будьте уверены, мы с ними столкнемся еще не раз.

Сервер HTTP, и не только

В составе Node.js довольно много средств реализации сетевого взаимодействия, что неудивительно, учитывая начальное предназначение этой системы. Собственно, в первой статье этого цикла [1] мы и начали её изучение с построения простого HTTP-сервера. Теперь давайте разберемся в вопросе несколько капитальнее и подробно рассмотрим различные сетевые ипостаси Node.js. И начнем с фундаментального уровня.

Создаем TCP-сервер

Протокол TCP, как известно, является базовым для большинства интернет-приложений. На платформе Node.js он реализован в модуле net, входящем в ядро системы. Построить TCP-сервер – задача довольно тривиальная. В отличие от HTTP-сервера, функция обратного вызова, являющаяся аргументом при создании TCP-сервера, принимает только один аргумент – экземпляр соединения. Он же сокет:

```
var net = require('net');
var server = net.createServer( function(socket) {
    console.log('Соединение с
                '+socket.remoteAddress+":"+socket.remotePort);
}).listen(8080);
console.log('listening on port 8080');
```

Запускаем этот сценарий:

```
$ node tcp2.js
listening on port 8080
```

И «стучимся» браузером по адресу http://localhost:8080. В самом браузере, естественно, ничего не отобразится, зато в консоли появится запись, подобная этой:

```
Соединение с 127.0.0.1:63948
```

Тут мы видим IP-адрес, с которого поступил запрос, и номер созданного TCP-сокета.

Да, кстати, о сокетах. А что это, собственно, вообще такое? Если у вас такого вопроса не возникает, с чистой совестью пропускайте следующую пару абзацев, просто мне слишком часто приходилось

встречать некоторое недопонимание этого термина, даже среди веб-программистов, поэтому хотелось бы пояснить, что я имею в виду. В общем случае сокет – это абстрактный объект, представляющий собой программный интерфейс для обеспечения обмена данными между процессами, вообще, любыми программными процессами. Попросту, сокет – это место встречи, пересечения, обмена данными, о котором договорились два процесса, столкнувшихся с необходимостью взаимодействовать.

Допустим, взаимодействуете вы и некий интернет-магазин, который отправляет вам товар. Он посылает его на ваш «сокет», определяемый номером почтового отделения и вашими личными данными. Вы регулярно этот сокет «опрашиваете», заходя на почту и предъявляя паспорт.

И да, разъемы на корпусе материнской платы к этим сокетам отношения не имеют.

По выполняемым ролям сокеты делятся на клиентские и серверные. Каждый процесс операционной системы может создать слушающий (серверный) сокет и привязать его к какому-нибудь локальному адресу (собственно, пара адресов – адрес компьютера в сети, локальный адрес – и определяют сокет как точку обмена данными). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. Клиентские сокеты используют различные клиентские приложения (например, браузер).

Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будет передавать данные между ним и сервером.

Для протокола TCP в стеке TCP/IP существуют TCP-сокеты, имеющие различные адресные пространства своих локальных адресов – портов. Пара из номера порта и адреса компьютера, то есть IP-адреса, определяет TCP-сокет. TCP-сокеты – это не единственный вид такого рода объектов. С начала 80-х известны BSD-сокеты (сокеты Беркли) – POSIX-стандарт для межпроцессорного взаимодействия (IPC), их воплощение – Unix-сокеты (Unix domain socket), схожие с интернет-сокетами, но не использующие сетевого протокола для обмена данными. Даже в семействе TCP/IP существуют еще UDP-сокеты, причем порт TCP будет указывать совсем не тот локальный адрес, что и порт UDP с тем же номером.

Созданный нами серверный TCP-сокет будет жить, пока мы его явным образом не уничтожим, при этом браузер будет вести себя так, как будто он непрерывно загружает страницу. Собственно, браузеру

и не положено напрямую общаться с TCP-сокетами, но кое-что для него мы сделать можем. Чуть дополним код сервера:

```
    console.log('Соединение с '+socket.remoteAddress+":"+socket.remotePort);
    socket.write('Hello TCP!');
    socket.end();
}).listen(8080);
```

Теперь при установке соединения мы сначала выведем клиенту текст «Hello TCP!». Обратите внимание, мы записываем это сообщение не в стандартный поток вывода, как уже делали при реализации HTTP-сервера, а непосредственно в созданный нами сокет.

Следующей строкой (**socket.end()**) мы закрываем сокет; если бы мы этого не сделали, то браузер продолжил бы чтение из сокета, и переданное сообщение не задержалось бы на экране. Сейчас же все в порядке (рис. 10).

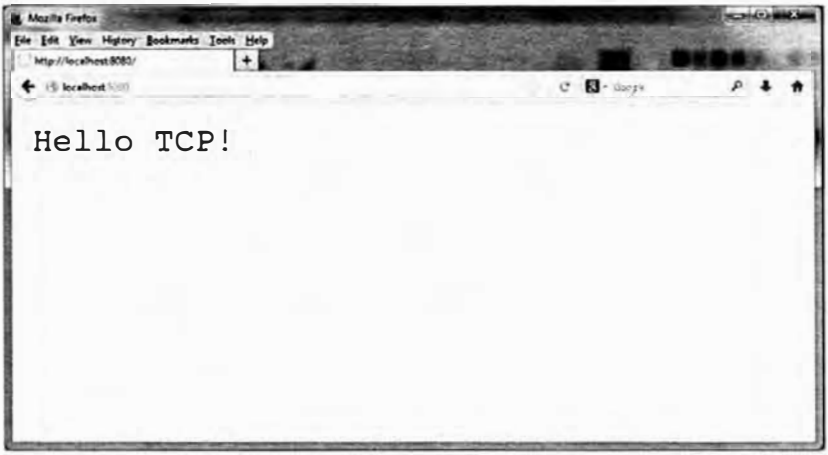

Рис. 10 ❖ TCP-сервер отвечает браузеру

Продемонстрировать непрерывную работу сокета (и завесить браузер) можно следующим деструктивным кодом:

```
socket.write('Hello TCP!');
var i=0;
while(socket){
  i++;
  var m = i+'';
  socket.write(m);
}
socket.end();
```

Результат работы этого безобразия – на рис. 11.

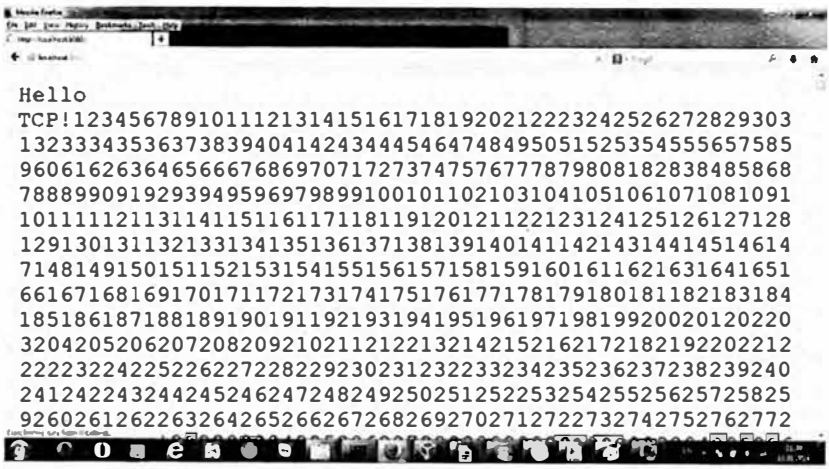

Рис. 11 ❖ Долго браузер так не протянет...

Но оставим деструктив и будем созидать. Заставим наш сокет слушать пару событий:

```
socket.write('Hello TCP!');
    //socket.end();
    socket.on('data', function (data) {
       console.log(data.toString());
       socket.write("Resived: "+data);
       socket.end();

    });
     socket.on('close', function () {
        console.log("Closed");
    });
}).listen(8080);
console.log('listening on port 8080');
```

Если сейчас мы наберем в адресной строке браузера что-нибудь вроде

```
http://localhost:8080/?test,
```

то в консоли мы получим следующий результат:

```
Соединение с 127.0.0.1:55986
GET /?test HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:25.0) Gecko/20100101 Firefox
```

```
25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive


Closed
```

Браузер при этом получит аналогичную информацию.

Впрочем, оставим браузер в покое, полноценный обмен данными с TCP-сервером с помощью него наладить трудно, да и не нужно. Есть более подходящие инструменты – например, утилита nc (netcat), присутствующая практически в любом linux-дистрибутиве, специально предназначенная для подобных тестов (для ОС-семейства [это сокращение от «операционной системы»] Windows netcat тоже существует – http://www.joncraton.org/blog/netcat-for-windows). Немножко дополним наш код:

```
    socket.write('Hello TCP!\n');
  // socket.end();
    socket.on('data', function (data) {
        if(data == 'exit\n') {
            socket.end();
        } else {
            console.log(data.toString());
            socket.write("Resived: "+data);
        }
   });
    socket.on('close', function () {
       console.log("Closed");
   });
}).listen(8080);
console.log('listening on port 8080');
```

И «общаемся» с сервером посредством nc, запущенного в другой консоли:

```
$ nc
Cmd line: localhost 8080
Hello TCP!
123
Resived: 123
test
Resived: test
exit
```

Уже неплохо. Впрочем, полноценного клиента TCP-сервера мы вполне способны написать и сами. Приступим.

```
var net = require('net');
var clientSocket = new net.Socket();
clientSocket.setEncoding('utf8');

clientSocket.connect ('8080','localhost', function () {
    console.log('connected to server');
    clientSocket.write('Hello');
});

clientSocket.on('data',function(data) {
    console.log(data);
});

clientSocket.on('close',function() {
console.log('Соединение закрыто');
});
```

Здесь мы создаем клиентский сокет, задаем кодировку передаваемых данных и устанавливаем соединение с TCP-сервером. Выбор кодировки, разумеется, ограничен протоколом, не следует надеяться на cp1251, только 'utf8', 'ascii' или 'base64'. Далее мы связываем с событиями сокета (получение данных и закрытие соединения) функции обратного вызова, и собственно все – наш клиент готов. Запустим его в отдельной консоли:

```
$ node client.js
connected to server
Hello TCP!

Resived: Hello
```

Таким образом, мы получили обмен данными с сервером. Последний отреагирует на обращение:

```
Соединение с 127.0.0.1:56834
Hello
```

Тут я хочу обратить внимание на одну небольшую деталь в коде TCP-сервера. При логгировании полученных данных мы явным образом привели их значение к строковому виду. Зачем? Да вот как раз для этого случая! Дело в том, что данные, которыми сейчас обмениваются сокеты, представлены отнюдь не в текстовом формате. В этом нетрудно убедиться, убрав приведение типов:

```
} else {
    console.log(data.toString());
    socket.write("Resived: "+data);
}
```

и посмотрев на результат:

```
Соединение с 127.0.0.1:62456
<Buffer 48 65 6c 6c 6f>
```

Да, это буфер.

Но вернемся к клиенту. Создадим интерфейс для ввода данных с клавиатуры:

```
clientSocket.connect ('8080','localhost', function () {
    console.log('connected to server');
    clientSocket.write('Hello',function(){
         process.stdin.resume();
            process.stdin.on('data', function (data) {
                clientSocket.write(data);
        });
    ));
});
```

Теперь взаимодействие будет более осмысленным:

```
$ node client.js
connected to server
>
Hello TCP!
Resived: Hello
test
Resived: test

hi
Resived: hi

exit
Соединение закрыто
```

Ну а для того, чтобы наш скрипт прочувствовал себя полновесным сервером, давайте попробуем подключить к нему нескольких клиентов. Для этого запустим несколько экземпляров нашего клиентского скрипта и будем пристально всматриваться в консоль.

Итак, приступим – запустим клиентов:

```
$ node client.js
connected to server
Hello TCP!

Resived: Hello
test
Resived: test

$ node client.js
```

```
connected to server
Hello TCP!

Resived: Hello
another test
Resived: another test
```

Консоль сервера будет выглядеть так:

```
node tcp1.js
listening on port 8081
Соединение с 127.0.0.1:60270
Hello
test

Соединение с 127.0.0.1:60272
Hello
another test
```

Чуть-чуть изменив код сервера, мы можем даже организовать нечто вроде TCP-чата:

```
var net = require('net');
var clients =[];
var tcpServer = net.createServer( function(socket) {
     clients[clients.length++] = socket;
    console.log('Соединение с '+socket.remoteAddress+":"+socket.remotePort);
     clients.forEach(
               function(client){
                        client.write("Here: "+socket.remoteAddress+":"+socket.
remotePort);
               }
          )
     socket.on('data', function (data) {
        console.log(data.toString());
          clients.forEach(
               function(client){
                    client.write("Resived: "+data+" from: "+socket.
remoteAddress+":"+socket.remotePort);
               }
          );
    });
}).listen(8081);
console.log('listening on port 8081');
```

Тут мы создаем массив подключаемых сокетов-клиентов, который сохраняется все время жизни сервера, каждый новый сокет, создаваемый подключением очередного клиента. Соответственно, теперь все поступающие сообщения передаются каждому из подключаемых участников. Вот как выглядит безбашенный TCP-чат со стороны консоли одного из клиентов:

```
C:\Users\Geol\node\tcp>node client.js
connected to server
Here: 127.0.0.1:60451Resived: Hello from: 127.0.0.1:60451
Here: 127.0.0.1:60452
Resived: Hello from: 127.0.0.1:60452
Hi!
Resived: Hi!
 from: 127.0.0.1:60451
Resived: Hello! How are You?
 from: 127.0.0.1:60452
Cool
Resived: Cool
 from: 127.0.0.1:60451
Here: 127.0.0.1:60467
Resived: Hello from: 127.0.0.1:60467
Resived: Hi sockets!
 from: 127.0.0.1:60467
```

(Полная картина – на рис. 12.)

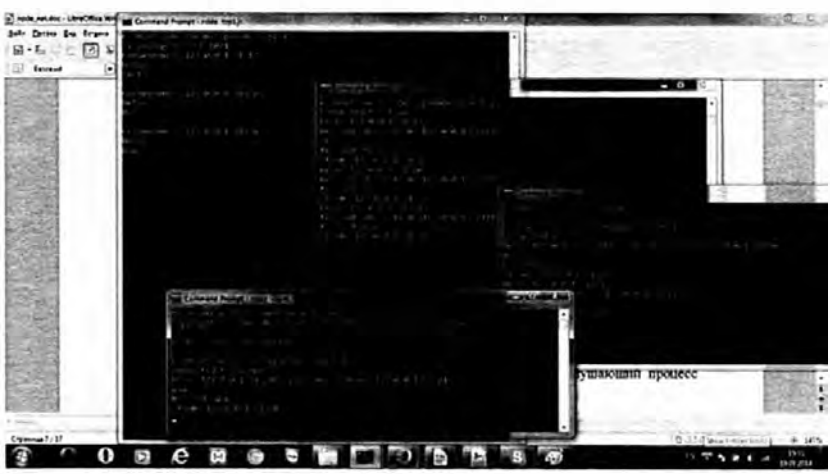

Рис. 12 ❖ Безбашенный TCP-чат

Что еще? Теперь можно приспособить нашего клиента для совершения каких-нибудь осмысленных действий. Например, управлением http-сервером. Внедрим в наш TCP-сервер http:

```
var net = require('net');
var http = require('http');
var httpServer;

function httpStart(){
```

```
    server = http.createServer(function (request, response) {;
       response.writeHead(200, {'Content-Type': 'text/html'});
       response.end('<h1>Hello Node!</h1>');
       ........................................
   }).listen(8080);

   return server;
}
var tcpServer = net.createServer( function(socket) {

     socket.on('data', function (data) {
        if(data.toString() == 'start\r\n'){
            httpServer = httpStart();
            socket.write("Server started");
         }
         if(data.toString() == 'stop\r\n') {
              httpStop(httpServer);
               socket.write("Server stoped");
         }
    });
     socket.on('close', function () {
        console.log("Closed");
    });
}).listen(8081);
```

Тут мы в зависимости от поступающих от клиента текстовых сообщений запускаем или останавливаем http-сервер. Со стороны клиента это будет выглядеть так:

```
start
Server started
stop
Server stoped
start
Server started
stop
Server stoped
```

UDP – тоже полезный протокол

Я знаю отличную шутку про UDP, но не уверен, что она до вас дойдет.
tcp/ip-шутка

Отличительная особенность протокола UDP (User Datagram Protocol – протокол пользовательских дата-грамм) состоит в том, что он не требует для своей работы выделенного соединения.

Используя UDP, мы можем просто отослать пакет по определенному IP-адресу и порту, и он будет передаваться от компьюте-

ра к компьютеру, пока не достигнет цели (или не потеряется). Это, в частности, означает отсутствие гарантированной доставки пакетов, меньшую, по сравнению с TCP, надежность. Однако надежность (как ни странно) – это не всегда ключевое требование к сетевому взаимодействию. Иногда большее значение имеют такие свойства, как легкость и скорость, позволяющая обслуживать взаимодействия реального времени. Например, в IP-телефонии надежность доставки – это нехорошо, но не критично. А вот её скорость... Словом, UDP в современной IT-инфраструктуре есть где применить.

UDP использует свой тип сокетов, и, что приятно, Node.js (а точнее, модуль **dghram**, входящий в ядро) его поддерживает.

Продемонстрируем работу протокола, создав простой сервер, принимающий UDP-пакеты:

```
var dgram = require('dgram');
var udpServer = dgram.createSocket("udp4");
udpServer.bind(8082);
```

Как видите, это обычный UDP-сокет, привязанный к определенному порту. Второй строчкой кода мы задаем тип udp-сокета (udp4 или udp6, в зависимости от версии протокола). Раз это сокет, то он просто обязан иметь событие **onmessage**, на которое мы «повесим» вывод информации о возможных поступающих данных в консоль:

```
server.on ("message", function(msg, info) {
    console.log("Message: " + msg + " from " + info.address + ":"
+ info.port);
});
```

Обратите внимание, с объектом соединения мы не работаем, его просто нет, как, собственно, и было обещано.Клиент будет ненамного сложнее:

```
var dgram = require('dgram');
var client = dgram.createSocket("udp4");
process.stdin.resume();
process.stdin.on('data', function (data) {
    console.log(data.toString('utf8'));
    client.send(data, 0, data.length, 8082, "localhost",
    function (err, bytes) {
        if (err)
            console.log('error: ' + err);
        else
            console.log('OK');
   fp});
});
```

Обратите внимание: проверяется только успех или неуспех отправки данных, получение отследить не представляется возможным. Зато тут же мы можем продемонстрировать преимущество протокола. Можно сколько угодно останавливать и запускать сервер – клиент останется в рабочем состоянии, и в моменты работы сервера данные будут доставлены.

Переходим на прикладной уровень – реализация HTTP

Основное назначение Node – работа в www, соответственно, реализация протокола HTTP (HyperText Transfer Protocol – протокол передачи гипертекста) здесь очень важна. Роль HTTP-сервера является главной для этой платформы.

Впрочем, с реализации HTTP-сервера мы и начали свое знакомство с Node.js. Теперь разберем его работу более подробно. Итак, сервер:

```
var http = require('http');
var server = http.createServer().listen(8080);
server.on('request', function(request, response){
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.end('<h1>Hello HTTP!</h1>');
    });
```

Запустив это приложение, мы можем открыть в браузере url http://localhost:8080/ и получить страничку, изображенную на рис. 13.

Рис. 13 ❖ Теперь HTTP

Подключаемый модуль **http** – это модуль ядра Node.js, он основан на модуле **net**, что делает доступным для приложения все аспекты сетевого взаимодействия. Объект, возвращаемый методом **createServer()**, реализует интерфейсы объекта **EventEmiter**. Объекту принадлежат следующие события:

- **connect** – возникает при установке http-соединения (точнее, при запросе клиентом HTTP метода CONNECT);
- **request** – возникает при каждом запросе к http-серверу со стороны клиента (обычно браузера);
- **upgrade** – возникает при получении заголовка upgrade от клиента (HTTP/2.0);
- **clientError** – возникает при ошибке клиентского соединения;
- **close** – возникает при закрытии соединения.

Основными методами объекта сервер являются **listen()** и **close()**, назначение которых очевидно.

В своем небольшом примере мы использовали единственный метод «request», то есть запрос со стороны клиента. Про объекты, принимаемые функцией обратного вызова как аргументы (request и response), мы уже немного говорили. Первый является воплощением в коде самого запроса и несет всю возможную информацию о нем. На самом деле это объект **http.IncomingMessage**. Второй дает доступ к свойствам ответа сервера. Второй – это объект **http.ServerResponse**. Именно в него мы пишем то, что собираемся передать клиенту, включая заголовки HTTP-ответа.

Давайте разберем эти объекты немного подробнее.

IncomingMessage – входящий HTTP-запрос

Этот объект, как уже упоминалось, несет информацию о входящем запросе. И эту информацию мы можем прочитать. Например, в консоль:

```
server.on('request', function(request, response){
        console.log(request.method);
        console.log(request.url);
        console.log(request.httpVersion);
}
```

При наборе в браузере url http://localhost:8080/main/ в консоль будет выведена примерно такая информация:

```
$ node server.js
GET
/main/
1.1
```

Следующим образом мы можем просмотреть заголовки запроса:

```
server.on('request', function(request, response){
    for(var header in request.headers) {
        console.log(header+": "+request.headers[header]);
    }
}
```

Результат:

```
host: localhost:8080
user-agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:26.0) Gecko/20100101 Firefox/
26.0
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language: en-US,en;q=0.5
accept-encoding: gzip, deflate
connection: keep-alive
cache-control: max-age=0
```

Еще один объект, к которому можно получить доступ через IncomingMessage, – TCP-сокет соединения:

```
server.on('request', function(request, response){
    console.log(request.socket.localPort);
    console.log(request.socket.writable);
    console.log(request.socket.domain);
    console.log(request.socket.address);
});
```

Результат:

```
8080
true
null
[Function]
```

ServerResponse

Объект **http.ServerResponse** создается уже на сервере, вернее, самим сервером. Соответственно, ему доступны все параметры, которые мы считывали в предыдущей части этой главы. В нашем примере мы использовали два метода этого объекта: **write()** – для отправки тела ответа на запрос и **end()** – для завершения сеанса ответа. На этом возможности данного объекта не заканчиваются. Он отвечает за полноценную реализацию HTTP-протокола в части ответа сервера, и с помощью него можно, например, задавать HTTP-заголовки:

```
var body = '<h1>hello HTTP</h1>';
response.writeHead(200, {
    'Content-Length': body.length,
```

```
    'Content-Type': 'text/plain',
    'My-Header': 'Kill All Humans!'
});
```

Результат можно посмотреть в браузере, например средствами Firebug (рис. 14).

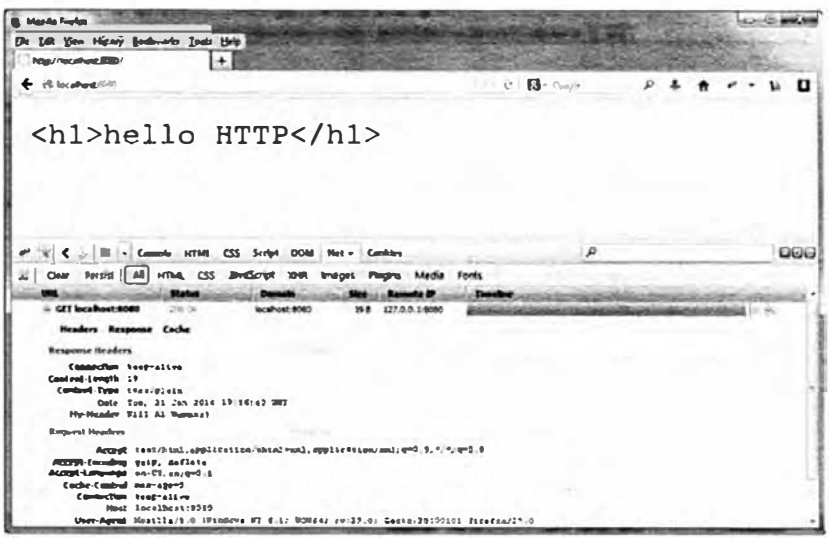

Рис. 14 ❖ Шлем свои HTTP-заголовки

Ну или так:

```
if(url.parse.host != myhost){
    response.writeHead(400});
}
```

В этом случае браузер получит стандартный «Page Not Found» ответ.

Заголовок можно убрать командой **removeHeader()**:

```
response.removeHeader("Content-Encoding");
```

или перезаписать посредством **response.setHeader()**:

```
response.setHeader("Content-Type", "text/html");
```

Получить уже заданные заголовки можно так:

```
console.log(response.getHeader('content-type'));
```

Надо только помнить, что любые заголовки, устанавливаемые командой **response.writeHead()**, отправятся в браузер не немедленно, а по завершении формирования ответа, которое производится командой **response.end()**:

```
response.end('<html>...</html>', "utf8");
```

Впрочем, если есть необходимость отправить заголовок (или что-нибудь еще) клиенту немедленно, для этого есть метод **response.write()**:

```
response.write("alarm", "utf8");
```

Данные в этом случае отправятся клиенту (например, браузеру) сразу. Правда, сеанс передачи данных не совершится – браузер будет ждать окончания ответа. Для его завершения все же потребуется **response.end()**.

Статус ответа можно задавать непосредственно, помимо заголовка, методом **response.statusCode()**:

```
response.statusCode = 400;
```

При этом к этому свойству можно обращаться из приложения, уже после отправки данных клиенту.

На этом с сервером закончим. Обратимся теперь к другому объекту модуля http, позволяющего нам буквально из нескольких строчек кода создать полноценного HTTP-клиента.

HTTP-клиент (грабим Центробанк)

Стоп! А зачем? Нет, правда, зачем создавать HTTP-клиента, если HTTP-клиент – это браузер? Все просто – иногда необходимо сделать запрос по этому протоколу непосредственно из нашего приложения. Примеров можно привести много – запрос курсов валют, биржевых котировок от веб-сервисов, «общающихся» по этому протоколу, проведение онлайн-платежей, взаимодействие с платежными системами и тому подобные вполне распространенные «кейсы».

При реализации подобного взаимодействия такие языки, как Си или PHP, используют библиотеку с URL (libcurl). Для Node.js, кстати, тоже есть соответствующий модуль, но обычно особой необходимости в нем нет – модуль **http** позволяет легко создать клиента, пользуясь родными средствами JavaScript. Вот так:

```
var http = require('http');
var param = {
  hostname: 'ya.ru',
```

```
  port: 80,
  method: 'POST'
};
var req = http.request(param, function(res) {
     console.log('STATUS: ' + res.statusCode);
     res.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
     });
});
req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});
req.end();
```

Что тут происходит? Ну, во-первых, мы создали экземпляр объекта http.request, в конструкторе которого передали объект, содержащий набор параметров сокета будущего HTTP-запроса. Аргументом функции обратного вызова будет результат этого запроса, объект, реализующий событийный механизм **EventEmiter**. В данном случае выбирать событие не приходится – это будет поступление данных (ответа на HTTP-запрос) со стороны сервера. Эти данные мы и выводим в консоль. Единственное событие, которое мы «навешиваем» на сам запрос, – это поручение ошибки, его обработка пояснений не требует.

Метод **req.end()** обязателен. Не забывайте, что мы создаем HTTP-запрос, и именно он его завершает. Без этого ответа от сервера ждать не стоит.

Такое обращение к сайту yandex.ru даст следующий, не очень желаемый, но вполне понятный результат:

```
$ node client.js
STATUS: 411
BODY: <html>
<head><title>411 Length Required</title></head>
<body bgcolor="white">
<center><h1>411 Length Required</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Это неудивительно, мы использовали метод POST, в общем-то не предназначенный просто для просмотра (сервер недоволен тем, что не указан обязательный в этом случае заголовок 'Content-Length') . Поменяв его на GET, мы сможем получить запрашиваемую страничку:

```
  port: 80,
  method: 'GET'
};
```

Весь ответ я публиковать не буду:

```
$ node client.js
STATUS: 200
BODY: <!DOCTYPE html><html><head><title>Яндекс</title><link rel="shortcut icon"
ref="//yandex.st/lego/_/pDu9OWAQKB0s2J9IojKpiS_Eho.ico"><link rel=search href="
/
.....................................................
```

Это html-страница, а мы совсем не собираемся подменять браузер. Давайте попробуем сотворить что-нибудь полезное. Допустим, добыть курс валюты с сервера Центробанка. Но сначала рассмотрим параметры конструктора запроса подробнее. Вот они:

- **host**: домен или IP-адрес для HTTP-сервера;
- **hostname**: параметр устанавливается специально для метода url.parse();
- **port**: порт удалённого HTTP-сервера;
- **localAddress**: локальный интерфейс для сетевых соединений;
- **socketPath**: путь до Unix-сокета (или просто пара host:port);
- **method**: метод по протоколу HTTP. Значения: 'GET' (по умолчанию), 'POST', 'PUT', 'DELETE';
- **path**: путь от корня HTTP-сервера, может включать строку запроса при использовании метода GET. Например: '/index.html?article=**77**';
- **headers**: объект, содержащий список HTTP-заголовков;
- **auth**: данные для базовой HTTP-авторизации;
- **agent**: задаёт свойства объекта Agent. Значения:
  - **undefined (default)**: использовать стандартный Agent для этого хоста и порта;
  - **объекта Agent**: использовать переданный Agent;
  - **false**: создать новый Agent для этого хоста и порта. Текущий Agent не будет больше использоваться.

Класс **Agent** мы сейчас использовать не будем, но имейте его в виду, на случай если вам, например, потребуется удобно оперировать с пулом запросов. А в остальном мы во всеоружии. Приступим:

```
var http = require('http');
var param = {
  hostname: 'www.cbr.ru',
  path: '/scripts/XML_daily.asp?date_req:01.11.2013',
  port: 80,
  method: 'GET'
};

var req = http.request(param, function(res) {
```

```
    console.log('STATUS: ' + res.statusCode);
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
       console.log('BODY: ' + chunk);
    });
});
req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});
 console.log(get_data);
req.end();
```

Все, что мы сделали, – это задали параметр path с строкой запроса (скрипт на сервере ЦБ принимает параметр date_req, дату, на которую запрашивается курс). Результат будет следующим:

```
$ node client.js
date_req=01.11.2013
STATUS: 200
BODY: <?xml version="1.0" encoding="windows-1251" ?>
<ValCurs Date="01.11.2013" name="Foreign Currency Market">
<Valute ID="R01010">
        <NumCode>036</NumCode>
        <CharCode>AUD</CharCode>
        <Nominal>1</Nominal>
        <Name>Австралийский доллар</Name>
        <Value>30,4399</Value>
</Valute>
<Valute ID="R01020A">
        <NumCode>944</NumCode>
        <CharCode>AZN</CharCode>
        <Nominal>1</Nominal>
        <Name>Азербайджанская мана</Name>
        <Value>40,9443</Value>
</Valute>
<Valute ID="R01035">
        <NumCode>826</NumCode>
        <CharCode>GBP</CharCode>
        <Nominal>1</Nominal>
        <Name>Фунт стерлингов Соединенного Королевства</Name>
        <Value>51,3854</Value>
</Valute>
<Valute ID="R01060">
.........................................
```

Все получилось. А как отправить POST-запрос и получить ответ? Пусть для данного случая это неактуально, но куча сервисов требует именно такого типа запроса. Это ненамного сложнее и позволит нам продемонстрировать еще один метод – **request.write()**:

```
var http = require('http');
var post_data = 'date_req=01.11.2013';
var param = {
  hostname: 'www.cbr.ru',
  path: '/scripts/XML_daily.asp',
  port: 80,
  method: 'POST',
  headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
          'Content-Length': post_data.length
      }
};
var req = http.request(param, function(res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.on('data', function (chunk) {
        res.setEncoding('utf8');
        console.log('BODY: ' + chunk);
    });
});
req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

req.write(post_data);

req.end();
```

Точно так же мы можем отравлять данные для hml-форм, других онлайн-сервисов.

HTTPS – шифруемся!

В наше непростое время трудно обойтись без шифрования трафика. Осуществляете ли вы электронный платеж, просматриваете почту, пользуясь веб-интерфейсом, получаете доступ к различной конфиденциальной информации – во многих случаях использование протокола **HTTPS** (HyperText Transfer Protocol Secure) – хорошее решение проблем безопасной передачи данных.

HTTPS не является отдельным протоколом. Это расширение HTTP, заставляющее его работать через механизмы шифрования SSL и TLS. Он предназначен для защиты от атак, основанных на прослушивании сетевого соединения – применении сценариев, man-in-the-middle и т. д.

Запускаем HTTPS-сервер

Работа с HTTPS в Node.js осуществляется посредством модуля **https**. Он очень похож на http, но для его работы нам потребуются закрытый и открытый ключи и SSL-сертификат. Сгенерировать их можно программой OpenSSL, существующей как для unix-like-систем, так и для операционной системы windows. Процедуру их создания я опущу, она не представляет ничего сложного. Код HTTPS-сервера тоже не очень сложен:

```
var https = require('https');
var fs = require('fs');

var options = {
  key: fs.readFileSync('keys/server.key'),
  cert: fs.readFileSync('keys/server.crt')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("Hello SSL world\n");
}).listen(8000);
```

Как видите, в качестве параметров для команды createServer мы предоставили наши ключ и (самоподписанный) сертификат. Запустим сервер и попробуем зайти по адресу https://localhost:8000. Если все сделано правильно, то в браузере мы увидим картину, подобную изображенной на рис. 15. Это естественно – наш сертификат «самопальный». Принимаем риск, добавляем исключение безопасности (ну, себе-то мы доверяем?) и получаем запрашиваемую страничку по защищенному протоколу.

И секретный клиент

Клиент https-сервера также строится на объекте request (естественно, https.request). Вот пример клиента (файрл **client.js**):

```
var https = require('https');

var options = {
  hostname: 'localhost',
  port: 8000,
  path: '/',
  method: 'GET'
};

var req = https.request(options, function(res) {
```

Рис. 15 ❖ Сертификат не подтвержден. Зато он наш!

```
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(data) {
    process.stdout.write(data);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

Список параметров объекта https.request практически совпадает с опциями http.request.

Если у вас все еще работает https-сервер из предыдущего примера, запустите этот код, открыв другую консоль. Результат должен быть таким:

```
$ node client.js
statusCode:  200
headers:  { connection: 'keep-alive', 'transfer-encoding': 'chunked' }
hello world
```

Ну что же, мы выяснили, что Node.js обладает всеми средствами для работы в среде Интернет, в том числе и в качестве платформы для различных сетевых сервисов. Но то, что мы рассмотрели сегодня, – это лишь основы, на самом деле дальше все гораздо интереснее, и об этом мы обязательно еще поговорим.

WebSockets – стандарт современного веба

Взаимодействие посредством технологии WebSockets уже давно не является чем-то экспериментальным, её используют в браузерных играх, платежных системах, интерактивных интерфейсах. И если вы веб-разработчик и до сих пор не знаете, как их использовать, немедленно читайте эту статью – в нашей профессии отставать от технологий совершенно недопустимо!

Браузер – веб-сервер. Надо что-то менять

Получение реакции браузера на события, происходящие на сервере, всегда представляло собой проблему. HTTP-протокол не критиковал, наверное, только очень ленивый человек, а проблема постоянного соединения с сервером стала настолько привычной головной болью разработчиков, что её практически перестали замечать. Дело в том, что сама ее реализация не предполагала подобного рода взаимодействия – при штатной работе (рис. 16) данные с сервера могли быть доставлены в браузер только в ответ на очередной HTTP-запрос (предполагающий перезагрузку страницы). Между тем жизнь сразу ставила перед веб-разработчиками задачи, требующие этого, – вспомним хотя бы старые добрые html-чаты. Естественно, находились решения разной степени ужасности, имитирующие push-действия сервера. Например, на клиенте организовывался фрейм, перегружающийся раз в секунду и таким образом запрашивающий сервер на предмет изменения состояния.

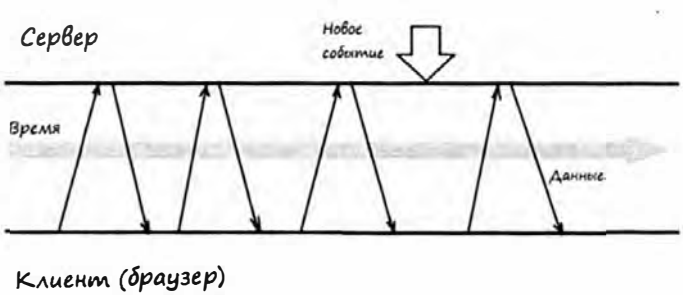

Рис. 16 ❖ Традиционное веб-взаимодействие (Pulling)

Минусов в этом подходе предостаточно – создается просто дикое количество лишних запросов, приходится организовывать клиентскую часть приложения таким образом, чтобы «рычаги управления» в любой момент времени получал этот самый, в общем, служебный фрейм. Но главная проблема – в том, что это лишь эмуляция реакции на серверное событие – браузер получает сведения с неизбежной задержкой, серверу же приходится хранить данные до тех пор, клиентский запрос сподобится его забрать.

С появлением в браузерах объекта XMLHTTPRequest положение немного улучшилось. Теперь появилась возможность выстраивать взаимодействие с сервером по схеме Long Polling (описанную ранее схему с опрашивающим фреймом принято называть просто Polling). Суть этого «длинного вытягивания» в следующем (рис. 17):

- клиент отправляет запрос на сервер;
- соединение не закрывается, клиент ожидает наступления события;
- событие происходит, и клиент получает ответ на свой «long»-запрос;
- клиент тут же отправляет новый запрос.

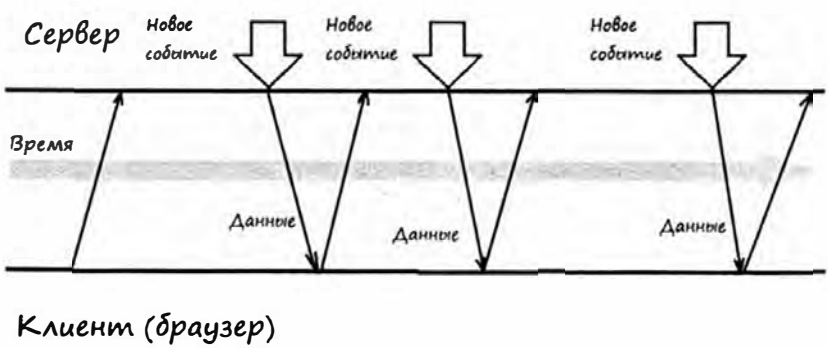

Рис. 17 ❖ Long Pulling

Воплощается, естественно, это асинхронным запросом серверу и сопоставлением удачному ответу функции обратного вызова.

Минусов в такой организации взаимодействия меньше, они связаны в основном со сложностью воплощения. Но главный, принципиальный недостаток так и не преодолен (хоть и влияние этого обстоятельства сведено до минимума) – сервер и серверные события здесь все еще не являются инициатором взаимодействия. С этим

можно мириться, но, впрочем, это еще не повод не ждать чего-то лучшего. Не так давно появившаяся технология WebSockets [1] представляет собой реализацию протокола полнодуплексной связи поверх TCP-соединения.

WebSockets – окончательное решение?

Что такого особенного может предоставить эта технология? Она дает полное переосмысление привычного в мире WWW-взаимодействия. Про диктуемую HTTP-модель «запрос/ответ на запрос» можно забыть. В рамках протокола WebSockets браузер и сервер превращаются в полноправных участников взаимодействия (в противовес прежней клиент-серверной модели) и, соответственно, могут принимать и посылать сообщения в тот момент, когда им это заблагорассудится! Взаимодействие становится полностью асинхронным и симметричным.

WebSocket – это протокол двунаправленного обмена данными, который характеризует полностью дуплексный характер взаимодействия. На практике это означает следующее. WebSockets устанавливает одно, причем единственное, соединение клиента с сервером. После необходимых проверок, подтверждающих, что сервер может работать с WebSocket, сервер и клиент могут отправлять через него текстовые сообщения, причем передача происходит сразу же, при отсылке – WebSockets создает двунаправленные каналы связи. Соединение постоянно держится открытым, что позволяет не передавать лишних HTTP-заголовков. При этом в веб-сокетах нет ограничений не на количество соединений, не на очередность запросов.

Взаимодействие по протоколу WebSocket на данный момент выглядит следующим образом.

Заголовок запроса браузера на установку соединения:

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==
Upgrade: websocket
Sec-WebSocket-Version: 13
```

Заголовок ответа сервера:

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2012 17:38:18 GMT
Connection: Upgrade
```

```
Server: Kaazing Gateway
Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org
Access-Control-Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type
```

В примерах мы видим работу первого и самого главного этапа работы по протоколу WebSocket, так называемое «рукопожатие» (Handshake), после успешного завершения которого устанавливается WebSocket-соединение.

На этом этапе только минимальные пояснения: заголовок Sec-WebSocket-Accept формируется из строковых значений заголовка Sec-WebSocket-Key и специальной строки с использованием SHA-1 и Base64. По его значению и определяется, согласен ли сервер общаться по протоколу WebSockets.

Подробнее поля заголовков мы сейчас рассматривать не будем, в случае необходимости с ними можно ознакомиться в The WebSocket Protocol, который в декабре 2011 г. обрел статус RFC (RFC 6455) [2].

Простой способ – модуль ws

Для того чтобы начать работать с веб-сокетами, нужны всего две вещи – браузер, поддерживающий WebSocket, и сервер, реализующий эту технологию. На стороне браузера все просто – WebSockets API входит в семейство JavaScript-интерфейсов, условно объединенных под названием HTML5, и поддерживается большинством современных версий браузеров. На серверах все не так благополучно – традиционные Apache, Nginx, IIS про WebSockets ничего не знают (либо узнали совсем недавно), а специализированные решения, как правило, плохо справляются с обязанностями HTTP-сервера общего назначения. И тут у Node есть колоссальное преимущество, поскольку разработка технологии велась в то время, когда идея WebSockets уже вовсю воплощалась в жизнь. Можно сказать, что серверная составляющая WebSockets присутствует в node «из коробки». Ну, почти так – все, что нужно сделать, – доставить соответствующий модуль:

```
npm install ws
```

Начинаем работу с ws

Сам WebSocket-сервер теперь можно написать так же легко, как и HTTP. Сначала подключим необходимый пакет:

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

И запустим WebSocket-сервер:

```
var webSocketServer = new WebSocketServer.Server({port: 8080});
```

Собственно, это все – код можно запускать, и минимальный и пока бесполезный WebSocket-сервер будет работать, ожидая запросов по протоколу ws:// на заданный порт (в нашем случае 8080).

Теперь попробуем заставить WebSocket работать не впустую. Сначала напишем функцию обратного вызова на событие соединения:

```
webSocketServer.on('connection',function(ws){
    console.log("Новое соединение");
})
```

Затем напишем небольшого клиента – веб-страницу, размещенную на обычном веб-сервере:

```
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script>
    onload = function(){
   // создаем соединение
   var ws = new WebSocket("ws://localhost:8080");
 }
</script>
<head>
    <body>
    </body>
</html>
```

Теперь при открытии страницы в браузере будет создан WebSocket, и, соответственно, на консоли появится надпись «Новое соединение» (рис. 18).

Это прекрасно, но пока полнодуплексного соединения, мягко говоря, не наблюдается. Что естественно – взаимодействия по направлению от сервера к клиенту у нас пока не происходит. Исправим это недоразумение, заодно учтя ту естественную возможность, что клиент, скорее всего, будет в одном экземпляре:

```
var WebSocketServer = new require('ws');
var wss = new WebSocketServer.Server({port: 8080});
var clients = [];
wss.on('connection', function(ws) {
    var id = clients.length;
     clients[id] = ws;
    console.log("Новое соединение № "+id);
     clients[id].send("Приветствуем! ваш идентификатор "+id);
```

Рис. 18 ❖ Socket Server работает!

```
    console.log(clients);
    for (var key in clients)  {
      if(key!=id){
         clients[key].send("К нам присоединился № "+id);
      }
    }
})
```

На клиенте напишем код для приема сообщений:

```
onload = function(){
   // создаем соединение
   var ws = new WebSocket("ws://localhost:8080");
   ws.onmessage  = function(event){
       alert(event.data);
   }
}
```

Теперь попробуем открыть страницу в нескольких окнах браузера (лучше в разных браузерах, еще лучше – на разных устройствах). Картина должна быть такая, как на рис. 19.

Как видите, двустороннее взаимодействие налицо, и... похоже, мы пишем чат. Что же, не будем останавливаться.

Реализация WebSocket-чата

Следует заметить, что на клиентской стороне тут используются WebSockets API и PostMessages API – JavaScript-интерфейсы, «традиционно» причисляемые к группе технологий HTML5. Прямое следствие этого – то, что при реализации подобного клиента мы можем столкнуться с отсутствием поддержки новых технологий некоторыми старыми браузерами. Впрочем, не все так страшно – большинство современных обозревателей с этими задачами справляется. Пользуясь именно этими возможностями, попробуем органи-

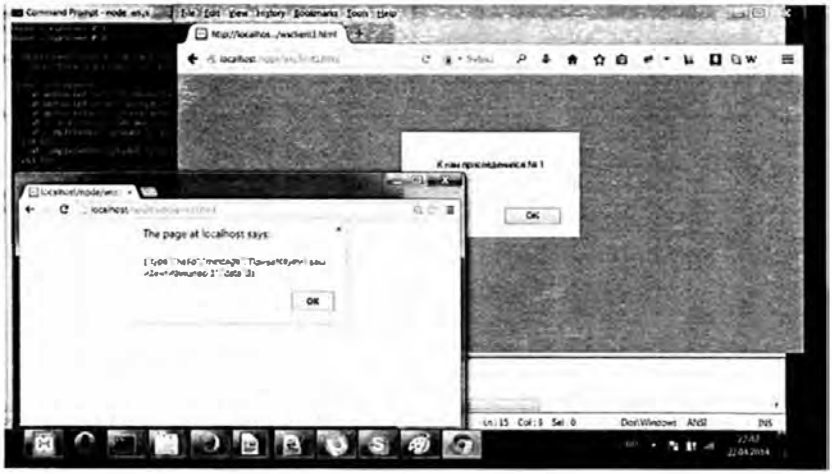

Рис. 19 ❖ Обслуживание нескольких соединений

зовать доставку текстовых сообщений от клиента (браузера) на наш WebSocket-сервер. Для этого сначала сделаем простую форму для отправки сообщений:

```
<form name="push">
  <input type="hidden" name="userId"/>
  <input type="text" name="message"/>
  <input type="submit" value="Отправить"/>
</form>
```

И javaScript-обработчик:

```
ws.onmessage  = function(event){
    alert(event.data);
}
document.forms.push.onsubmit = function() {
   ws.send(this.message.value);
   return false;
};
}
```

Теперь озаботимся приемом сообщений на сервере:

```
wss.on('connection', function(ws) {
    var id = clients.length;
     clients[id] = ws;
    console.log("Новое соединение № "+id);
     clients[id].send("Приветствуем! ваш идентификатор "+id);
     for (var key in clients)  {
```

```
        if(key!=id){
           clients[key].send("К нам присоединился № "+id);
        }
    }
    ws.on('message', function(message) {
        console.log('получено сообщение ' + message);
         for (var key in clients)  {
             if(key!=id){
                        clients[key].send(message);
            }
         }
    })
})
```

Собственно, все – обмениваться сообщениями в реальном времени уже можно. Но мы не какие-нибудь гики, а серьезные веб-разработчики и доведем наш чат до пристойного состояния. То есть напишем более комфортный интерфейс взаимодействия. Начнем с сервера:

```
var WebSocketServer = new require('ws');
var wss = new WebSocketServer.Server({port: 8080});
var clients = [];wss.on('connection', function(ws) {
    var id = clients.length;
     clients[id] = ws;
    console.log("Новое соединение № "+id);
     clients[id].send(JSON.stringify( { type: 'hello', message: "Приветствуем!
ваш идентификатор "+id, data: id} ));
     for (var key in clients)  {
       if(key!=id){
           console.log("send");
          clients[key].send(JSON.stringify({type: 'info', message:"К нам присо-
единился № "+id}));
       }
    }
     ws.on('message', function(message) {
        console.log('получено сообщение ' + message);
         for (var key in clients)  {
              clients[key].send(JSON.stringify( { type: 'message', message:
message, author: id} ));

         }
    })
})
```

Тут мы создаем что-то, подобное протоколу взаимодействия. Следуя его реализации, теперь мы обменивается с клиентом не текстовыми фрагментами, а сообщениями в формате JSON. Метод **JSON.stringify** преобразует объект JavaScript в сериализованную строку – в чистом виде JavaScript объект через сокет передать не получится.

Теперь усовершенствуем код клиента:

```
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script>
onload = function(){
   var ws = new WebSocket("ws://localhost:8080");

   document.forms.push.onsubmit = function() {
      ws.send(this.message.value);
      return false;
    };
   ws.onmessage = function(event) {
      var message = JSON.parse(event.data);
       var text = "";
       switch (message.type) {
          case "info": {
              text = message.message
              break;
          }
          case "message": {
              text = message.author+": "+message.message;
              break;
          }
          default: {
              alert(message.message);
              break;
          }
       }
      var messageElem = document.createElement('div');
      messageElem.appendChild(document.createTextNode(text));
      document.getElementById('subscribe').appendChild(messageElem);
     };
 }
</script>
<head>
    <body>
<form name="push">
  <input type="hidden" name="userId"/>
  <input type="text" name="message"/>
  <input type="submit" value="Отправить"/>
</form>
<div id="subscribe">

</div>
    </body>
</html>
```

Обмен бинарными данными

Как уже говорилось, через WebSocket можно передать сообщение в текстовом формате. Но стандарт этим не ограничивается – можно также передавать бинарные данные, попробуем дополнить функционал нашего чата возможностью обмениваться картинками. Правда, задача эта не очень простая – сначала надо дополнить функционал интерфейса на стороне клиента. Тут нам на помощь придут новые возможности стандарта HTML5:

```
onload= function(){
.....................................................................
var target = document.getElementById("target");
target.ondrop = function(e) {
    target.style.backgroundColor = "#fff";
    try {
        e.preventDefault();
        handleFileDrop(e.dataTransfer.files[0]);
        return false;
    } catch(err) {
        console.log(err);
    }
}
target.ondragover = function(e) {
    e.preventDefault();
    target.style.backgroundColor = "#6fff41";
}
target.ondragleave = function() {
     target.style.backgroundColor = "#fff";
}
function handleFileDrop(file) {
    var reader = new FileReader();
    reader.readAsArrayBuffer(file);
    reader.onload = function() {
        ws.send(reader.result);
    }
}
}
</script>
<h1>Переместите картинку на прямоугольник</h1>
<div style="width:400px;height:200px;border: 1px solid red;" id = 'target'>
</div>
```

Так мы создаем интерфейс для загрузки картинок, перетаскиванием их на веб-страницу (рис. 20). Здесь используются такие технологии, как FileAPI и HTML5 Drag'n'Drop. Их мы описывать не будем, обратите внимание только на передачу файла на сервер:

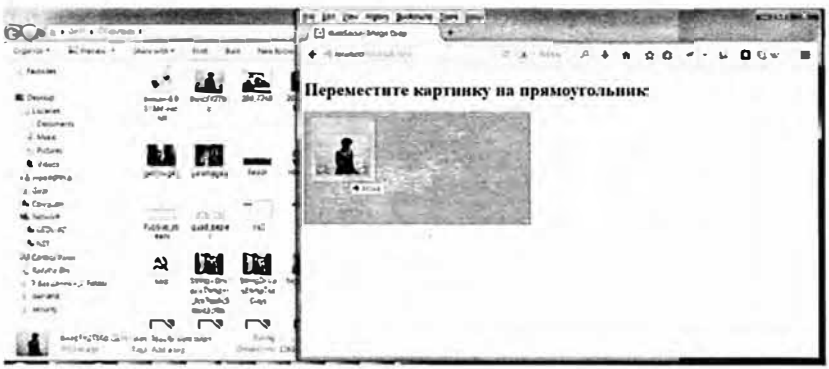

Рис. 20 ❖ Посылаем симпатичные бинарные данные

```
var reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function() {
    ws.send(reader.result);
}
```

Данные, считанные из файла, преобразуются в относительно новый для WWW тип – arrayBuffer – и в таком виде пишутся в сокет. Для достойного их приема модернизируем код на стороне сервера:

```
ws.on('message', function(message) {
            if(typeof message ==='object'){
        for (var key in clients)  {
                    clients[key].send(message);
                }
            } else {
            for (var key in clients)  {
                    clients[key].send(JSON.stringify( { type: 'message',
message: message, author: id} ));
                }
            }
    })
})
```

В общем-то, все, что мы делаем, обнаружив, что присланное сообщение является бинарным объектом (буфером), – просто рассылаем его всем клиентам в неизменном виде. Соответственно, и там надо их встретить:

```
ws.onmessage = function(event) {
    if(typeof event.data === 'object'){
        var blob = event.data;
       if (window.webkitURL) {
```

```
        URL = webkitURL;
      }

      var uri = URL.createObjectURL(blob);
      var img = document.createElement("img");
      img.src = uri;
      document.getElementById('subscribe').appendChild(img);

  } else{

  var message = JSON.parse(event.data);
        var text = "";
  switch (message.type) {
     case "info": {
```

***

Теперь все – WebSocket-чат готов. Честно говоря, легкость разработки немного окрыляет. Перспективы у этого вида клиент-серверного взаимодействия, по всей вероятности, огромные, но до сих пор для применения этой технологии существует одно значительное препятствие, и оно пока не собирается окончательно исчезать. Я говорю о поддержке WebSocket браузерами (вернее, о ее отсутствии). Нет, современные версии обозревателей (даже того самого!) с веб-сокетами «дружат», но больше, чем хотелось бы, пользователей все еще сидят на старом ПО. И мы не хотим их терять. И не будем!

Socket.io – webSockets для пролетариата

Итак, преимущества веб-взаимодействия на основе WebSocket и перспектив приложений, на них основанных, очевидны. Но вот с реальностью в виде браузеров, не поддерживающих эту технологию, предстоит, как показывает опыт, бороться еще очень долго. А хотелось бы творить уже здесь и сейчас.

Реальное время для всех!

На счастье, у нас есть возможность воспользоваться преимуществами взаимодействия реального времени прямо сейчас. Есть – благодаря одному из сооснователей LearnBoost Guillermo Rauch и его замечательной библиотеке socket.io.

Эта библиотека призвана осуществлять взаимодействие реального времени тем способом, который наиболее подходит для его участников. На практике это обычно обозначает – тем, который доступен для данного браузера. Вот список доступных методов:

- WebSocket;
- Adobe Flash Socket;
- Server-Sent Events;
- XHR long polling;
- XHR multipart streaming;
- Forever Iframe;
- JSONP Polling;
- ActiveX HTMLFile.

Так, для современной версии браузера Google Chrome socket.io будет использовать WebSocket, при отсутствии в клиенте поддержки такой технологии задействует flash, а если и его нет, будет применяться, например, Forever Iframe. Подобный подход позволяет работать в реальном времени практически во всех используемых в настоящее время браузерах. Вот список браузеров, которые поддерживаются сейчас:

- Internet Explorer 5.5+;
- Safari 3+;
- Google Chrome 4+;
- Firefox 3+;
- Opera 10.61+;
- iPhone Safari;
- iPad Safari;
- Android WebKit;
- WebOs WebKit.

Еще socket.io добавляет к WebSocket-взаимодействию несколько дополнительных возможностей. Но об этом позже.

Начинаем работать с socket.io

```
npm install socket.io
```

Простейший сокет-сервер с использованием socket.io выглядит так (serverio.js):

```
var io = require('socket.io').listen(8080);

io.sockets.on('connection', function (socket) {
    socket.emit('hello', 'hello from io!');
});
```

Клиент будет приблизительно таким:

```
<!doctype html>
    <head>
        <title>SocketIO Client</title>
        <meta charset="utf-8">
```

```
    </head>
        <script src="http://localhost:8080/socket.io/socket.io.js"></script>
    <body>
        <script>
            var socket = io.connect('http://localhost:8080');
                socket.on('hello', function (data) {
                alert(data);
            });
        </script>
    </body>
</html>
```

Тут все должно быть понятно. При правильной установке клиентский код просто обязан найти библиотеку socket.io по url http://[host]:[port]/socket.io/socket.io.js и задействовать её возможности (параметры [host] и [port] должны соответствовать хосту, на котором запущен серверный скрипт, и порту, который этот скрипт слушает).

Запустим сервер:

```
$ node serverio.js
   info  - socket.io started
```

Вывод в консоль отладочной информации – инициатива самой socket.io, теперь открываем в браузере url *http://localhost/node/io/client.html* (рис. 21). У нас все получилось. В консоль, между тем, продолжает поступать разнообразная информация. Она местами довольно полезна:

```
   debug - served static content /socket.io.js
   debug - client authorized
   info  - handshake authorized uOTsh7Nk6aPPZzGuiUDk
   debug - setting request GET /socket.io/1/websocket/uOTsh7Nk6aPPZzGuiUDk
   debug - set heartbeat interval for client uOTsh7Nk6aPPZzGuiUDk
```

Рис. 21 ❖ Hello Socket.io

```
debug - client authorized for
debug - websocket writing 1::
debug - websocket writing 5:::{"name":"hello","args":["hello from io!"]}
debug - emitting heartbeat for client uOTsh7Nk6aPPZzGuiUDk
debug - websocket writing 2::
debug - set heartbeat timeout for client uOTsh7Nk6aPPZzGuiUDk
debug - fired heartbeat timeout for client uOTsh7Nk6aPPZzGuiUDk
info  - transport end (heartbeat timeout)
debug - set close timeout for client uOTsh7Nk6aPPZzGuiUDk
debug - cleared close timeout for client uOTsh7Nk6aPPZzGuiUDk
debug - discarding transport
```

Важное замечание – в данном примере мы запустили socket.io как standalone-приложение. Оно отлично справляется со своей ролью, но эта роль ограничена функциями WebSocket-сервера. Другой вариант использования технологии заключается в запуске socket.io на существующем HTTP-сервере:

```
var server = require('http').Server();
var io = require('socket.io')(server);
io.on('connection', function(socket){
  socket.on('event', function(data){
    ..................................
  });
    ..................................
});
server.listen(8080);
```

В реальном приложении подобный способ применения дает массу возможностей, ну а мы в наших примерах пока ограничимся одиночным socket.io.

Теперь попробуем сотворить при помощи socket.io что-нибудь полезное. Ну, например... Да ладно! Что может быть полезнее чата? Правда?

Простой чат на socket.io

Сначала соорудим простой пользовательский интерфейс:

```
<script>
  window.onload = function() {
      document.getElementById('startButton').onclick = function(){
      iochat(document.getElementById('nickname').value);
  }
  }
</script>
Введите ник: <input type ="text" id="nickname">
<input type="button" value = "Start chat" id="startButton">
```

Функцию **chatio** реализуем следующим образом:

```
function iochat(nick){
    socket = io.connect('http://localhost:8080');
    socket.on('connect', function () {
        socket.send(nick);
        socket.on('message', function (msg) {
            alert(msg);
        });
    });
}
</script>
```

Пока все, что она делает, – соединяется с сервером, представляется и ждет ответа. Теперь напишем соответствующий код на сервере:

```
var io = require('socket.io').listen(8080);
io.sockets.on('connection', function (socket) {
  socket.on('message', function (msg) {
      var time = (new Date).toLocaleTimeString();
      socket.send("Hello "+msg+"!");
      socket.broadcast.send(time+" К нам присоеденился "+msg);
      console.log(msg+" connect! "+time);
});
```

Тут все просто. При соединении сокет-сервер начинает «слушать» клиента и при получении сообщения посылает приветствие новому участнику чата. Метод **socket.broadcast.send** рассылает сообщения о новом участнике всем остальным жителям чата. Дабы проверить все это в работе, достаточно войти в чат с разных браузеров (можно даже на одном компьютере (рис. 22).

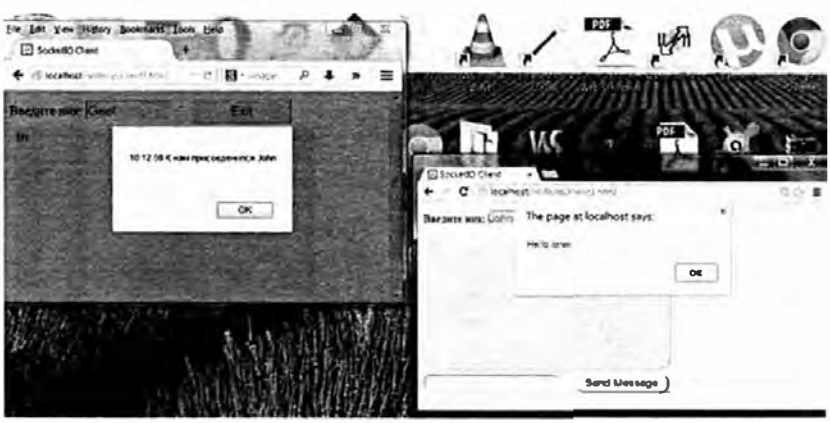

Рис. 22 ❖ Общаемся браузерами

Усложняем

Разумеется, с текстовым форматом сообщений чат организовать сложновато – ведь сообщения от клиентов и сервера надо как-то различать. Поэтому логичнее обмениваться в формате JSON, как мы это уже делали, организуя чат на «чистых» WebSockets. Немного изменим код клиента:

```
function iochat(nick){
  socket = io.connect('http://localhost:8080');
  socket.on('connect', function () {
       socket.json.send({"type":"hello","nick" : nick});
      socket.on('message', function (msg) {
       if(msg.type='hello' || 'announce'){
          alert(msg.message);
       }
   });
});
```

и сервера:

```
socket.on('message', function (msg) {
    var time = (new Date).toLocaleTimeString();
     if(msg.type == 'hello'){
        socket.json.send({"type":"hello", "message":"Hello "+msg.nick+"!"});
        socket.broadcast.json.send({"type":"announce", "message":time+" К нам
присоединился "+msg.nick});
        console.log(msg.nick+" connect! "+time);
     }
});
```

Тут стоит обратить внимание на возможность обмениваться информацией в JSON-формате, прямо «из коробки».

Теперь можно заняться сообщениями чата. Сначала добавим элементы интерфейса для ввода и отображения сообщений:

```
<span id="nut" >Введите ник: </span>
<input type ="text" id="nickname">
<input type="button" value = "Start chat" id="startButton">
<div id="board"></div>
<input type ="text" id="message" >
<input type="button" value = "Send Message" id="input">
```

Теперь модернизируем клиентский код:

```
<script>
 window.onload = function() {
    document.querySelector('#startButton').onclick = function(){
      iochat(document.getElementById('nickname').value, 'hello');
      document.querySelector('#startButton').disabled = true;
```

```
    }
    document.querySelector('#input').onclick = function(e) {
      message = escape(document.querySelector('#message').value);
      iochat(
          document.getElementById('nickname').value, 'message'
      );
      document.querySelector('#message').value = '';
    }
}
function iochat(nick){
  socket = io.connect('http://localhost:8080');
  socket.on('connect', function () {
    sendMessage(socket,"hello",  nick);
    document.querySelector('#input').onclick = function(e) {
      var message = document.querySelector('#message').value;
      sendMessage(socket, "message", message);
      printMessage(message, 1);
      document.querySelector('#message').value = '';
    }
  socket.on('message', function (msg) {
    if(msg.type=='hello'  || 'announce'){
      alert(msg.message);
    }
    if(msg.type=='message'){
      printMessage(msg.message, 0)
    }
  });
});
}

  functio endMessage(socket, type, text){
    socket.json.send({"type":type,"message" : text});
  }

  function printMessage(text, own){
    if(own ==1 )слишком
      text = "<b>"+text+"</b>";
    var chat = document.querySelector('#board');
    chat.innerHTML += mtext+"<br>";
  }
</script>
```

Клиент стал заметно «толще», но ничего нового мы здесь не применили – просто добавили еще один тип сообщений и возможность их вывода. Серверная часть чата потребует гораздо меньшей доработки:

```
var io = require('socket.io').listen(8080);
io.sockets.on('connection', function (socket) {
  socket.on('message', function (msg) {
      console.log(msg);
```

```
    var time = (new Date).toLocaleTimeString();
     if(msg.type == 'hello'){
        socket.json.send({"type":"hello", "message":"Hello "+msg.message+"!"});
        socket.broadcast.json.send({"type":"announce", "message":time+" К нам
присоединился "+msg.message});
        console.log(msg.nick+" connect! "+time);
     }
     if(msg.type == 'message'){
         console.log("test");
         socket.broadcast.json.send({"type":"message", "message":msg.message});
     }
});
});
```

Теперь можно болтать (рис. 23).

Рис. 23 ❖ Чат на Socket.io

Совершенствуем приложение – дополнительные возможности socket.io

Чат мы написали быстро, но, по сути, он мало чем отличается от созданного в первой части главы. Однако популярность библиотеки socket.io (да и её размеры) подсказывает, что эмуляцией WebSocket взаимодействия её возможности не ограничиваются. И это действительно так. Правда, выбранный нами формат приложения – веб-чат – эффектной демонстрации всех возможностей socket.io не производит, но реал-тайм – браузерную стратегию – мы тут писать точно не будем

(по крайней мере, пока) и постараемся применить возможности библиотеки к тому, что есть. Прежде всего рассмотрим события серверной и браузерной частей socket.io.

Надо сказать, что хоть и идея построения socket.io-приложений состоит в том, что обе её части – серверная и клиентская – имеют одинаковые свойства и методы, список обработчиков событий для них различается, у сервера он сильно меньше. Собственно, предопределенных событий только три:

- **connection** – событие наступает при установке соединения с клиентом;
- **message** – событие наступает при получении сообщения от клиента;
- **disconnect** – дисконект, он и тут дисконект, то есть разрыв соединения.

Если взглянуть на документацию, то можно обнаружить еще одно событие – anything, и оно, наверное, самое главное, ведь это JavaScript, Node.js и EventEmiter! Другими словами, события можно назначить самому, чем мы сейчас и займемся, немного структурировав серверный код:

```
if(msg.type == 'hello'){
  socket.emit('hello', {"message", "Hello"+msg.message+"!"});
  socket.broadcast.emit("announce",
                        {"message":time+
                        " К нам присоединился "+msg.message});
  console.log(msg.nick+" connect! "+time);
}
if(msg.type == 'message'){
  socket.broadcast.emit("post",{"message":msg.message});
}
```

Тут мы эмитируем три события:

- **hello** – приветствие нового участника;
- **annonce** – объявление о новом участнике остальному сообществу чата;
- **post** – поступление в чат нового сообщения.

Можно добавить сюда сколько угодно своих событий. Например, извещение о достижении определенного числа участников или наступлении нового времени суток.

На клиентской стороне эти события обрабатываются самым естественным образом, код при этом становится более легким и модифицируемым:

```
    }
    socket.on('hello', function (msg) {
        alert(msg.message);
    });
    socket.on('announce', function (msg) {
        alert(msg.message);
    });
    socket.on('post', function (msg) {
        printMessage(msg.message, 0)
    });
  });
}
```

Так гораздо лучше, правда?

У браузерной части socket.io событий будет немного больше. Перечислим их в примерном порядке следования:

- **connecting** – событие наступает в процессе установления соединения с сервером;
- **connect_failed** – событие наступает при неудачной попытке соединения;
- **connect** – событие наступает при установке соединения с сервером;
- **message** – событие наступает при получении сообщения от сервера;
- **disconnect** – событие наступает при разрыве соединения с сервером;
- **reconnecting** (может возникать неоднократно) – событие наступает при попытке восстановления соединения;
- **reconnect** – событие наступает при восстановлении соединения;
- **error** – ну как без этого;
- **anything** – да! И здесь тоже.

Понятно, что, несмотря на такое «богатство», нам больше всего интересно последнее событие, с помощью которого мы сделаем код клиента еще более вменяемым:

```
function iochat(nick){
  socket = io.connect('http://localhost:8080');
  socket.on('connect', function () {
    socket.emit('hello',nick);
    document.querySelector('#input').onclick = function(e) {
      var message = document.querySelector('#message').value;
       socket.emit('post',message);
      printMessage(message, 1);
      document.querySelector('#message').value = '';
    }
```

Естественно, и на стороне сервера теперь требуется доработка:

```
io.sockets.on('connection', function (socket) {
  socket.on('hello', function (nick) {
      var time = (new Date).toLocaleTimeString();
      socket.json.emit('hello', { "message":"Hello "+nick+"!"});
      socket.broadcast.emit("announce", { "message":time+" К нам присоединился
"+nick});
      console.log(nick+" connect! "+time);
  });
  socket.on('post', function (message) {
          socket.broadcast.emit("post", message);
    });
});
```

Код стал гораздо приятнее, с ним легче работать, его проще модифицировать, но вот дополнительной функциональности у нашего чата не появилось. Исправим это. По крайней мере, очевидно, что сообщения от участников чата нужно различать, они должны быть авторизованы. Задача проста – требуется сохранять в объекте сокета имя (nick) клиента. И реализуется она просто – всего несколько строк кода на стороне сервера:

```
socket.on('hello', function (nick) {
    time = (new Date).toLocaleTimeString();
    socket.set("name", nick, function(){
        socket.json.emit('hello', { "message":"Hello "+nick+"!"});
        socket.broadcast.emit("announce", { "message":time+" К нам присоединился
"+nick});
        console.log(nick+" connect! "+time);
    });
});
socket.on('post', function (message) {
    var time = (new Date).toLocaleTimeString();
    socket.get('name', function (err, name) {
        console.log(name+": "+message);
        message = name + ": " + message;
        socket.broadcast.emit("post", {"message": message});
    });
});
```

Да-да, вот такие сеттеры и геттеры.

Кстати, если в процессе экспериментов перезапускаете серверный скрипт (а как по-другому?), то должны обратить внимание – клиент при этом не «отваливается» и восстанавливает соединение немедленно после очередного запуска сервера. Это замечательно, ведь иначе во многих случаях подобную функциональность пришлось бы реализовать вручную. Правда, учитывая, что мы как порядочные разработ-

чики должны реализовать функционал выхода клиента из чата. На счастье, это совсем просто. Сначала обозначим кнопку:

```
document.querySelector('#startButton').onclick = function(){
    iochat(document.getElementById('nickname').value);
    document.querySelector('#startButton').value = "Exit";
}
```

И напишем вызов метода:

```
function iochat(nick){
    ..................................
    socket.emit('hello',nick);
    var button = document.querySelector('#startButton');
    button.onclick = function() {
        socket.disconnect();
    }
```

На серверной стороне тоже ничего сложного:

```
    socket.on('disconnect', function () {
        time = (new Date).toLocaleTimeString();
        socket.get('name', function (err, name) {
            socket.broadcast.emit("announce", { "message":time+" Нас покинул
"+name});
            console.log(name+" left "+time+"...");
        });
    });
```

Пространства имен

Еще одной интересной возможностью soket.io является использование механизма пространства имен (namespace). Он позволяет создавать несколько подключений к серверу, например для различных компонентов приложения (реально соединение, скорее всего, будет одно, но применение namespace позволяет создать несколько логических соединений со своим собственным окружением). Зачем это может понадобиться? Во-первых, браузеры способны открывать ограниченное число одновременных подключений, и этот лимит можно таким образом расширить. Другая причина состоит в элементарном удобстве (да и культуре) проектирования – каждой задаче в составе бизнес-логики приложения можно выделить отдельное соединение.

Работа с пространством имен в socket.io выглядит так:

```
var io = require('socket.io').listen(8080);
var chat = io
  chat.of('/chat')
  chat.on('connection', function (socket) {
```

```
    socket.emit('a message', {
        that: 'only'
      , '/chat': 'will get'
    });
    chat.emit('a message', {
        everyone: 'in'
      , '/chat': 'will get'
    });
  });

var news = io
  news.of('/news')
  news.on('connection', function (socket) {
    socket.emit('item', { news: 'item' });
  });
Это сервер, а вот клиент:
<script>
window.onload = function() {
     var chat = io.connect('http://localhost:8080/chat:8080')
     var news = io.connect('http://localhost:8080/news:8080');
     chat.on('connect', function () {
     chat.emit('hello');
  });
  news.on('news', function () {
       news.emit('woot');
  });

  }
</script>
```

Это чуть модифицированный код из руководства, и он во многом просто обозначает намерения, но я думаю, что идея ясна. Состоит она в том, чтобы и клиент, и сервер отдельно обрабатывали сообщения и прочие события, поступающие по каналу чата, а отдельно передавали с сервера и обрабатывали на клиенте поступающие новости. Кстати, таким образом можно дополнить наш чат каналом новостей или просто передавать список вошедших и вышедших из сервиса. Но реализацию этого функционала я оставляю в качестве домашнего задания. А мы посмотрим, что у socket.io есть еще интересного.

«Летучие» сообщения

Еще одна замечательная возможность socket.io – отправка нестабильных (**volatile**) сообщений, не требующих обязательной доставки. Это может понадобиться при отсутствии гарантированного канала связи, отказов обслуживаний ввиду высокой нагрузки и тому подобных ситуациях, некоторые из которых уже не считаются критическими, –

это норма своевременного веба: клиенты мобильны, а серверы высоконагружены. Для случаев, когда недоставка сообщений допустима и не должна вызывать сбои, и придуманы **volatile**:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
    setInterval(function () {
     socket.volatile.emit('alarm', 'Хватит чатиться!');
    });
  }, 10000);
});
```

Извещения (acknowledgements)

Эта возможность, появившаяся в socket.io относительно недавно, хоть и удостоена собственного названия, представляет собой просто расширение механизма функций обратного вызова.

Сервер:

```
var io = require('socket.io').listen(8080);

io.sockets.on('connection', function (socket) {
  socket.on('foo', function (name, fn) {
    fn('yes!');
  });
});Клиент:
<script src="http://localhost:8080/socket.io/socket.io.js">
</script>
<script>
  var socket = io.connect('http://localhost:8080');
  socket.on('connection', function () {
    socket.emit('foo', 'bar', function (data) {
      console.log(data); // 'yes!'
    });
  });
</script>
```

Все еще проще, правда?

Конфигурация

Хотя состояние модуля socket.io «из коробки» вполне рабочее и параметры по умолчанию нечасто нуждаются в коррекции, как и всякая сложная исполняемая среда, он имеет API для собственной конфигурации. Метод **configure()** в его арсенале предназначен специально для этого:

```
var io = require('socket.io').listen(80);

io.configure('production', function(){
```

```
  io.enable('browser client etag');
  io.set('log level', 1);

  io.set('transports', [
    'websocket'
  , 'flashsocket'
  , 'htmlfile'
  , 'xhr-polling'
  , 'jsonp-polling'
  ]);
});

io.configure('development', function(){
  io.set('transports', ['websocket']);
});
```

Метод **configure()** в качестве первого аргумента принимает режим работы сервера, тут мы даем различные настройки для режима разработки (development) и промышленной эксплуатации (production) программы. Опустив этот аргумент, мы сделаем настройки, доступные всем режимам. Строго говоря, такие названия режимов не стандартизированы, можно применять любые ('testing', 'justforfun' и т. д.). Задается режим работы значением переменной окружения NODE_ENV:

```
$ export  NODE_ENV=production
```

Второй аргумент является функцией обратного вызова, устанавливающей набор параметров.

Директивой мы позволяем клиенту делать условные запросы, используя entity tag. Есть еще несколько подобных настроек: 'browser client minification', 'browser client gzip' – тут все понятно по их названиям.

Метод **set()** устанавливает значение параметра, причем ввиду открытости системы сам параметр и его значение могут быть произвольными, и строка вида **io.set('foo', 'bar')**; ошибки не вызовет. Это касается работы метода **configure()** не только в socket.io. Что касается предопределенных параметров конфигурации, то их относительно немного. Вот основные:

- **transports** – перечисление доступных способов взаимодействия с сервером. Очередность имеет значение.
- **log level** – уровень журналирования взаимодействия:
  0 – error;
  1 – warn;
  2 – info;

3 – debug.
По умолчанию – 3.

- **resource** – начальная точка, от которой socket.io ищет входящих соединений, должна совпадать у клиента и сервера. По умолчанию – /socket.io.
- **authorization** – еще одна интересная возможность socket.io, о которой мы не упомянули. Может, зря, но пусть освоение авторизации тоже будет домашним заданием для читателя. По умолчанию этот параметр равен false, но при необходимости можно организовать что-нибудь вроде этого:

```
io.configure(function(){
  io.set('authorization', function(handshakeData, callback){
        findbyIP(handshakeData.address.address,
          function(err, data) {
            if(err)return callback(err);
            if (data.authorized) {
              handshakeData.foo = 'bar';
              for(var prop in data){
                handshakeData[prop] = data[prop];
              }
              callback(null, true);
            } else {
              callback(null, false);
            }
        })
    });
});
```

- **origins** – задает адреса (в формате url:port), с которых разрешены соединения с sockei.io-сервером. По умолчанию можно всем (*:*).
- **store** – определяет место хранения данных клиента.
- **close timeout** – срок закрытия сокет-соединения при отсутствии активности (по умолчанию 60 мин).

Остальные настройки можно изучить на страницах документации socket.io [7].

Что в итоге? Мы обнаружили, что socket.io – прекрасный инструмент, который прямо сейчас можно использовать для решения самого широкого круга задач.

Пирамиды судьбы – асинхронный поток выполнения и как с ним бороться

Ну вот. С асинхронностью предлагается еще и бороться. Зачем? Во вступительных главах довольно подробно разъяснялись преимущества асинхронного, неблокирующего выполнения процессов, да и приведенные выше примеры вроде бы хорошо демонстрируют все плюсы подобного подхода. Все это так, но, к сожалению, в подобного рода взаимодействии есть и существенные минусы, о которых я пока не упоминал, но с которыми вы наверняка столкнетесь (если не столкнулись уже).

О чем я говорю? Не будем теоретизировать, разберем небольшой пример – не очень сложную и вполне жизненную задачу.

Условия следующие.

В файловой системе хранится музыка моих любимых рок-групп, в самом что ни на есть пиратском mp3-формате. Естественно, композиции сгруппированы по папкам-альбомам, альбомы сгруппированы в папки по исполнителю (рис. 24). Задача заключается в составлении

Рис. 24 ❖ Структура коллекции

списка всех произведений. Вроде ничего сложного, запас уже имеющихся знаний должен быть достаточным для успешного ее решения. Приступим.

Начинаем строить пирамиды

Для начала организуем обход папок:

```
var fs = require('fs');
var base ='G:/bands';
fs.readdir(base, function(error, items){
    console.log(items);
    items.forEach(function(band){
        console.log(band);
        fs.readdir(base+'/'+band, function(error, subitems ){
            subitems.forEach(function(album){
                console.log("  "+album);
            });
        });
    });
});
```

Как видите, мы делаем это с помощью последовательности вызова функций обратного вызова. Сначала мы получаем список папок (исполнителей), потом итератором forEach() обходим его, выводя название и, в свою очередь, получая для каждой список подпапок (альбомов). Затем проходим итератором и по альбомам, выводя, опять же функцией обратного вызова, в консоль их названия. Вроде все логично, и все действия выполняются в необходимой последовательности. Разве что визуально небольшой код уж слишком растянулся по ширине. Но что с этими кэлбэками поделаешь? Попробуем запустить код:

```
$ node async
Pink Floyd
The Beatles
Traffic
  1967 - The Piper at the Gates of Dawn
  1968 - A Saucerful of Secrets
  1969 - More
  1969 - Ummagumma
  1970 - Atom Heart Mother
  1968 - Mr. Fantasy
  1968 - Traffic
  1970 - John Barleycorn Must Die
  1963 - Please Please Me
  1963 - With The Beatles
```

```
1964 - A Hard Day's Night
1964 - Beatles For Sale
1965 - Help!
1965 - Rubber Soul
1966 - Revolver
1967 - Sgt Pepper's Lonely Hearts Club Band
```

Немного не то, что мы хотели, правда? Вместо задуманного формата:

```
Группа1
    Альбом1
    Альбом2
    Альбом3
Группа2
    Альбом1
    Альбом2
```

получилось безобразие – названия групп сбились в кучу в начале списка, альбомы высыпались в конце, да еще и в неправильном порядке. Последнее обстоятельство, если вы не очень хорошо знакомы с английским роком, можно и не заметить, но попробуйте запустить программу несколько раз – порядок альбомов, скорее всего, будет разным. Пока просто отметим этот факт и продолжим. Теперь нам надо перебрать все файлы в папке альбома и вывести названия тех из них, которые являются композициями в формате mp3. Опять возводим кэлбэки:

```
subitems.forEach(function(album){
    console.log("  "+album);
    fs.readdir(base+'/'+band+'/'+album, function(error, files ){
       files.forEach(function(song){
         var file = base+'/'+band+'/'+album+'/'+song;
         fs.stat(file, function(err,state){
            if(state.isFile()){
                fs.readFile(file,'utf8', function(error,data){
                   if(data.substr(0,3) == 'ID3'){
                      var size = Math.round( state.size/(102 4*102 4)* 100 ) /
100+'Mb';
                      console.log("      " + song + " - " + size);
                   }
                });
            }
         });
       });
    }
});
```

Тут мы перебираем все, что найдем в папке альбома, отбираем файлы, считываем их содержимое и проверяем наличие ID3-метатег.

(Я не спорю, есть и менее затратные способы идентифицировать MP3-формат, но так, по крайней мере, точнее, чем по расширению файла, да и мне, как тому прапорщику из анекдота, сейчас производительность только мешает.) Если все в порядке, выводим название композиции в консоль, снабдив его полезной информацией – «весом» файла (раз уж мы создаем объект **fs.stat**, почему бы не воспользоваться информацией, которую он предоставляет?).

Теперь запустим итоговый сценарий. Весь вывод я приводить не буду, только фрагмент:

```
$ node async
[ 'Pink Floyd', 'The Beatles', 'Traffic' ]
Pink Floyd
The Beatles
Traffic
  1967 - The Piper at the Gates of Dawn
  1968 - A Saucerful of Secrets
  1969 - More
  1969 - Ummagumma
  1970 - Atom Heart Mother
  1963 - Please Please Me
  1963 - With The Beatles
  1964 - A Hard Day's Night
  1964 - Beatles For Sale
  1965 - Help!
  1965 - Rubber Soul
  1966 - Revolver
  1967 - Sgt Pepper's Lonely Hearts Club Band
  1968 - Mr. Fantasy
  1968 - Traffic
  1970 - John Barleycorn Must Die
      02 - Berkshire Poppies.mp3 - 7.33 Mb
      01 - Heaven Is In Your Mind.mp3 - 10.59 Mb
      04 - No Face, No Name, And No Number.mp3 - 8,76 Mb
      05 - Dear Mr. Fantasy.mp3 - 13,60 Mb
      09 - Hope I Never Find Me There.mp3 - 5386883
      07 - Utterly Simple.mp3 - 8160007
      08 - Coloured Rain.mp3 - 6753571
      03 - House For Everyone.mp3 - 5173700
      06 - Dealer.mp3 - 7913387
      02 - The Nile Song.mp3 - 8325673
      01 - Cirrus Minor.mp3 - 12776231
      03 - Crying Song.mp3 - 8596389
      10 - Giving To You.mp3 - 10579987
      04 - Who Knows What Tomorrow May Bring.mp3 - 7971691
      04 -  Up The Khyber.mp3 - 5356393
      12 - Feelin' Alright  [Mono Single Mix].mp3 - 10001931
```

```
09 - Ibiza Bar.mp3 - 8014625
10 - More Blues.mp3 - 5352547
06 - Vagabond Virgin.mp3 - 13113580
06 - Cymbaline.mp3 - 11660705
12 - A Spanish Piece.mp3 - 2652077
03 - Don't Be Sad.mp3 - 8315399
02 - Pearly Queen.mp3 - 10623579
01 - You Can All Join In.mp3 - 8919371
10 - Means to an End.mp3 - 6575653
```

Ну вот, теперь хаос очевиден. Композиции оказались после перечисления всех альбомов и в весьма произвольном порядке.

Причины этого понятны, более того, странно, если бы все происходило по-другому. Дело в том, что практически все операции происходят асинхронно, а значит, и несвязанно друг с другом. Самая первая команда **forEach()**, вызывая функцию обратного вызова, не дожидается окончания ее работы, а продолжает выполняться. Она успевает отработать, выведя список групп, еще до того, как считывается содержимое первой папки – альбома. Так же поток управления при обходе альбомов. Ну а при чтении файлов начинается настоящее соревнование – кто быстрей «прочитался», тот и будет выведен первый.

Конечно, с этими неприятностями можно бороться – например, собрать все названия в сложный массив и вывести его обычными синхронными методами по окончании сбора сведений. Правда, отследить сам этот момент – задача не очень тривиальная.

Другую проблему представляет то футуристическое нагромождение кэлбэков, в которые превратился наш код (рис. 25). Да-да, это те самые «Пирамиды судьбы», вынесенные в название главы, и это то, за что критикуют javaScript. Код уже довольно сложно читается. А что, если мы добавим какой-нибудь дополнительный функционал?

Еще одна проблема – обработка ошибок. Вы заметили, что я в вышеприведенном коде им вообще не уделял внимания? И зря. На каждом этапе они могут вмешаться в ход исполнения программы – это же файлы! По сути, физические объекты, ну или, по крайней мере, объекты на реальном физическом носителе. Надо проверять наличие ошибок везде, но вы только представьте, во что превратится внешний вид кода, исполни я эти (в общем случае уместные) рекомендации!

В общем, с этим надо что-то делать. Решением в лоб могло бы стать использование синхронных методов, но это значит, что мы отказываемся от преимущества асинхронного доступа к файловой системе. Зачем тогда было вообще связываться с Node.js?

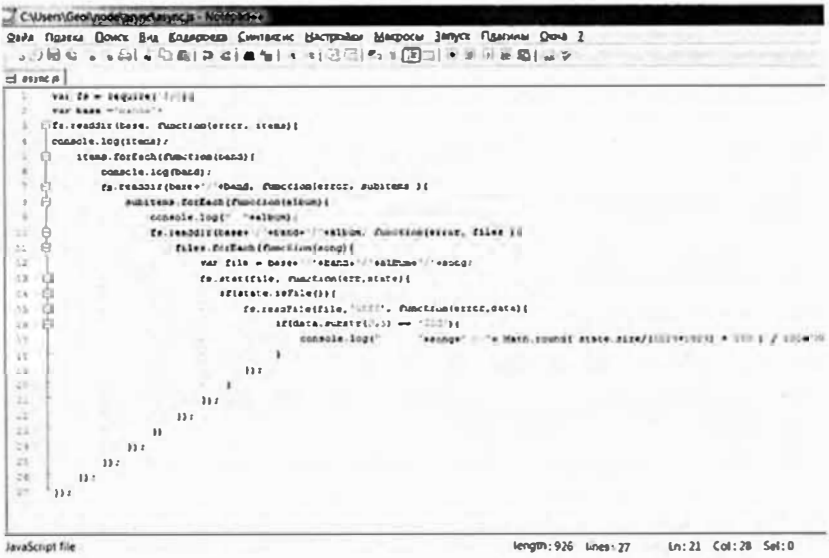

Рис. 25 ❖ Знакомо?

Долой анонимность!

Одним из не то чтобы решений, а просто выходов может стать отказ от анонимных функций (подобные действия входят в рекомендации по стилю кодирования Node).

Попробуем «отрефакторить» наш код, поименовав каждую функцию обратного вызова:

```
var fs = require('fs');
var base ='bands';
fs.readdir(base, function(error, items){
    console.log(items);
    items.forEach(function(band){
        path = base+'/'+band;
        readGroup(path, band);
    });
});
function readGroup(path, band){
console.log(band);
    fs.readdir(path, function(error, subitems ){
        subitems.forEach(function(album){
            readAlbum(path+'/'+album, album);
        });
```

```
    });
}

function readAlbum(path, album){
    console.log("  "+album);
    fs.readdir(path, function(error, files ){
        files.forEach(function(song){
            readSong(path+'/'+song,song);
        });
                })

}
function readSong(path,song){
    var file = path;
    fs.stat(file, function(err,state){
        if(state.isFile()){
            fs.readFile(file,'utf8', function(error,data){
                if(data.substr(0,3) == 'ID3'){
                    console.log("      "+song+" - "+ Math.round( state.size/
(1024*1024) * 100 ) / 100+'Mb');
                }
            });
        }
    });
}
```

Так все выглядит гораздо понятнее, с таким кодом можно работать (ведь и платят нам обычно не за ширину, а за длину написанного кода! Сарказм, если что). Но основная проблема остается – порядок (а вернее, непорядок) исполнения, от перехода к именованию функций, не поменяется. Так что же делать? Оставить тот код, с целью дальнейшей модификации, и посмотреть, нет ли среди средств Node.js чего-нибудь для решения данной проблемы.

Node.js control-flow

Естественно, не мы первые сталкиваемся с проблемами, которые порождает асинхронность. Создано довольно много средств контроля потока исполнения, основанных на различных принципах. Например, очень популярный модуль с не очень длинным названием q эти проблемы решает реализацией концепции обещаний (promises). Я не планировал вдаваться в подробности ни концепции, ни реализации, но грамотное применение q позволяет связать «пирамиды судьбы» контрактами, преобразовав такую знакомую нам картину:

```
step1(function (value1) {
    step2(value1, function(value2) {
```

```
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Какие-либо действия с value4
            });
        });
    });
});
```

в нечто более удобоваримое:

```
Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // какие-либо действия с value4
})
.catch(function (error) {
    // обработчик ошибок для каждого шага
})
.done();
```

Большую популярность имеет чрезвычайно удобный и простой в применении модуль Step – инструмент, организующий упрощенные параллельные и последовательные потоки исполнения. Вот пример его работы:

```
Step(
  function readSelf() {
    fs.readFile(__filename, this);
  },
  function capitalize(err, text) {
    if (err) throw err;
    return text.toUpperCase();
  },
  function showIt(err, newText) {
    if (err) throw err;
    console.log(newText);
  }
);
```

Тут мы получаем гарантию, что изменение регистра не будет применяться до получения текста файла (асинхронная и чреватая ошибками операция), а вывод текста не начнется до его обработки.

Существуют еще интересные модели, такие как slade, co или bluebird, но во всем этом многообразии один модуль выделяется комплексным универсальным подходом.

Async – берем поток исполнения в свои руки

Модуль async – это, наверное, самый проработанный механизм управления асинхронным потоком управления. Впрочем, не только. В его арсенале – отличный выбор инструментов для работы с коллекциями данных, в том числе собственные реализации некоторых стандартных функций JavaScript. С этого инструментария и начнем изучение возможностей модуля.

Инструменты async

Далее перечислены методы async для работы с данными. Почти все они имеют общие аргументы:

- **arr** – массив для итерации;
- **iterator(item, callback)** – конструкция, итеративно применяющая к каждому элементу массива функцию обратного вызова.
- **callback(result)** – функция обратного вызова, вызываемая после завершения работы функции итератора. Результатом будет значение true или false в зависимости от результата асинхронной проверки.
- **async.each(array, iterator, calback)** – метод применяет функцию iterator к каждому члену массива array. Работа этого метода происходит параллельно, то есть теоретически итератор применяется сразу ко всем субъектам массива. Вторым аргументом функции-итератора является кэлбэк-функция, возвращающая в качестве значения

```
var async = require('async');
var toDouble = function (num, callback) {
  console.log(num * 2);
  return callback(null);
};async.each([1, 2, 3, 4], toDouble, function (err) {
  if(!err)
  console.log("Done!");
});
)
```

вывод:

```
$ node async
2
4
```

```
6
8
Done!
```

Поскольку обработка папок происходит параллельно, естественно, что результаты могут появиться в ином порядке, чем исходные данные располагались в массиве. Чтобы продемонстрировать такой случай, чуть-чуть изменим функцию-итератор, добавив перед выводом случайную задержку:

```
var toDouble = function (num, callback) {
  setTimeout(function () {
    console.log( num);
    return callback(null);
  }, 5000 * Math.random());
};
```

Теперь вывод будет разнообразнее:

```
$ node async
3
4
2
1
Done
$ node  asyncut
4
1
3
2
Done
$ node  asyncut
2
4
```

- **async.map(arr, iterator, callback)** – метод применяет функцию iterator к каждому члену массива array и в результате передает функции обратного вызова новый массив:

```
var toDouble = function (num, callback) {
  return callback(null, num * 2);
};

async.map([1, 2, 3, 4], toDouble, function (err, results) {
  console.log(results);
});
```

Вывод:

```
$ node async
2
```

```
4
6
8
[ 2, 4, 6, 8 ]
```

Обработка данных также происходит параллельно, но порядок значений массива не будет нарушен. Это касается и следующего метода.

- **async.filter(array, iterator, callback)** – метод возвращает массив, отфильтрованный с помощью функции-итератора:

```
var isEven = function (num,callback) {
  return  callback( num%2 == 0);
};async.filter([1, 2, 3, 4], isEven, function (results) {
  console.log(results);
});
```

Вывод:

```
$ node node async
[ 2, 4 ]
```

В комплекте идет еще метод **reject(arr, iterator, callback)**, действие которого прямо противоположно.

- **async.reduce(array, memo, iterator, callback)** – метод редуцирует массив в единственное значение с помощью последовательных действий функции-итератора. **Memo** – начальное значение редуцированного результата:

```
function addAll(memo, item, callback) {
    callback(null, (memo + item));
}


async.reduce([1,2,3], 8,addAll, function(err, result){
  console.log( result);
});
```

Данное действие выполняется последовательно, и для асинхронного паттерна **map/reduceit** больше подходит **Array.prototype.reduce()**.

- **detect(arr, iterator, callback)** – метод возвращает первое из значений массива **array**, с которым функция-итератор вернет истину. Метод выполняется параллельно, при этом вернется то значение, проверка которого закончится раньше:

```
async = require('async');
var isEven = function (num,callback) {
```

```
  return  callback( num%2 == 0);
};

async.detect([1,3,7,4,11,2], isEven, function(result){
    console.log(result);
});
```

Вывод:

```
$ node async.js
4
```

- **sortBy(arr, iterator, callback)** сортирует массив по результатам работы функции-итератора:

```
async.sortBy(['file1','file2','file3'], function(file, callback){
    fs.stat(file, function(err, stats){
        callback(err, stats.mtime);
    });
}, function(err, results){
    //  тут результатом будет массив названий файлов,
    //  отсортированный по дате их модификации
});
```

- **some(arr, iterator, callback)** возвращает истину, если какой-нибудь из элементов массива удовлетворяет асинхронному тесту.
- **every(arr, iterator, callback)** возвращает истину, если каждый элемент в массиве удовлетворяет асинхронному тесту. Например, следующий код выведет в консоль единицу, если все файлы, указанные в массиве, существуют:

```
async.every(['file1','file2','file3'], fs.exists, function(result){
    console.log(result);
});
```

Если бы мы использовали тут **async.some()**, достаточно было бы существования одного файла.

- **concat(arr, iterator, callback)** возвращает соединение результатов действий итератора над каждым членом массива, то есть буквальное соединение – конкатенацию. Следует учитывать, что итерации выполняются параллельно и начальный порядок исходных аргументов не гарантирован.

Ну как, набор впечатляет? С таким арсеналом, если постараться, можно соорудить вполне вменяемую модель выполнения сложного асинхронного приложения, причем не одну. Самая же хорошая новость заключается в том, что это уже сделано, причем в рамках рассматриваемого нами модуля.

Control Flow средствами async

Как уже было упомянуто, async предоставляет несколько моделей выполнения приложений, и некоторые из них мы сейчас разберем.

Series – серийный убийца неоднозначности

Основная идея этого метода заключается в возможности гарантированного последовательного исполнения асинхронных (и синхронных) функций.

Семантика series следующая: **series(tasks, [callback])**, где **tasks** – массив задач (функций), а **callback** – необязательная функция обратного вызова. На практике это выглядит примерно так:

```
var async = require('async');async.series([
    function(callback){
        console.log("one");
        callback(null, 'one');
    },
    function(callback){
        console.log("two");
        callback(null, 'two');
    },
    function(callback){
        console.log("three");
        callback(null, 'three');
    }],
function(err, results){
    console.log(results);
});
```

Результат:

```
$ node async
one
two
three
[ 'one', 'two', 'three' ]
```

Ничего особенного? Давайте немного изменим код:

```
    function(callback){
        setTimeout(console.log("one"),200);
        callback(null, 'one');
    },
```

Как известно, JavaScript-функции **setTimeout/setInterval** не приостанавливают выполнение программы, а отсрочивают вызов указанного метода, не прерывая потока выполнения. Тем не менее порядок

исполнения функций не изменится. Остальные функции просто «подождут».

Правда, не надо надеяться, что если, скажем, в заданных функциях есть вложенные асинхронные вызовы, то сами по себе они тоже организуются, – чудес не бывает. Но зато (по крайней мере, в JavaScript) бывают замыкания, с помощью которых можно решать реальные задачи. Вот, например, как можно организовать получение информации о композициях музыкального альбома и вывод их в правильном порядке:

```
var async = require('async');
var fs = require('fs');
var album = 'A Saucerful of Secrets';
var songs = fs.readdirSync('A Saucerful of Secrets');

callFunctions = new Array();
function makeCallbackFunc(file) {
     return function(callback) {
         fs.stat(album+'/'+file, function(error,data){
            data.name = file;
            callback(error, data);
         });
     };
}
for (var i = 0; i < songs.length; i++) {
    callFunctions.push(makeCallbackFunc(songs[i]));
}async.series(callFunctions,
    function(err, results){
     for (var i = 0; i < results.length; i++) {
        console.log(results[i].name+" "+(results[i].size/(1024*1024) * 100 ) /
100+' Mb');
     }
});
```

Результат:

```
$ node asyncs
(01) [Pink Floyd] Let There Be More Light.mp3 12.986783981323242 Mb
(02) [Pink Floyd] Remember A Day.mp3 10.489208221435547 Mb
(03) [Pink Floyd] Set The Controls For The Heart Of The Sun.mp3 12.552858352661
33 Mb
(04) [Pink Floyd] Corporal Clegg.mp3 9.684459686279297 Mb
(05) [Pink Floyd] A Saucerful Of Secrets.mp3 27.52260971069336 Mb
(06) [Pink Floyd] See-Saw.mp3 10.633848190307617 Mb
(07) [Pink Floyd] Jugband Blues.mp3 6.941537857055664 Mb

$
```

Распараллель это!

Следующий метод модуля async, **parallel()**, делает прямо противоположное предыдущему – **parallel(tasks, [callback])**.

Он запускает функции на параллельное выполнение, передавая управление функции обратного вызова, в тот момент, когда все функции либо отработают, либо повернут ошибку:

```
var async = require('async');async.parallel([
    function(callback){
        setTimeout(function(){
            console.log("1");
            callback(null, 'one');
        }, 200);
    },
    function(callback){
        setTimeout(function(){
            console.log("2");
            callback(null, 'two');
        }, 100);
    }
],
function(err, results){
    console.log(results);
});
```

Результат:

```
$ node parallel
2
1
[ 'one', 'two' ]
```

Как видите, функции действительно исполнялись параллельно, функция же обратного вызова отработала по завершении работы последней из них.

Попробуем применить этот метод на практике. Организуем вывод в консоль содержимого (списка файлов) нескольких папок – альбомов:

```
var async = require('async');
var fs = require('fs');
callFunctions = new Array();
albums = ['Beatles For Sale','Help!','Revolver'];
function makeCallbackFunc(dir) {
     return function(callback) {
         fs.readdir(dir, function(error,data){
            var album = {
```

```
                name: dir,
                songs: data
            }
            callback(error, album);
        });
    };
}
for (var i = 0; i < albums.length; i++) {
    callFunctions.push(makeCallbackFunc(albums[i]));
}async.parallel(callFunctions,
    function(err, results){
        for (var i = 0; i < results.length; i++) {
            console.log(results[i].name);
            songs = results[i].songs
            for (var j = 0; j < songs.length; j++) {
                console.log("    "+songs[j]);
              }
        }
});
```

Результат выглядит вполне пристойно:

```
$ node parallel.js
Beatles For Sale
    01. No Reply.mp3
    02. I'm A Loser.mp3
    03. Baby's In Black.mp3
...
    13. What You're Doing.mp3
    14. Everybody's Trying To Be My Baby.mp3
Help!
    01. Help!.mp3
    02. The Night Before.mp3
    03. You've Got To Hide Your Love Away.mp3
    04. I Need You.mp3
  ...
    27. Yesterday [1965 Stereo Mix].mp3
    28. Dizzy Miss Lizzy [1965 Stereo Mix].mp3
Revolver
    01. Taxman.mp3
    02. Eleanor Rigby.mp3
    03. I'm Only Sleeping.mp3
...
```

Модификация этого метода, **parallelLimit(tasks, limit, [callback])**, устанавливает лимит одновременно исполняемых процессов. Функция обратного вызова все равно будет исполнена по завершении всех входящих в массив функций, но отрабатывать они будут порциями с указанным размером (limit).

Живительный водопад (async.waterfall)

Патерн работы с асинхронным потоком, который наиболее соответствует решению нашей начальной задачи (точнее, тому способу решения, который мы выбрали), называется **waterfall** – водопад. По этой схеме выполнения функции вызываются по очереди, передавая друг другу контекст выполнения. Результаты всех этих вызовов передаются очередной функции обратного вызова в виде списка аргументов:

```
var async = require('async');async.waterfall([
    function(callback){
        callback(null, 'one', 'two');
    },
    function(arg1, arg2, callback){
      console.log(arg1+" "+arg2)
        callback(null, 'three');
    },
    function(arg1, callback){
        console.log(arg1)
        callback(null, 'done');
    }
], function (err, result) {
   console.log(result);
});
```

Результат выполнения:

```
$ node  waterfall.js
one two
three
done
```

Идея ясна? Ну а теперь вернемся к начальной задаче и перепишем с помощью **async.waterfall** обход папок групп и альбомов:

```
var async = require('async');
var fs = require('fs');
var base = 'bands';async.waterfall([
  function readBands(callback) {
        fs.readdir(base, function(err, bands) {
           callback(err, bands);
    });
],
 function (bands, callback) {
        bands.forEach(function (band) {
            callback (null, band );
    });
},
    function readBandbase(band,callback) {
        fs.readdir(base+'/'+band, function(err, albums) {
```

```
            callback(err,albums, band);
    });
},
    function (albums, band, callback) {
        albums.forEach(function (album) {
            callback (null, album , band);
    });
},
    function readAlbum( album , band, callback) {
        fs.readdir(base+'/'+band+'/'+album, function(err, songs) {
           callback(err,songs, album, band);
        });
    }

],
function(err, songs, album, band){
    console.log(band);
    console.log("    "+album);
    for (var j = 0; j < songs.length; j++) {
        console.log("        "+songs[j]);
    }

});
```

Итак, вывод в консоль производит только одна функция – функция обратного вызова, получающая результат работы всего «водопада», причем названия группы и альбома будут привязаны строго к своим композициям. Следует пояснить два момента. Во-первых, абсолютно все функции «водопада» могут быть анонимны – в примере некоторые из них именованы просто для удобочитаемости. Во-вторых, в цепочке могут участвовать не только асинхронные функции – в этом случае сообщение об ошибке «пробрасывается» от предыдущей асинхронной. Ну а результат такой (в сокращении):

```
$ node wf
Pink Floyd
    1967 - The Piper at the Gates of Dawn
        (01) [Pink Floyd] Astronomy Domine.mp3
        (02) [Pink Floyd] Lucifer Sam.mp3
...

        (11) [Pink Floyd] Bike.mp3
Pink Floyd
    1969 - Ummagumma
        folder.jpg
        Live Album
        Studio Album
Pink Floyd
```

```
    1970 - Atom Heart Mother
        (01) [Pink Floyd] Atom Heart Mother.mp3
...
        (05) [Pink Floyd] Alan's Psychedelic Breakfast.mp3
The Beatles
    1965 - Rubber Soul
        01. Drive My Car.mp3
        02. Norwegian Wood (This Bird Has Flown).mp3
        03. You Won't See Me.mp3
...

        27. If I Needed Someone [1965 Stereo Mix].mp3
        28. Run For Your Life [1965 Stereo Mix].mp3
The Beatles
    Beatles For Sale
        01. No Reply.mp3
        02. I'm A Loser.mp3
        03. Baby's In Black.mp3
...
        14. Everybody's Trying To Be My Baby.mp3
The Beatles
    Help!
        01. Help!.mp3
        02. The Night Before.mp3
        03. You've Got To Hide Your Love Away.mp3
...
        28. Dizzy Miss Lizzy [1965 Stereo Mix].mp3
The Beatles
    Please Please Me
        01. I Saw Her Standing There.mp3
        02. Misery.mp3
        03. Anna.mp3
        04. Chains.mp3
        05. Boys.mp3
        06. Ask Me Why.mp3

        14. Twist And Shout.mp3
The Beatles
    Revolver
        01. Taxman.mp3
        02. Eleanor Rigby.mp3
        03. I'm Only Sleeping.mp3
...
        14. Tomorrow Never Knows.mp3
The Beatles
    With The Beatles
        01. It Won't Be Long.mp3
        02. All I've Got To Do.mp3
        03. All My Loving.mp3
```

```
        14. Money.mp3
Traffic
    1968 - Mr. Fantasy
        01 - Heaven Is In Your Mind.mp3
        02 - Berkshire Poppies.mp3
        03 - House For Everyone.mp3
...
        10 - Giving To You.mp3
Traffic
    1968 - Traffic
        01 - You Can All Join In.mp3
        02 - Pearly Queen.mp3
        03 - Don't Be Sad.mp3
        04 - Who Knows What Tomorrow May Bring.mp3
...
        10 - Means to an End.mp3
```

По-моему, все прекрасно! Нет, я, конечно, не довел дело до конца, то есть не стал читать файлы, теперь-то просто и я оставляю данную задачу как домашнее задание.

В составе модуля asycn есть еще несколько интересных методов, позволяющих управлять потоком исполнения. Призываю не лениться и потратить время на их изучение – оно того стоит. Ну а сейчас мы должны двигаться дальше.

Node.js и данные. Базы данных

На настоящий момент Node.js имеет модули для работы практически со всеми распространёнными СУБД. Собственно, этой фразой можно и ограничиться и дальше не продолжать, но все же есть нюансы работы платформы с различными системами баз данных, которые достойны рассмотрения. Поэтому мы изучим особенности взаимодействия Node.js с наиболее популярными БД и начнем, разумеется, с популярнейшей системы управления данными в веб-программировании.

MySQL и Node.js

Для работы Node.js с mysql существует несколько модулей, но самым распространённым для традиционной модели взаимодействия является node-mysql, его мы и рассмотрим.

Node-mysql – это коннектор, написанный на JavaScript. Он появился одним из самых первых, и именно с его помощью написана большая часть node.js/mysql-приложений. Если у вас уже есть опыт работы с mysql с помощью php или perl, применять его будет просто и привычно. Сначала установка:

```
npm install mysql
```

Это все. За что отдельное спасибо создателям данного модуля – его работа не требует дополнительно устанавливать в систему клиента MySQL.

Теперь попробуем произвести подключение к базе данных (разумеется, mysql-сервер должет быть доступен по заданному url):

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'root',
  password : 'secret',
});
connection.connect();
```

Если запустить данный сценарий, он не завершится сам, поскольку открытое соединение с сервером mysql – достаточный повод не завершать event loop. Поэтому соответствующую команду закрытия соединения **connection.end()** опускать нельзя:

```
connection.connect();
.....................
connection.end();

connection.end(function(err) {
    if (error){
            throw error;
        }
  console.log("Disconect!");
});
```

Этот метод гарантирует, что все активные запросы успеют завершить работу. Есть и более радикальный метод – **destroy()**, он не щадит никого.

Соединению можно задавать и другие параметры, вот некоторые из них:

- **charset** – кодировка, по умолчанию – UTF8_GENERAL_CI';
- **timezone** – временная зона, по умолчанию – 'local';
- **multipleStatements** – разрешает многократное исполнение SQL-инструкции за один запрос. По умолчанию эта возможность отключена – защита от SQL injection;
- **flags** – список флагов соединения (если их набор отличается от принятого по умолчанию).

Запрос к базе данных производится командой **connection.query()**, и в целом взаимодействие с базой данных напоминает, например, работу php-mysql с одним существенным отличием – обработка результатов запроса происходит через функции обратного вызова:

```
 password : 'secret',
));
connection.connect();
connection.query('SELECT Title from band', function(error, rows, fields) {
    if (error){
      throw error;
  }
  // действия с результатами запроса
});

connection.end();
```

Четыре буквы – CRUD

CRUD (create read update delete – «Создание чтение обновление удаление») – термин, введенный Джеймсом Мартином (James Martin), который обозначает сокращённое именование четырех базовых функций при работе с базами данных – создание, чтение, редактирование и удаление этих самых данных. В современных реляционных СУБД этим действиям соответствуют четыре базовые команды языка SQL – INSERT, SELECT, UPDATE и DELETE. Впрочем, тут речь идет не о командах, а именно о действиях, операциях с данными, освоив которые на конкретной СУБД, мы в первом приближении сможем с ней работать (и вписывать в резюме).

Именно поэтому лучший способ получить представление о работе с БД – испытать на ней эти четыре буквы. Чем мы сейчас и займёмся. Начнём с «create»:

```
connection.connect();
var person;
person = 'Ginger Baker';
var query = connection.query("INSERT INTO band SET name = '
cream'", function(err, result) {
    if(err) {
```

```
        console.log('error:', err);
    } else {
        console.log(result);
    }
});
```

Да, мы опять занимаемся английской рок-музыкой, но на отечественный шансон я перейти пока как-то не готов. Результат будет следующим:

```
sukhov@geol:~/node$ node mysql.js
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 5,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0 }
sukhov@geol:~/node$
```

Как видите, мы успешно вставили одну строку в таблицу band.

Preparing Queries

Это, конечно, хорошо, но в наше время, когда злые хакеры и кошмарные SQL-инъекции подстерегают разработчика повсюду, такая прямолинейная работа с базой выглядит, мягко говоря, небезопасной. В СУБД MySQL уже давно появилась реализация prepared statements – подготовка запросов. В модуле node-mysql есть возможность работать с этими конструкциями. Модуль нормально умеет делать Preparing Queries (собственно, те же подготовленные запросы) и работать с Placeholders (заполнителями) – специальными типизированными маркерами, которые пишутся в строке SQL-запроса вместо явных значений:

```
connection.connect();
var person;
person = 'var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'root',
  password : 'secret',
});
var person Eric Clapton';
var query = connection.query("INSERT INTO artist SET name = ?", [person],
function(err, rows) {
    if(err) {
        console.log('error:', err);
    } else {
        console.log(rows);
    }
});
```

Тут все просто: символ «?» представляет собой этот самый «местодержатель» и при формировании запроса подменяется значением переменной person.

Можно все сделать ещё круче:

```
connection.connect();
var person;
person = 'Ginger Baker';
born = '1949-08-19';
var query = connection.query("INSERT INTO ?? SET ?? = ?", [ 'artist', 'name',
person], function(err, rows) {
    if(err) {
        console.log('error:', err);
    } else {
        console.log(rows);
    }
});
```

Чтение, обновление и удаление данных

Теперь освоим букву R – read:

```
 password : 'polarus',
});
connection.connect();
connection.query('SELECT title from band', function(error, rows, fields) {
    if (error){
      throw error;
}
    for(var i = 0; i < rows.length; i++){
      console.log("The band's name is: ", rows[i].Title);
    }
});

connection.end();
```

Результат:

```
$ node mysql.js
The band's name is:  The Beatles
The band's name is:  The Rolling Stones
The band's name is:  Cream
```

Мы видим, что функция обратного вызова получает от выполненного запроса три аргумента – возникшую ошибку (если таковая имеет место быть), массив объектов результата запроса и массив описаний полей из множества полей результата. В нашем случае последний параметр выглядит следующим образом:

```
[ ( catalog: 'def',
    db: 'bands',
    table: 'band',
```

```
    orgTable: 'band',
    name: 'Title',
    orgName: 'Title',
    filler1: <Buffer 0c>,
    charsetNr: 33,
    length: 765,
    type: 253,
    flags: 4097,
    decimals: 0,
    filler2: <Buffer 00 00>,
    default: undefined,
    zeroFill: false,
    protocol41: true } ]
```

В общем, тут масса полезной информации.

Запрос на обновление (буква U от CRUD) ничуть не сложнее:

```
var oldValue = 'cream';
var newValue = 'Cream';

connection.query("UPDATE band SET title = ? WHERE title = ? ", [newValue, oldValue],
                function(error, rows, fields) {
                  if (error){
                        throw error;
                              }
                        console.log(rows);
                    }
);
```

Результат:

```
$ node mysql
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 34,
  warningCount: 0,
  message: '(Rows matched: 1  Changed: 1  Warnings: 0',
  protocol41: true,
  changedRows: 1 }
```

Понятно, что в результате обновлена одна строка.

Теперь последняя буква – D (delete):

```
connection.query("DELETE FROM band WHERE title = ? ", [value], function(error,
rows, fields) {
    if (error){
      throw error;
  }
       console.log(rows);

});
```

Результат:

```
$ node mysql
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 34,
  warningCount: 0,
  message: '',

  protocol41: true,
  changedRows: 0 }
```

Работа с пулом соединений

Еще одной интересной и в условиях Highload-разработки очень востребованной возможностью модуля является работа с пулом соединений с базой данных. Необходимость этого действия я сейчас прояснять не буду (подобный паттерн знаком любому разработчику, работавшему с базами данных на высоконагруженных проектах), а лучше продемонстрирую реализацию:

```
var mysql = require('mysql');
var pool  = mysql.createPool({
  host     : 'localhost',
  user     : 'root',
  password : 'secret',
});
pool.on('connection', function(connection) {
  connection.query('USE bands');
});
```

Тут все просто – вместо одиночного соединения мы сразу создаем пул (как там модуль это реализует – не наше дело, мы действуем на другом уровне). Теперь создаем (извлекаем из пула) соединение:

```
pool.getConnection(function(err, connection) {
  connection.query( 'INSERT INTO INTO band SET name = 'Them' ');
  .......
});
```

И самое главное – если соединение нам больше не нужно (пока, по крайней мере), мы возвращаем его в пул (кому-нибудь другому оно сейчас, возможно, просто необходимо!):

```
 connection.query( 'INSERT INTO sINTO band SET name = 'Them' ',
  function(err, rows) {
```

```
    connection.release();
));
```

Пул соединений принимает все те же параметры, что и простое соединение, добавляя к ним несколько своих:

- **waitForConnections** – определяет действие пула в момент, когда ни одно соединение недоступно (лимит исчерпан). При значении true (по умолчанию) пул будет ожидать свободного соединения, при false – вернет ошибку;
- **ConnectionLimit** – определяет максимальное количество соединений в пуле (по умолчанию – 10);
- **queueLimit** – определяет максимальное число попыток соединений getConnection в соединении. Значение 0 (по умолчанию) обозначает отсутствие ограничений.

ORM-система Sequelize

Что такое ORM и зачем нам это нужно? **ORM** (*Object-relational mapping – объектно-реляционное отображение*) – технология программирования, связывающая базы данных с моделью объектно-ориентированных языков программирования, создавая для элементов СУБД некий виртуальный объектный интерфейс. ORM избавляет программиста от написания большого количества кода, часто однообразного и подверженного ошибкам, тем самым значительно повышая скорость разработки. Кроме того, большинство современных реализаций ORM позволяет программисту при необходимости самому жёстко задать код SQL-запросов, который будет использоваться при тех или иных действиях (сохранение в базу данных, загрузка, поиск и т. д.) с постоянным объектом.

Было разработано довольно много ORM-систем, в основном привязанных к конкретному языку программирования. Для PHP это Propel, Doctrine и Qcodo. Для Python – SQLAlchemy, Storm. Для Java – Hibernate и жуткое количество других решений. JavaScript на этом фоне явно не повезло. По вполне естественным причинам – кто будет заниматься разработкой объектного доступа к СУБД для языка, исполняемого в браузере?

Но, как мы уже знаем, положение изменилось, Javascript уже на сервере и уже активно работает с реляционными базами данных. Разумеется, программисты, не желающие учить SQL, работая на питоне или php, не стали его учить, и переключившись на JavaScript. Так возникло несколько наработок той или иной степени удачности, самой популярной из которых оказалась ORM Sequelize. Ее мы и будем изучать.

Начинаем работать с Sequelize

Начнем с установки соответствующего модуля:

```
npm install  sequelize
```

Теперь устанавливаем соединение с сервером mysql:

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('band', 'root', 'polarus', {
      dialect: "mysql",
      port:    3306,
    });
```

Параметр dialect тут обозначает тип базы данных (еще может быть 'sqlite', 'postgres' и 'mariadb').

Собственно, с полученным экземпляром соединения уже можно работать:

```
sequelize
  .authenticate()
  .complete(function(err) {
    if (!!err) {
      console.log('Ошибка соединения:', err);
    } else {
      console.log('Соединение установлено.');
      sequelize
              .query('SELECT * FROM band', null, { raw: true })
          .success(function(projects) {
              console.log(projects)
            });
    }
  });
```

Результат:

```
$ node  sequelize.js
Executing (default): SELECT 1+1 AS result

Соединение установлено.

Executing (default): SELECT * FROM band

[ { ID: 1, Title: 'The Beatles' },

  { ID: 2, Title: 'The Rolling Stones' },

  { ID: 6, Title: 'Trogs' } ]
```

Но для получения такого результата необходимости в Sequelize нет. Попробуем сделать что-нибудь пограндиознее.

Для использования ORM нам прежде всего надо задать модель – источник данных. В простом случае модель – это таблица базы данных. То есть, создавая модель, мы описываем поля таблицы:

```
var Band = sequelize.define('Bands', {
  id: { type: Sequelize.INTEGER,
primaryKey: true,
     autoIncrement: true },
  title: { type: Sequelize.STRING,
               unique: true },
  state: Sequelize.STRING,
  description: Sequelize.TEXT
});
```

Тут все просто – мы задали наименования полей и их тип. Типы в Sequelize приводятся в соответствии с типами той базы данных, диалект которой используется. Для MySQL соответствия будут следующими:

SequelizeMySQL

Sequelize.STRING	VARCHAR(255)
Sequelize.STRING(1234)	VARCHAR(1234)
Sequelize.STRING.BINARY	VARCHAR BINARY
Sequelize.TEXT	TEXT
Sequelize.INTEGER	INTEGER
Sequelize.BIGINT	BIGINT
Sequelize.FLOAT	FLOAT
Sequelize.DECIMAL	DECIMAL
Sequelize.DATE	DATETIME
BOOLEAN	TINYINT
ENUM	ENUM
Sequelize.BLOB	BLOB
Sequelize.BLOB('tiny')	TINYBLOB

Имеются еще типы данных, не имеющие соответствия в MySQL. Это Sequelize.ARRAY – массивы и Sequelize.UUID – глобальный уникальный идентификатор. Они оба есть в СУБД PostgreSQL. Кроме типов полей, мы указываем необходимые флаги:

```
(primaryKey, autoIncrement, unique), задавать значения по умолчанию, писать комментарии.
```

После описания модели следует ее синхронизировать с реальной базой данных. Мы исходим из того, что таблица Bands на самом деле еще не существует:

```
sequelize
  .sync({ force: true })
  .complete(function(err) {
    if (!!err) {
      console.log( err)
    } else {
      console.log('It worked!')
    }
  })
```

Теперь в базе у нас появилась новая таблица. Если мы посмотрим ее структуру, то там окажутся дополнительные поля, которые мы «не заказывали»:

```
CREATE `Bands` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(255) DEFAULT NULL,
  `state` varchar(255) DEFAULT NULL,
  `description` text,
  `createdAt` datetime NOT NULL,
  `updatedAt` datetime NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `title` (`title`)
)
```

Это самодеятельность Sequelize, простим ей это (кстати, поле id тоже бы создалось, не позаботься мы об этом сами) и вставим в таблицу пару строк.

CRUD на Sequelize

Для этого в полном соответствии с концепцией ORM сначала создаем объект:

```
var band = Band.build({
  title: 'Nirvana',
  state: 'USA',
 description : "американская рок-группа, созданная вокалистом и гитаристом Куртом
Кобейном и басистом Кристом Новоселичем в Абердине, штат Вашингтон, в 1987 году."
})
```

Затем сохраняем его:

```
band.save() .complete(function(err) {
    if (!!err) {
      console.log('The instance has not been saved:', err);
    } else {
      console.log('We have a persisted instance now');
    }
});
```

С помощью специальной команды это можно сделать за один проход:

```
Band.create({
  title: 'Muse',
  state: 'UK',
  description : "британская рок-группа, образованная в 1994 году в городе Тинмут
(графство Девон)."
})
  .complete(function(err, band){
    if(!err){
        console.log('Данные сохранены');
        console.log(band);
    }
  })
```

Тут мы выводим в консоль объект band, только что сохраненный в базе данных. И надо сказать, что этот объект значительно сложнее исходных данных. Вот часть вывода, и в ней можно найти немало интересного:

```
Данные сохранены
{ dataValues:
   { title: 'Muse',
     state: 'UK',
     description: 'британская рок-группа, образованная в 1994 году в городе Тинмут (графство Девон).',
     updatedAt: Tue Sep 09 2014 16:36:32 GMT+0400 (MSK),
     createdAt: Tue Sep 09 2014 16:36:32 GMT+0400 (MSK),
     id: 3 },
  _previousDataValues:
   { title: 'Muse',
     state: 'UK',
     description: 'британская рок-группа, образованная в 1994 году в городе Тинмут (графство Девон).',
     updatedAt: Tue Sep 09 2014 16:36:32 GMT+0400 (MSK),
     createdAt: Tue Sep 09 2014 16:36:32 GMT+0400 (MSK),
     id: 3 },
  __options:
   { timestamps: true,
     createdAt: 'createdAt',
     updatedAt: 'updatedAt',
     deletedAt: 'deletedAt',
     instanceMethods: {},
     classMethods: {},
```

```
    validate: {},

  ..................
    scopes: null,

    hooks: { beforeCreate: [], afterCreate: [] },

    omitNull: false,

    uniqueKeys: {},

    hasPrimaryKeys: true },

  options: { isNewRecord: true, isDirty: true },

  hasPrimaryKeys: true,

  selectedValues:

   { title: 'Muse',

    state: 'UK',

    description: 'британская рок-группа, образованная в 1994 году в городе Тин-
мут (графство Девон).' },

  __eagerlyLoadedAssociations: [],

  isNewRecord: false }

sukhov1@sukhov-System-Product-Name:~/node$
```

На остальных CRUD-операциях я особо останавливаться не буду, просто покажу на примере:

```
// сохраняем еще одну группу
Band.create({
  title: 'Mudhoney',
  state: 'UK',
  description : "американская гранж-группа, сформировавшаяся в 1988 году в городе
Сиэтл вследствие распада гр. Green River. "
})
// выведем все британские группы
Band.findAll({where:{ state: 'UK' }}).complete(function(err, bands) {
  bands.forEach(function(band){
     console.log(band.title);
      });
} );

// Mudhoney
```

```
// Muse

// Непорядок - Mudhoney группа американская.
// исправляем

Band.find({where:{ title: 'Mudhoney' }}).complete(function(err, band) {
     band.state = 'US';
     band.save();
});

// и удаляем группу (ну надо же кого-то удалить?)

Band.find({where:{ title: 'Mudhoney' }}).complete(function(err, band) {
     band.destroy();
});
```

Естественно, приведенный выше листинг, запущенный как один скрипт, корректно работать не будет. Все методы асинхронны, и если действительно необходима подобная последовательность действий, нужно заключить их в функции обратного вызова или воспользоваться механизмами модуля **async** либо сходным средством.

Связи

Теперь посмотрим, как средствами Sequelize можно осуществлять связи между данными в таблицах. Сначала создадим таблицу Artists:

```
var Artist = sequelize.define('Artists', {
  name: { type: Sequelize.STRING,
              unique: true },
  state: Sequelize.STRING
});
```

Теперь заполним ее. Вставлять записи поодиночке утомительно. К счастью, в Sequelize предусмотрен механизм для проведения цепочки действий (запросов). Он так и называется – QueryChainer. Как им пользоваться, понятно из листинга ниже:

```
var chainer = new Sequelize.Utils.QueryChainer

chainer.add(Artist.create({name: 'Kurt Cobain'}));
chainer.add(Artist.create({name: 'Krist Novoselic'}));
chainer.add(Artist.create({name: 'Matthew Bellamy'}));
chainer.add(Artist.create({name: 'Christopher Wolstenholme'}));
chainer.add(Artist.create({name: 'Dominic Howard'}));
chainer
  .run()
  .success(function(){ console.log('Yes!');})
  .error(function(errors){console.log(errors);});
```

После этого обозначаем связь двух таблиц. У нас она будет выражаться отношением «многие ко многим». Да, самый сложный из возможных вариантов:

```
Band.hasMany(Artist)
Artist.hasMany(Band)sequelize.sync()
```

После этого у нас образуется дополнительная таблица, специально предназначенная для реализации many-to-many-связи:

```
CREATE TABLE `ArtistsBands` (

  `createdAt` datetime NOT NULL,

  `updatedAt` datetime NOT NULL,

  `ArtistId` int(11) NOT NULL DEFAULT '0',

  `BandId` int(11) NOT NULL DEFAULT '0',

  PRIMARY KEY (`ArtistId`,`BandId`)

)
```

Теперь связать любимую группу моего сына с ее составом – дело пары строчек кода:

```
Band.find({where:{ title: 'Muse' }}).complete(function(err, band) {
  Artist.findAll({ where: {name :  ['Matthew Bellamy',
                                                    'Christopher Wolstenholme',
                                                    'Dominic Howard']}})
      .success(function(members) {

    members.forEach(function(member){
                    band.setArtists([member]);
                });
      });

} );
```

Пользоваться полученной связью можно так:

```
Band.find({where:{ title: 'Muse' }}).complete(function(err, band) {
  band.getArtists().success(function(artists) {
    artists.forEach(function(artist){
     console.log(artist.name);
      });
  })

} );
```

Результат:

```
$ node mysql

Executing (default): SELECT * FROM `Bands` WHERE `Bands`.`title`='Muse' LIMIT 1;

Executing (default): CREATE TABLE IF NOT EXISTS `ArtistsBands` (`createdAt`
DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, `ArtistId` INTEGER , `BandId`
INTEGER , PRIMARY KEY (`ArtistId`, `BandId`)) ENGINE=InnoDB;

Executing (default): SELECT `Artists`.*, `ArtistsBands`.`createdAt` as
`ArtistsBands.createdAt`, `ArtistsBands`.`updatedAt` as `ArtistsBands.
updatedAt`, `ArtistsBands`.`ArtistId` as `ArtistsBands.ArtistId`,
`ArtistsBands`.`BandId` as `ArtistsBands.BandId` FROM `Artists`, `ArtistsBands`
WHERE `ArtistsBands`.`BandId`=1 AND `ArtistsBands`.`ArtistId`=`Artists`.`id`;

Matthew Bellamy

Christopher Wolstenholme

Dominic Howard
```

Если честно, я не очень люблю ORM-системы. Более того, считаю, что их применимость в высоконагруженных проектах, по крайней мере, под вопросом. И в своем мнении я не одинок. Причем главная претензия к ORM даже не в том, что между приложением и базой данных возникает дополнительная прослойка, а в том, что пользоваться этими инструментами, особенно на сложных запросах, часто не очень удобно. Причем для человека, умеренно знающего SQL, почти всегда удобнее «общаться» с базой данных без посредников. И тем не менее не могу не признать, что Sequelize – решение, в своем классе просто замечательное. Гибкий, понятный и нетяжелый инструмент, который действительно облегчает работу. И ведь о многих возможностях я не рассказал! Впрочем, это вы легко можете исправить, ознакомившись с очень внятной документацией на сайте проекта [9].

NoSQL

Как мы установили, у Node.js нет проблем с использованием реляционных баз данных. А как насчет NoSQL? Тут все очень интересно. Дело в том, что модули для работы с нереляционными хранилищами появились у нее даже раньше. Это совершенно естественно, ведь Node.js – это порождение Highload-эпохи, и именно ей мы обязаны своим широким распространением технологии NoSQL. Кстати, а что вообще такое NoSQL-решения и зачем они нужны?

Прежде всего хочу прояснить два распространённых заблуждения. Во-первых, NoSQL вовсе не означает Not SQL (нет SQL'ю), об отказе от технологии SQL речь не идёт (как бы этого не хотелось той части человечества, которая не осилила конструкции вроде OUTER JOIN). NoSQL расшифровывается как Not only SQL (не только SQL).

Второе заблуждение заключается в том, что NoSQL часто рассматривают как какую-то отдельную новую технологию, между тем это скорее название для группы довольно различных решений, некоторые из которых новыми назвать сложно. Общее у них только одно – это Not only SQL, альтернатива реляционному подходу к хранению данных и манипуляциям над ними.

Реляционные базы данных как способ организации информации всем давно знакомы и привычны. Настолько привычны, что ещё пару лет назад большинству разработчиков просто не приходило в голову использовать какую-либо альтернативу. Все это в полной мере касалось и области веб-программирования, где MySQL, PostgreSQL, Oracle, MSSQL вполне успешно использовались в их реляционной ипостаси.

Но вот настало время того самого Highload, а точнее высоконагруженных, масштабируемых, распределённых веб-приложений, и оказалось, что у SQL-модели есть слабые места, она не совсем универсальна. Появилась потребность в новых технологиях.

Тут имеет смысл начать с такого простого, но чрезвычайно востребованного в мире высоких нагрузок инструмента, который если и можно назвать хранилищем данных, то очень с большой натяжкой. Я говорю сейчас о системе Memcached.

NodeJS и Memcached

В своё время эта система была разработана компанией Danga Interactive для LiveJournal. После этого была выпущена как открытый проект и взята на вооружение множеством сайтов (Wikipedia, YouTube, Facebook и др.).

Memchashed – это высокопроизводительная программа, предназначенная для кэширования объектов в оперативной памяти. Она реализует NoSQL-хранилище, построенное по принципу ключ/значение (давно известному и используемому, например, в хэш-таблицах). Самой сильной стороной этой системы является отказ от использования жесткого диска, применение только оперативной памяти для хранения данных. Это дает просто фантастический прирост производительности. Правда, эта же черта является и её недостатком, накладывая ограничения на род хранимых данных, – надежным хранилищем Memchashed считать нельзя, то есть нельзя доверять ему данные, потеря которых критична. Они просто потеряются при любом сбое или даже переполнении памяти. С другой стороны, есть немало задач, для которых нужна особая надёжность.

Memchashed реализует только базовые функции хранилища данных (собственно, необходимый минимум). Это соединение с сервером, добавление/удаление/обновление объекта, получение значения объекта, а также (что на самом деле довольно важно) инкремент, декремент и ещё пара операций.

Основы Memchashed

Немного подробнее. После соединения с хранилищем клиенту доступны следующие команды:

- **set** – установить ключу объект в качестве значения;
- **add** – добавить кэш-пару ключ/объект (совершенно аналогична set, но выполняется только в том случае, если такого ключа в системе не было);
- **replace** – поменять значение ключа;
- **get** – получить объект из кэша по указанному ключу;
- **delete** – удаляет ключ (естественно, вместе со значением).

Всё это немного напоминает INSERT/UPDATE/DELETE, не так ли?

Дополнительных операций в системе совсем немного, но их значение трудно переоценить:

- **cas** – установить ключу объект в качестве значения, в том случае если оно не было изменено с момента последней операции чтения. Cas (check and set) – это отдаленный аналог транзакций;
- **gets** – делает то же, что и get, но получает дополнительную информацию – версию пары ключ/значение;
- **incr/decr** (инкремент и декремент) – операция увеличивает или уменьшает целочисленное значение ключа на единицу;
- **append/prepend** – добавляет данные в конец или в начало строкового значения ключа.

Наличие этих пар команд открывает перспективы для реализации в Memchached достаточно сложных структур – счетчиков, векторов и т. д.

Физически Memchached реализована в качестве демона (daemon), слушающего TCP-порт (по умолчанию 11211), и работать с системой можно путем непосредственного взаимодействия, но, естественно, это не очень удобно. Да и необходимости такой нет – для большинства языков программирования, используемых в веб-разработке, давно созданы клиентские библиотеки, которые, помимо удобства работы, предоставляют ещё и дополнительную функциональность. Само взаимодействие с Memcached с помощью клиентской программы не сложнее работы с SQL-базой данных, а в большинстве случаев гораздо проще.

Реализация

Начало работы с Memcached на платформе Node.js будет традиционным:

```
$ npm install memcached
```

После инсталляции модуля попробуем установить соединение с memcached-сервером на том же компьютере (естественно, он должен быть установлен и запущен):

```
var Memcached = require('memcached');
var memcached = new Memcached('localhost:11211');
```

Тут мы создаем объект Memcached, в конструкторе которого указываем адрес сервера. Номер порта 11211 является для Memcached параметром по умолчанию. Явным образом уничтожать объект нет необходимости – он живет только во время работы нашей короткой

программы. А вот задать дополнительные опции (это второй аргумент конструктора) можно. Вот некоторые из них:

- **maxKeySize** – максимально допустимый размер ключа (по умолчанию 250 символов);
- **MaxExpiration** – максимальное время жизни ключей (мс);
- **maxValue** – максимальный разрешенный размер значения;
- **poolSize** – максимальное разрешенное число соединений;
- **Reconnect** – интервал между попытками соединиться с сервером, помеченный как «dead» (mc);
- **retries**: – число попыток выделения сокета для данного запроса;
- **failures**: – число неудачных попыток, перед тем как сервер будет помечен «dead»;
- **retry**: – интервал между попытками восстановления работы с сервером после отказа в обслуживании (мс);
- **remove**: – режим, когда сервер отмечен как «dead», он может быть удален из пула, оставшиеся получают его ключи (по умолчанию выключен).

То есть в более сложном случае создание объекта могло бы выглядеть, например, так:

```
var memcached = new Memcached('localhost:11211',
                  {retries:10,
                  retry:10000,
                  remove:true,
                  failOverServers:['192.168.0.101:11212']});
```

Попробуем разместить в Memcached какие-нибудь данные:

```
var Memcached = require('memcached');
var memcached = new Memcached('localhost:11211');
var lifetime = 86400; //24hrs memcached.set('greeting', 'Hello Memcached',
lifetime, function( err, result ){
  if( err ) console.error( err );
  console.dir( result );
});
```

Метод **memcached.set()** соответствует одноименной команде Memcached. Мы задаем ключ greeting, присваиваем ему значение 'Hello Memcached' и назначаем время жизни 24 часа.

Результат:

```
$ node mc.js
true
```

Теперь попробуем извлечь данные:

```
var Memcached = require('memcached');
var memcached = new Memcached('localhost:11211');
```

```
var lifetime = 86400;
  memcached.get('greeting', function( err, result ){
  if( err ) console.error( err );
  console.dir( result );
});
```

Тут мы используем метод **memcached.get()** – также соответствующий одноименной Memcache-команде.

Результат:

```
$ node mc.js
'Hello Memcached'
```

Обратите внимание: мы создаем совершенно новый объект, но данные никуда не делись (если вы, конечно, не перегрузили операционную систему между двумя сценариями). На оперативную память иногда можно положиться, и что особенно интересно – данные в Memcached будут доступны другим процессам системы. Это позволяет использовать подобную технологию и для различных аспектов межпроцессового взаимодействия. Memcached-команда gets тут также реализована:

```
memcached.gets('greeting', function( err, result ){
  if( err ) console.error( err );
  console.dir( result );
});
```

Результат:

```
$ node mc.js
{ greeting: 'Hello World', cas: '45' }
```

Есть и аналог add:

```
memcached.add('greeting', 'Hi Memcached', 86400, function( err, result ){
  if( err ) console.error( err );
  console.dir( result );
});
```

Результат легко предсказуем:

```
$ node mc.js
{ [Error: Item is not stored] notStored: true }
false
```

Нет, ну а что вы хотели? Ключ 'greeting' у нас уже занят.

Создаем приложение

Впрочем, не будем дальше пересказывать руководство. Лучше давайте напишем небольшое приложение и посмотрим, как все работает. Это будет скрипт-голосование, выводящее список музыкальных кол-

лективов (да-да, британских рок групп!) с возможностью отдать свой голос одной из них.

Подготовим данные:

```
memcached.set('The_Beatles', 0, lifetime); memcached.set('The_Rolling_Stones', 0,
lifetime); memcached.set('The_Who', 0, lifetime); memcached.set('The_Trogs', 0,
lifetime);
```

Конечно, использовать названия групп в качестве ключей – практика сомнительная, но сейчас нам нужно сделать все как можно проще (да и потом, у нас же highload и участников голосования будет минимум пара миллионов!).

Теперь само приложение. Сначала вывод:

```
var Memcached = require('memcached');
var memcached = new Memcached('localhost:11211');
var lifetime = 864000;

memcached.getMulti(['The_Beatles',
                    'The_Rolling_Stones',
                    'The_Who',
                    'The_Trogs'],
                   function (err, data) {
                       for (var i in data) {
                             console.log(i+ " " + data[i]);
                       }

                });
```

Мы используем метод **memcached.getMulti()**, позволяющий работать с несколькими ключами.

Результат:

```
sukhov@geol:~/node$ node mc.js
The_Beatles 0
The_Rolling_Stones 0
The_Who 0
The_Trogs 0
```

Теперь добавим возможность ввода данных:

```
    for (var i in data) {
    console.log(i+ " " + data[i]);
   }
    process.stdin.resume();
    process.stdin.setEncoding('utf8');
    console.log('Название:');
          process.stdin.once('data', function(input) {
              input = input.replace(/\r|\n/g,'');
```

```
                memcached.incr(input, 1, function(){

                    process.exit();
                ));
        });

});
```

Теперь запускаем и голосуем:

```
sukhov@geol:~/node$ node mc.js
The_Beatles 0
The_Rolling_Stones 0
The_Who 0
The_Trogs 0
Название:
The_Who
sukhov@geol:~/node$ node mc.js
The_Beatles 0
The_Rolling_Stones 0
The_Who 1
The_Trogs 0
Название:
```

Ну, что можно сказать? Все работает, но программа вышла отвратительной. Причем во всех отношениях – тут тебе и «хардкодинг» (названия групп прямо в тексте программы), и неудобство для пользователя (вписывать в консоль название любимой группы? Я не случайно выбрал самую короткую!).

Нужно немного усложнить наш проект. Как известно, memcached может хранить не только простые данные. В нашем случае это означает, что значениями ключей могут быть не только строки, но и массивы и даже объекты. Объекты! Это именно то, что нам нужно. Попробуем занести начальные данные в memcache в несколько ином виде:

```
var bands = ['The Beatles', 'The Rolling_Stones', 'The Who', 'The Trogs'];
for(var i=0; i < bands.length; i++){
    band = {title:bands[i], vote:0};
    memcached.set(i, band, lifetime, function( err, result ){
       if( err ) console.error( err );
         i++;
    });
}
```

Теперь у нас хранятся не просто названия групп, а объекты вида {'The Beatles', vote:0}. Чтобы вывести список для голосования в консоль, напишем небольшую функцию:

```
}
showList(memcached) ;

function showList(mcd){
    for(j = 0; j < 4; j++){
        mcd.gets(j, function( err, result){
            if( err ) console.error( err );
            key =  this.key;
            console.log(key+" "+result[key].title+" "+result[key].vote);
        });
  }
}
```

Результат будет примерно следующим:

```
$ node  mc.js
0 The Beatles 0
1 The Rolling Stones 0
2 The Who 0
3 The Trogs 0
```

Почему примерно? Мы не предусмотрели тут средств контроля асинхронности, и порядок вывода может быть любым, но как с этим бороться, мы знаем и здесь описывать не будем.

Сам процесс голосования теперь тоже потребует некоторой доработки. Теперь мы не можем воспользоваться методом инкремента значений и вынуждены прибавлять голоса к счетчику «вручную». А для обновления значений воспользуемся методом **memcached.replace()**:

```
showList(memcached);

process.stdin.resume();
process.stdin.setEncoding('utf8');
console.log('Введите номер группы:');
            process.stdin.once('data', function(input) {
                var input = input.replace(/\r|\n/g,'');
                memcached.get(input, function( err, result){
                    var vote = result.vote+1;
                    band = {title:result.title, vote:vote};
                    memcached.replace(input, band, lifetime, function (err) {
                    process.exit();
                   });
                })
});
```

Теперь все более или менее по-человечески:

```
$ node md
Введите номер группы:
0 The Beatles 0
1 The Rolling Stones 0
```

```
2 The Who 0
3 The Trogs 0
2
sukhov@geol-System-Product-Name:~/node$ node md
Введите номер группы:
0 The Beatles 0
1 The Rolling Stones 0
2 The Who 01
3 The Trogs 0
```

(А выбрал я все равно The Who. Ну люблю я их вне зависимости от длины названия.)

MemcacheDB – все-таки DB

Дальнейшим развитием идеи (по мнению злопыхателей, её на корню сгубившим) стало решение все-таки сохранять данные на физическом носителе. Правда, только при необходимости. Воплощением этой концепции стала система, вполне претендующая на звание СУБД – MemcacheDB. MemcacheDB – это система распределенного хранения данных в виде пар ключ/значение, полностью совместимая с memcached API. На практике это обозначает, что клиент системы может работать как с оперативной памятью, так и с физическим носителем для хранения данных, совершенно прозрачно, не делая различий в методах. В качестве бэкэнд-хранилища данных во втором случае выступает BerkeleyDB, а это, в частности, означает, что могут быть задействованы многие возможности, недоступные Memcached, например транзакции, репликация, логгирование и бэкап самих файлов базы данных. Еще MemcacheDB может распределенно работать на нескольких серверах, используя различные стратегии репликации и синхронизации данных. При этом можно с определенной степенью свободы выбирать между гарантированной сохранностью данных и скоростью работы.

А теперь самое главное – MemcacheDB работает по протоколу memcached, и как следствие – модуль **memcached** для Node.js будет с ней работать без каких-либо дополнительных ухищрений:

```
var Memcached = require('memcached');
var memcachedb = new Memcached('127.0.0.1:21201');memcachedb.set("test", "Hello
world", 0, function(){
    memcachedb.get("test", function(err, result){
        if (err) throw err;
        console.log(result);
    });
});
```

В настоящее время MemcacheDB и подобные ей системы большого распространения не получили. И произошло это, вероятно, в силу «промежуточности» положения этих продуктов. С одной стороны, в их реализации утрачена потрясающая простота memcached, с другой – по своему функционалу они всё-таки страшно далеки от традиционных РСУБД. Приложением, которому, пожалуй, максимально удалось сбалансировать требования по функциональности и приспособленности к высоким нагрузкам, стало, уже без всяких оговорок, полноценное хранилище данных – Redis.

Redis – очень полезный овощ

Redis – это key-value-хранилище данных, использующее для хранения оперативную память, по мере необходимости записывая данные на физический носитель. Данные сохраняются на диск при превышении заданного количества запросов и просто через настраиваемый временной интервал. Это означает не что иное, как гарантированную сохранность данных (на что в memcache рассчитывать не приходится).

Redis – что он умеет?

Сохранение данных производится Redis двумя режимами. Штатный режим – snapsotting, предполагает асинхронную запись данных на диск через некоторые промежутки времени. Причем Redis может быть настроен на сохранение данных как через определенный промежуток времени, так и после определенного количества атомарных изменений.

При запуске сервера данные грузятся из дампа данных. При такой схеме в случае сбоя допускается потеря нескольких последних запросов. При безопасном режиме, Append Only File (AOF), каждая команда, изменяющая данные, записывается в специальный файл – ASAP – и заново выполняется при перезапуске сервера, восстанавливая утраченные данные.

Redis обладает такими возможностями и механизмами, как журналирование, снимки, поддержка работы с типами данных, для каждого из которых существует свой набор команд.

Среди других возможностей Redis – транзакции, неблокирующая (!) master-slave репликация на несколько узлов, поддержка использования нескольких БД с возможностью атомарного переноса ключей между ними, Pipeline (отправка на выполнение набора команд и отложенное получение ответа). Работает Redis с данными следующих типов:

- строки (String);
- списки (List);
- множества (Set);
- упорядоченные множества (Sorted Sets);
- хэши (Hash – появились в последней версии).

Теоретически хранилище key-value не должно интересовать сохраненное значение, работа ведется только с ключами, но Redis нарушает этот принцип – для каждого типа содержимого у него своя логика работы и даже свой набор команд. Не красиво? Может быть. Зато очень производительно! Давайте кратко пройдемся по командам и возможностям Redis.

Основы работы с Redis

Опять же, я исхожу из того, что хранилище данных Redis у вас установлено и запущено (если нет – сюда: http://redis.io/download). Запустим консольного клиента redis:

```
sukhov@geol-System-Product-Name:~/node/https/wm$ redis-cli

redis 127.0.0.1:6379>
```

Как видим, все работает (порт 16379 используется по умолчанию). Что будем делать? Ну почему бы не построить такую же голосовалку, какую мы делали на Memcached? Попробуем для этого строковой тип данных:

```
redis 127.0.0.1:6379> set bands:The_Beatles '0'
OK

redis 127.0.0.1:6379> set bands:The_Rolling_Stones '0'

OK

redis 127.0.0.1:6379> set bands:The_Who '0'

OK

redis 127.0.0.1:6379> set bands:The_Trogs '0'

OK
```

Тут все как раньше – имена групп служат ключами, двоеточие не имеет никакого особого значения, но разработчиками принято использование его как разделителя.

Получить значение можно командой get():

```
redis 127.0.0.1:6379> get bands:The_Trogs

"0"
```

К строковым значениям, хранящимся в redis, можно выполнять различные запросы. Например, узнавать длину (strlen), извлекать подстроку (getrange). Нас же сейчас интересуют операции инкремен-

та и декремента (incr, incrby, decr и decrby). Строковое значение можно интерпретировать как число:

```
redis 127.0.0.1:6379> set myscalar 10
OK
redis 127.0.0.1:6379> incrby myscalar 5
(integer) 15
redis 127.0.0.1:6379> get myscalar
"15"
redis 127.0.0.1:6379> set myscalar "test"
OK
redis 127.0.0.1:6379> get myscalar
"test"
redis 127.0.0.1:6379> incrby myscalar 5
(error) ERR value is not an integer or out of range
redis 127.0.0.1:6379>
```

При голосовании за группу мы можем просто применять команду **incr** к соответствующему ключу. Но не собираемся же мы допускать голосующим непосредственно вводить команду в базу данных? Пора подключать Node.js.

Модуль Redis для Node.js

Для начала установим соответствующий модуль:

```
npm install redis
```

Работать с ним несложно. Поключение к Redis-северу будет выглядеть следующим образом:

```
var redis = require("redis"),
client = redis.createClient();
client.select(0, function() {
    client.on("error", function (err) {
        console.log("Error " + err);
    });
//...............
    client.quit();

});
```

После создания redis-клиента мы выбираем базу данных 0 (так в redis базы данных определяются. Хотя в данном случае это лишний шаг – база 0 используется по умолчанию). Теперь можно записывать и получать данные:

```
client.on("error", function (err) {
    console.log("Error " + err);
});
client.set("string key", "string val", redis.print);
client.get("string key", function(err, reply) {
    console.log(reply);
});
```

Результат:

```
$ node redis.js

Reply: OK

string val
```

Допускается работа со множеством ключей, следующая команда разом приготовит данные для голосования:

```
client.mset("The Beatles", "0",
            "The Rolling Stones", "0",
            "The_Who", "0",
            "The Kinks", "0",
            function (err, res) {
                console.log(result);
});
```

Ну а первый, примитивный скрипт голосования будет очень похож на то, как мы делали это с memcache:

```
var redis = require("redis"),
    client = redis.createClient();
var bands = ['The Beatles',
                'The Rolling Stones',
                'The Who',
                'The Kinks'];
client.on("error", function (err) {
    console.log("Error " + err);
});
bands.forEach(function(item){
     client.get(item,
                function (err, result) {
                  console.log(item+" "+result);
               })

});
```

```
process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.once('data', function(input) {
        input = input.replace(/\r|\n/g,'');
        client.incr(input);
        client.end();
  });
```

В принципе, все то же самое, только чуть-чуть удобнее. И с теми же недостатками. Чтобы его радикально улучшить, посмотрим, какими еще структурами данных оперирует redis.

Хэши (Hashes)

Я надеюсь, читателю знакомо понятие ассоциативного массива в php или хэш-массива в perl? Хотя если нет – ничего страшного. Хэш – это не очень сложная структура. Он во всем похож на обычный массив. Важным отличием является то, что вместо числовых индексов его ключами являются строковые значения, в терминах redis это поля (fields), и они предоставляют дополнительный уровень адресации. Пользуясь этой структурой данных, мы можем представить информацию по нашим группам в следующем виде:

```
{
   name   => 'The Beatles',
   id   => 1,
   vote => 0,
   state =>'uk'
}
```

Для операций с этими данными в redis существуют команды hset и hget (аналоги set и get для скалярных типов):

```
redis 127.0.0.1:6379> hset band:1 name 'The Beatles'

(integer) 1

redis 127.0.0.1:6379> hset band:1 id 1

(integer) 1

redis 127.0.0.1:6379> hset band:1 vote 0

(integer) 1

redis 127.0.0.1:6379> hset band:1 state 'uk'

(integer) 1

redis 127.0.0.1:6379> hget band:1 name
```

```
"The Beatles"

redis 127.0.0.1:6379> hgetall band:1

1) "name"

2) "The Beatles"

3) "id"

4) "1"

5) "vote"

6) "0"

7) "state"

8) "uk"

redis 127.0.0.1:6379>
```

Соответствующие команды есть и для redis-модуля node.js, что дает возможность переписать наше приложение, используя более сложные структуры данных. Сначала занесем остальные данные в redis:

```
client.hmset('bands:1', 'name', 'The Beatles','id', 1, 'vote',0,'state', 'us');
client.hmset('bands:2', 'name', 'The Rolling Stones','id', 2, 'vote',0,'state', 'us');
client.hmset('bands:3', 'name', 'The Who','id', 3, 'vote',0,'state', 'us');
client.hmset('bands:4', 'name', 'The Kinks','id', 4, 'vote',0,'state', 'us');
```

hmset – команда redis, позволяющая заполнить сразу несколько полей.

Теперь само голосование:

```
var redis = require("redis"),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});
for(j=1;j<5;j++){
    client.hgetall("bands:"+j, function (err, band) {
        console.dir(band.id+" "+band.name+"("+band.state+") "+band.vote);
    });

}
process.stdin.resume();
process.stdin.setEncoding('utf8');
    process.stdin.once('data', function(n) {
```

```
        var band = 'bands:'+n.replace(/\r|\n/g,'');
    console.log(band);
    client.hincrby(band, 'vote',1);
    client.quit();
});
```

Тут мы использовали redis-команду hincrby, позволяющую инкрементировать значение отдельного поля хэша. Вроде бы код приложения существенно улучшился, но не надо быть Стивом Макконнеллом, чтобы увидеть недостатки – «захардкодженное» количество групп, «железобетонный» порядок перечисления... Хотелось бы чего-то более гибкого, и тут нам на помощь может прийти следующий тип данных redis – списки (Lists).

Списки позволяют хранить и манипулировать массивами значений для заданного ключа. Работая с этими структурами данных, можно добавлять значения в список, получать первое и последнее значение из списка и манипулировать значениями с заданными индексами. Пример работы со списками, взятый из руководства (чуть изменённый):

```
redis 127.0.0.1:6379> rpush messages "Hello how are you?"

(integer) 1

redis 127.0.0.1:6379>  rpush messages "Fine thanks. I'm having fun with Redis"

(integer) 2

redis 127.0.0.1:6379> rpush messages "I should look into this NOSQL thing ASAP"

(integer) 3

redis 127.0.0.1:6379> rpush messages "O_o"

(integer) 4
redis 127.0.0.1:6379> lrange messages 0 2

1) "Hello how are you?"

2) "Fine thanks. I'm having fun with Redis"

3) "I should look into this NOSQL thing ASAP"

redis 127.0.0.1:6379>
```

Тут сообщения чата последовательно заносятся в список messages, затем выводятся три первых сообщения. Как этот тип данных поможет нам? Список прямо напрашивается на то, чтобы хранить в нем...

хм, именно список групп. Вернее, список ключей, по которым мы сможем извлечь данные по группе из хэша. Создадим этот список:

```
for(j=4;j>0;j--){
    client.lpush("bands", "bands:"+j, redis.print);
}
```

Благодаря параметру redis.print мы можем наблюдать в консоли результат операции:

```
$ node redis.js

Reply: 1

Reply: 2

Reply: 3

Reply: 4
```

Теперь извлечь список групп можно гораздо универсальнее и без «магических чисел»:

```
client.on("error", function (err) {
    console.log("Error " + err);
});
client.llen("bands", function(err,n) {
    client.lrange("bands", 0, n,function(err, range){
       range.forEach(function(item){
           client.hgetall(item, function (err, band) {
              console.dir(band.id+" "+band.name+
              "("+band.state+") "+band.vote);
           });

       })

    });
});
```

Ну что же, у нас уже получилось что-то удобоваримое. Теперь несложно, например, написать интерфейс для добавления новых коллективов и т. д. Этим можно заняться потом. А пока давайте посмотрим, что еще нам может предложить redis.

Множества (Sets)

Следующим (по крайней мере, по порядку изложения в документации redis) типом данных являются множества (sets). Они представляют собой непорядочный набор уникальных значений и предостав-

ляют набор различных специфических операций. Помимо обычных операций добавления или удаления, над ними можно выполнять операции пересечения, объединения, дополнения и т. д. Вот простой пример работы с множествами Redis из руководства:

```
redis 127.0.0.1:6379>  sadd myset 1

(integer) 1

redis 127.0.0.1:6379>  sadd myset 2

(integer) 1

redis 127.0.0.1:6379>  sadd myset 3

(integer) 1

redis 127.0.0.1:6379> smembers myset

1) "1"

2) "2"

3) "3"

redis 127.0.0.1:6379> sismember myset 3

(integer) 1

redis 127.0.0.1:6379> sismember myset 8

(integer) 0

redis 127.0.0.1:6379>
```

Существенная разница со списком состоит в том, что множество – это неупорядоченный набор элементов, в котором их можно добавлять и удалять атомарно. В виде множеств можно, например, представить разных музыкантов, играющих в разных группах:

```
redis 127.0.0.1:6379> sadd band:deep_purple Paice Gillan Glover Morse Lord
Blackmore Hughes Coverdale Bolin Turner

(integer) 9

redis 127.0.0.1:6379> sadd band:rainbow Blackmore Dio Driscoll Gruber Bain Powell
Daisley Glover Bonnet Turner

(integer) 10
```

Теперь посмотрим, играл ли Ян Пейс в Rainbow.

```
redis 127.0.0.1:6379> sismember band:rainbow  Paice

(integer) 0
```

Нет, не играл. А в Deep Purple?

```
redis 127.0.0.1:6379> sismember band:deep_purple Paice

(integer) 1
```

Ну конечно же! А были ли вообще в этих группах общие музыканты?

```
redis 127.0.0.1:6379> sinter band:rainbow band:deep_purple

1) "Blackmore"

2) "Glover"

3) "Turner"
```

Просто замечательная аналитика, правда? Идем дальше.

Упорядоченные множества (Sorted Sets)

Последней и самой мощной структурой данных являются упорядоченные множества. Они похожи на множества, но данные в них имеют счетчики – данные, предоставляющие возможности ранжирования и упорядочивания. То есть каждому члену множества соответствует некий рейтинг, с помощью которого эти члены можно, например, сортировать при выводе (подобно ORDER BY в SQL). Естественно, эта структура так и просится, чтобы ее использовали для хранения данных нашего голосования. Давайте немного модернизируем наше приложение.

Прежде всего перенесем список групп в упорядоченное множество (rangeBands):

```
client.llen("bands", function(err,n) {
  client.lrange("bands", 0, n,function(err, range){
   range.forEach(function(item){
       client.hgetall(item, function (err, band) {
           client.zadd("rangeBands",
                        band.vote,
                             item,
                       redis.print);
       });
       })
    });
});
```

Теперь будем выводить группы в соответствии с рейтингом:

```
client.on("error", function (err) {
    console.log("Error " + err);
});


client.zrevrange("rateBands",0, -1, function(err, range){
    range.forEach(function(item){
     client.hgetall(item, function (err, band) {
            console.dir(band.id+" "+band.name+"("+band.state+") "+band.vote);
     });
    });
});
```

Результат:

```
$ node redis.js

'2 The Rolling Stones(us) 2'

'4 The Kinks(us) 1'

'3 The Who(us) 0'

'1 The Beatles(us) 0'
```

Неплохо. Правда, тут присутствуют одни неочевидные грабли – метод **forEach()** работает асинхронно, и теоретически порядок вывода может быть нарушен. Проблему можно решить, например, используя модуль **async**, но это уже другая история.

Механизм Publish/Subscribe

Особый интерес представляет механизм Publish/Subscribe (Pub/Sub), являющийся воплощением парадигмы публикации/подписки. Это довольно близко к механизму очередей сообщений. Суть в следующем – отправители сообщений (пользователи, системные демоны, триггеры, кто угодно) записывают сообщения в некие специальные каналы. Получатели сообщений имеют (возможно, отложенный по времени) доступ ко всем сообщениям канала вне зависимости от персоналий отправителя. В самом простом виде этот функционал можно продемонстрировать следующим кодом:

```
var redis = require("redis");
var client1 = redis.createClient();
var client2 = redis.createClient();
```

```
var msg_count = 0;

client1.on("subscribe", function (channel, count) {
    client2.publish("channel1", "First message");
    client2.publish("channel1", "Second message");
    client2.publish("channel1", "Last message");
});

client1.on("message", function (channel, message) {
    console.log(msg_count+" client1 channel " + channel + ": " + message);
    msg_count += 1;
 if(message == "Last message"){
     client1.unsubscribe();
        client1.end();
        client2.end();
 }
});

client1.subscribe("channel1");
```

Тут мы создаем двух redis-клиентов, одного из которых подписываем на сообщения на канале «channel1». Прямо на событие подписки устанавливаем обработчик, который заставляет второго клиента отправить в этот канал несколько сообщений. При получении сообщений (обработчик onmessage) первый клиент выводит их содержимое в консоль, а если текст этого сообщения будет «Last message», отписывается от канала. Результат работы:

```
$ node redis

0 client1 channel channel1: First message

1 client1 channel channel1: Second message

2 client1 channel channel1: Last message
```

Впрочем, работа двух клиентов в реальности, если клиент будет просто слушать канал:

```
var redis = require("redis");
var client = redis.createClient();
var msg_count = 0;
 client.on("message", function (channel, message) {
    console.log(msg_count+" channel " + channel + ": " + message);
    msg_count += 1;
});

client.subscribe("belomorcannal", function(){
    console.log("Ready");
});
```

Запустим этот код и в другой консоли запишем в канал несколько сообщений, воспользовавшись непосредственно клиентом redis:

```
redis 127.0.0.1:6379>  publish "belomorcannel" "Hello noder-client"

(integer) 1

redis 127.0.0.1:6379>  publish "belomorcannel" "Message 1"

(integer) 1

redis 127.0.0.1:6379>  publish "belomorcannel" "Message 2"

(integer) 1

redis 127.0.0.1:6379>  publish "belomorcannel" "Message 3"

(integer) 1

redis 127.0.0.1:6379>
```

Цифра 1 в ответе сервера обозначает не что иное, как количество благодарных подписчиков. Теперь смотрим работу нашего клиента:

```
$ node redis

Ready

0 channel belomorcannel: Hello noder-client

1 channel belomorcannel: Message 1

2 channel belomorcannel: Message 2

3 channel belomorcannel: Message 3
```

Все работает!

Механизм Publish/Subscribe, например, позволит нам модифицировать нашу голосовалку, разделив сбор голосов и показ результатов. Пусть голосование проходит на неком веб-сайте, тогда сервер, их собирающий (по запросам вида http://our_site.com?band_id=n), будет выглядеть так:

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

http.createServer(function (req, res) {
```

```
  var requestURL = url.parse(req.url).query;
  var bandID = querystring.parse(requestURL).band_id;
  if(bandID > 0){
      var client = redis.createClient();
      client.publish("channel1", bandID, function(error, result){
     if(result>0){
         res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end('Ваш голос принят\n');
     }
      });
  }
}).listen(80);
```

А демон, получающий сообщения и вносящий изменения в данные, – так:

```
var redis = require("redis");
var client = redis.createClient();

var msg_count = 0;
client.on("message", function (channel, message) {
    console.log( message);
    var band = 'bands:' + message;
    client.hincrby(band, 'vote',1);
    client.zincrby('rangeBands', 1,band);
});

client.subscribe("channel1");
```

Нетрудно убедиться, что подобная связка будет хорошо и надежно работать. Правда, с одним «но» – не в высоконагруженной системе. Ведь в нашем голосовании примут участие не менее нескольких десятков миллионов человек, иначе зачем было вообще все это начинать?

В случае высоких нагрузок неизбежно наступит момент, когда наш демон просто не сможет справиться с огромным числом сообщений, поступающих по подписке. Ситуация эта хорошо изучена, и для ее решения придумана концепция отложенного выполнения. Реализуют эту концепцию в подобных случаях использованием такой структуры данных, как очередь сообщений (Message Queue).

Очередь сообщений с помощью Redis

Что это такое? **Очередь** – это структура данных с правом доступа к элементам «первый пришёл – первый вышел». Добавление элемента возможно лишь в конец очереди, выборка – только из начала очереди, при этом выбранный элемент из очереди удаляется. Для на-

шего случая сервер (серверы! Много серверов!), принимающий данные голосования, будет писать эти данные в очередь, а демон (один демон), разбирающий результаты, будет читать их из очереди по мере возможности и обрабатывать. При этом, разумеется, между фактом отдачи голоса и его учетом будет возникать небольшая задержка, но по логике работы приложения это совершенно допустимо (ну появятся результаты на десять секунд позже? Это страшно? Нет!). Существует множество реализаций очередей: **RabbitMQ** – популярная платформа, реализованная на Erlang, **Apache ActiveMQ**, сервис **Amazon Simple Queue Service** (*Amazon SQS*) и т. д. На данной задаче нам не нужно таких мощных решений, к тому же мы имеем в своем распоряжении нечто не менее мощное – Redis. С его помощью очередь сообщений организовать просто до неприличия. Мы будем использовать тип данных «Список».

Сначала напишем, вернее чуть переделаем, сервер для получения результатов голосования из примера про Pub/Sub:

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

http.createServer(function (req, res) {
  var requestURL = url.parse(req.url).query;
  var bandID = querystring.parse(requestURL).band_id;
  console.log(bandID);
  if(bandID > 0){
      var client = redis.createClient();
      client.rpush("queue", bandID, function(error, result){
     if(result>0){
         res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end('Ваш голос принят\n');
     }
      });
  }
}).listen(8080);
```

Как видите, изменений немного. А вот демон придется переписать чуть радикальнее:

```
var redis = require("redis");
client.on("error", function (err) {
          console.log("Error " + err);
        });
var client = redis.createClient();
(function schedule() {
    setTimeout(function() {
```

```
        client.lpop("queue", function(error, message) {

        console.log( message);
        if(message){
                console.log( message);
              var band = 'bands:' + message;
              client.hincrby(band, 'vote',1);
              client.zincrby('rangeBands',1,band);

       }
     ));

           schedule();
       }, 20);
}());
```

Здесь redis-клиент с интервалом 20 мс достает из списка-очереди идентификатор группы и вносит необходимые изменения в данные. Если сообщения перестанут поступать – ничего страшного, значение null просто будет проигнорировано:

```
$ node redis2

1

3
2
3

4

4

3

3

null

null

null

null
```

Redis – штука замечательная, очень быстрая и подкупающе простая. Но вот хранить в ней сложные структуры не то чтобы совсем невозможно, но довольно затратно. На счастье, в мире NoSQL есть решение для хранения сложных и даже очень сложных структур данных.

MongoDB: JavaScript – он везде!

Самым, пожалуй, естественным хранилищем данных для Node.js-приложений является документоориентированное NoSQL-хранилище MongoDB. Почему? Для тех читателей, у которых встает подобный вопрос, позволю небольшой ликбез по этой базе данных.

MongoDB – документоориентированная система управления базами данных с открытым исходным кодом, написанная на языке C++. Она реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным БД.

При разработке этого продукта авторы исходили из необходимости специализации баз данных, благодаря чему им удалось отойти от принципа «один размер подо всё».

MongoDB, по мнению разработчиков, должна заполнить разрыв между простейшими NoSQL-СУБД, хранящими данные в виде «ключ/значение», и большими реляционными СУБД (со структурными схемами и мощными запросами). Используемая модель данных – документ (в виде JSON, а точнее BSON – Binary JavaScript Object Notation). С ней очень просто работать, она проще управляется (в том числе за счёт применения так называемого «бессхемного стиля» (schemaless style), а внутренняя группировка релевантных данных обеспечивает дополнительный выигрыш в быстродействии.

Вся система MongoDB может представлять не только одну базу данных, находящуюся на одном физическом сервере. Функциональность MongoDB позволяет расположить несколько баз данных на нескольких физических серверах, и эти базы данных смогут легко обмениваться данными и сохранять целостность.

- Естественно, за документоориентированность приходится платить. В MongoDB, в отличие от традиционных СУБД, отсутствует оператор «join». Обычно данные могут быть организованы более денормализованным способом, но на разработчиков ложится дополнительная нагрузка по обеспечению непротиворечивости данных.
- Нет такого понятия, как «транзакция». Атомарность гарантируется только на уровне целого документа, то есть частичного обновления документа произойти не может.

- Отсутствует понятие «изоляции». Любые данные, которые считываются одним клиентом, могут параллельно изменяться другим клиентом.

Это маленькие ограничения, давайте разберемся, ради чего кому-нибудь может понадобиться пойти на них.

Для чего?

Зачем и кому нужно подобное представление информации? Для примера возьмем задачу, которую я тут постоянно подсовываю для примеров, а именно построение приложения справочника по любимым рок-группам.

Казалось бы, в задаче организации данных о дискографиях и составах команд нет ничего сложного. Это три таблицы в БД и все. Но вот беда – эти рок-музыканты – народ, крайне не организованный. Они покидают свои коллективы (иногда в разгар работы над новым альбомом), переходят в другие, иногда играют сразу в двух составах, возвращаются, а бывает – записывают сольные работы (сами переходя в ранг «группы») и (о ужас!) привлекают для этого дела своих коллег, окончательно запутывая ситуацию.

Что касается дискографий, то, что тут творится, вообще трудно вообразить – приглашенные музыканты, совместные альбомы, сборники, записи сейшенов...

Я не говорю, что эту информацию совсем невозможно представить с помощью реляционной модели. Можно. Но это будет достаточно сложная схема базы данных, и запросы к ней будут тоже непростыми, а как следствие довольно затратными (я про производительность, а не про зарплаты программистов, если что). Нет, я и сам когда-то был любителем написать SQL-конструкцию на три экрана и гордиться этим творением, но сейчас, в эпоху высоких нагрузок, мы часто просто не можем так поступать с базой данных. Что будет, если этот наш красивый и сложный запрос будет вызываться хотя бы 100 раз в минуту? А это сейчас совсем не предел.

Чем же нам тут поможет MongoDB? Эта база данных воплощает один из принципов работы хранилища данных высоконагруженных систем – хранение всей предполагаемо запрашиваемой информации в одном месте. На практике это выражается в том, что мы можем создать коллекцию «Bands», каждый экземпляр которой будет представлять собой JavaScript-объекты, в которую можно включить в качестве

массива объектов всех музыкантов, имеющих отношение к данному составу, массив объектов релизов группы и т. д. Таким образом, чтобы получить всю необходимую информацию о группе, нам понадобится ровно один, причем простой запрос.

Правда, тут есть нюансы. Мы можем, например, хранить всю информацию об отдельном музыканте (включая его биографию, послужной список, прочие данные) как полноценный объект «Artist», член массива «Members» в объекте «Band». Другим решением будет создать коллекцию Artists, а в Members хранить только ссылки на соответствующие объекты в ней. Никаких требований к эталонной нормализации и тому подобных принципов тут нет, конечное решение определяют только способы использования данных.

В нашем случае хотелось бы, чтобы при запросе информации о группе нам бы мгновенно выдавался список участников (достаточно просто их имен), а при запросе данных о музыкантах мы также быстро получали бы список коллективов, в котором этот музыкант успел засветиться. Понятно, что искать следы пребывания конкретного участника по всей коллекции групп будет нелучшим решением. Поступим следующим образом: создадим коллекции объектов «Bands» и «Artists». В объектах «Band» будем хранить только э... ФИО музыканта и ссылку на него в виде идентификатора объекта в коллекции Artist. (Надо заметить, что механизма внешних ключей в MongoDB нет, и реализация таких ссылок – задача приложения, а не базы данных. Впрочем, не слишком сложная.) В свою очередь, объекты в коллекции Artists будут содержать аналогичный массив с названиями групп, в составе которых довелось побывать музыканту – члену коллекции.

Основы работы с MongoDB

Хватит рассуждать, приступим. Надеюсь, сама MongoDB же установлена и запущена (если нет, вам сюда: http://docs.mongodb.org/manual/installation/). Воспользуемся консольным клиентом этой базы данных, утилитой mongo:

```
mongo
MongoDB shell version: 2.4.6
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
.....
```

Переходим к базе данных, которую мы собираемся использовать (то, что она пока не создана, не проблема, все появится после записи первых данных):

```
> use bands
switched to db bands
>
```

Теперь заносим данные в коллекцию band (которая также будет создана по мере записи данных):

```
> db.band.insert({name:'Led Zeppelin', bid:'1'});
> db.band.insert({name:'Pink Floyd', bid:'2'});
>
```

Добавим составы:

```
db.band.update({bid:'1'},{$set:{members:[{name: "Jimmy Page", id:"1"},{name:"R
bert Plant",id:"2"},{name:"John Bonham",id:"3"},  {name:"John Paul Jones",id:"4
}]}});

db.band.update({bid:'2'},{$set:{members:[{name: "Syd Barrett", id:"5"},
{name:"Roger Waters",id:"6"},{name:"Nick Mason",id:"7"}, {name:"Richard
Wright",id:"8"}, {name:"David Gilmour",id:"9"}]}});
```

Тут мы видим одно из преимуществ MongoDB – нам не надо следовать какой-нибудь схеме, захотели добавить такой непростой тип данных, как массив объектов, – запросто добавили!

Немного про обновление полей. Мало того, что мы добавляем данные, не декларируя заранее их наличия (не было же у нас предусмотрено поле members?), MongoDB еще имеет в своем арсенале очень гибкие средства для модификации данных. Одно из них – оператор $set – мы только что применили. Он заставляет команду update модифицировать только те ключи, которые ему переданы в выражении. Другие операторы-модификаторы:

- **$unset** – удаляет указанный ключ:

  ```
  db.collection.update({id: 2},{$unset {myKey: 1}})
  ```

- **$inc** – этот оператор увеличивает значение поля на указанную величину:

  ```
  db.collection.update({id: 2},{$inc:{counter: 1}})
  ```

- **$rename** – позволяет переименовывать поля:

  ```
  db.collection.update({id: 2},{$rename:{'oldName': 'newName'}})
  ```

Несколько модификаторов предназначены специально для работы с массивами:

- **$push** – добавляет значение в конец массива:

```
db.band.update({bid:'2'},{$push:{members:{name: "David Gilmor",
id:"7"}}});
```

  (В полном соответствии с идеологией MongoDB, если поле не являлось массивом до этой операции, оно им станет. Впрочем, есть и более деликатный аналог этой команды – **$addToSet**, он отработает, только если поле действительно является массивом, при этом значение проверяется на уникальность.)
- **$pushAll** – этот оператор позволяет добавить в массив несколько значений.
- **$pop** – удаляет последнее (последнее указанное или добавленное) значение.
- **$pull** – удаляет указанное значение.
- **$pullAll** – удаляет несколько указанных значений.

Все это нам еще понадобится, но потом. А пока посмотрим, что у нас сейчас в базе:

```
> db.band.find()
{ "_id" : ObjectId("522374640c5879c03c8239fb"),
  "bid" : "1",
  "members" : [
        {"name" : "Jimmy Page", "id" : "1" },
        {"name" : "Robert Plant", "id" : "2" },
        {"name" : "John Bonham", "id" : "3" },
        {"name" : "John Paul Jones", "id" : "4" }
      ],
  "name" : "Let Zeppelin"
}
{ "_id" : ObjectId("522374c30c5879c03c8239fc"),
  "bid" : "2",
  "members" : [
        {"name" : "Syd Barrett", "id" : "5" },
        {"name" : "Roger Waters", "id" : "6" },
        {"name" : "Nick Mason", "id" : "7" },
        {"name" : "Richard Wright", "id" : "8" },
    ],
  "name" : "Pink Floyd"
}
```

Тут сразу бросается в глаза, что MongoDB сама добавила к каждому члену коллекции дополнительное поле – _id. Это первичный ключ – уникальный идентификатор объекта. Его можно задать и вручную, но если этого не сделано, оно будет создано автоматически. Это даже лучше, так мы гарантированно не ошибемся с уникальностью. А зачем же нам понадобилось вводить свой индекс – поле bid (band id)?

Да просто первичный ключ нужен для своих целей – обеспечения целостности данных и связей между документами. Что касается нашей нумерации, мы можем превратить ее в настоящий индекс, необходимый для эффективного доступа к объектам коллекции:

```
> db.band.ensureIndex({bid :1})
```

Тут мы построили индекс по ключу bid в порядке возрастания. Убедиться в этом можно, просмотрев индексы командой **getIndexes()**:

```
> db.band.getIndexes()
[
  {
    "v" : 1,
    "key" : {
              "_id" : 1
            },
    "ns" : "bands.band",
    "name" : "_id_"
  },
  {
    "v" : 1,
    "key" : {
              "bid" : 1
            },
    "ns" : "bands.band",
    "name" : "bid_1"
  }
]
```

Теперь создадим коллекцию artist:

```
> db.artists.nsert({name:'Jimmy Page', aid:'1'});
> db.artists.insert({name:'Robert Plant', aid:'2'});
> db.artists ({name:'John Bonham', aid:'3'});
> db.artists ({name:'John Paul Jones', aid:'4'});
> db.artists ({name:'Syd Barrett', aid:'5'});
.....................................................
> db.artist.sensureIndex({bid :1})
```

И, как и было обещано, добавим артистам группы:

```
db.artist.update({aid:'1'},{$set:{bands:[{name: 'Led Zeppelin', id:"1"},{name:"Fi
rm"},{name:"Jimmy Page & Robert Plant"}]}});
```

У последних двух групп не хватает id? Да, мы их еще не успели внести в соответствующую коллекцию. Это не страшно и вполне допустимо. Посмотреть на имеющиеся в нашей базе данных коллекции можно следующим образом:

```
> db.artist.find().pretty()
```

Результат:

```
{ "_id" : ObjectId("522378f60c5879c03c8239fe"), "name" : "Robert Plant", "aid" :
"2" }
{ "_id" : ObjectId("522379a00c5879c03c8239ff"), "name" : "John Bonham", "aid" :
"3" }
{ "_id" : ObjectId("52237a310c5879c03c823a00"), "aid" : "4", "name" : "John Paul
Jones" }
{ "_id" : ObjectId("522378410c5879c03c8239fd"), "aid" : "1", "bands" : [
{"name" : "Led Zeppelin", "id" : "1" },{"name" : "Firm"},{"name" : "Jimmy Page &
Robert Plant" } ], "name" : "Jimmy Page"}
```

Оператор find() – это оснвной инструмент получения данных (выборки):

```
> db.artist.find({aid:'2'}).pretty()
{ "_id" : ObjectId("522378f60c5879c03c8239fe"), "name" : "Robert Plant", "aid" :
"2" }
```

Команда pretty() не влияет на логику – она просто выводит результат в удобном для чтения виде; без нее мы бы получили строку, что неудобно при работе со сложными объектами.

Запросы могут содержать логические операторы. Например, так мы можем получить названия групп, где играли либо Джимми Пейэдж, либо Сид Барретт:

```
db.bands.find({ $or: [{name : "Jimmy Page"'}, {name: 'Syd Barrett'}]}б, { name:
1, _id: 0 } )
```

На этом непосредственную работу с базой данных прекратим и вернемся к Node.js.

Native MongoDB

Вообще, как уже упоминалось выше, MongoDB сразу стала рассматриваться как наиболее подходящая система управления данными для Nopde.js-приложений. И различных модулей для взаимодействия с этой БД было написано довольно много. Даже слишком много. Мы будем использовать классический коннектор node-mongodb-native, сейчас именуемый просто mongodb:

```
$ npm install mongodb
```

Напишем небольшой сценарий, устанавливающий соединение с базой данных:

```
var mongodb = require('mongodb');
var host = 'localhost';
```

```
var port = mongodb.Connection.DEFAULT_PORT;
var mongoServer = new mongodb.Server(host, port, {auto_reconnect: true});
var db = new mongodb.Db('Bands', mongoServer, {safe:false});
db.open(function(err, db) {
    if(err!='null'){
        console.log("Hello mongo!");
        db.close();
    }
});
```

В этом сценарии, после после подключения модуля mongodb, мы определяем хост нашего установщика MongoDB и порт по умолчанию (если мы не делали с базой ничего нетрадиционного, будет порт 27017).

Затем мы создаем экземпляр mongoDB-сервера. Третьим аргументом его конструктора служит набор параметров. В нашем случае мы задаем автоматическую попытку переустановить соединение при разрыве.

Далее мы создаем экземпляр объекта базы данных, подключившись к нашей базе Bands (если бы такой базы не существовало, то данный конструктор её бы создал в полном соответствии с mongo-идеологией). После этого на созданный объект можно «навешивать» функции обратного вызова, что мы и сделали, пока просто для проверки установленного соединения.

Теперь попробуем выполнить что-нибудь конструктивное. Например, произвести чтение коллекции:

```
var mongodb = require('mongodb');
var host = 'localhost';
var port = mongodb.Connection.DEFAULT_PORT;
var mongoServer = new mongodb.Server(host, port, {auto_reconnect: true});
var db = new mongodb.Db('bands', mongoServer, {safe:false});
db.open(function(err, con) {
    if(err!='null'){
         var bands = con.collection("band");
         bands.findOne({name:'Let Zeppelin'}, function(err, item) {
            console.log(item);
            con.close();
        })
       // db.close();
    }
});
```

Я думаю, этот код не требует особых пояснений. Закрытие соединения с базой данных перенесено в кэлбэк-функцию с целью предотвращения преждевременного вызова этого события (не будем забывать, что функция работает асинхронно). Итог работы будет следующим:

```
$ node mongo.js
{ _id: 522374640c5879c03c8239fb,
  bid: '1',
  members:
   [ { name: 'Jimmy Page', id: '1' },
     { name: 'Robert Plant', id: '2' },
     { name: 'John Bonham', id: '3' },
     { name: 'John Paul Jones', id: '4' } ],
  name: 'Let Zeppelin' }
```

Результат получен, но что-то он не очень впечатляет. Даже в консоли хотелось бы более человеческого и осмысленного вывода. И тут мы можем воспользоваться той особенностью MongoDB, за которую её любят JavaScript-программисты, – представление данных в формате JSON. Ведь, по сути, в результате запроса мы получили JavaScript-объект, и обращаться мы с ним можем соответствующе. Оформим код для осмысленного вывода в виде функции:

```
var mongodb = require('mongodb');
var host = 'localhost';
var port = mongodb.Connection.DEFAULT_PORT;
var mongoServer = new mongodb.Server(host, port, {auto_reconnect: true});
var db = new mongodb.Db('bands', mongoServer, {safe:false});

function getBands(bandName,connection){
     var bands = connection.collection("band");
     bands.findOne({name:bandName}, function(err, item) {
        console.log("Группа: "+item.name);
          console.log("Состав: ");
          var persons = item.members
          for(i=0;i < 4;i++){
              console.log("     "+persons[i].name);
          }
        connection.close();
    });
}
db.open(function(err, con) {
    if(err!='null'){
         getBands('Let Zeppelin',con);
     } else {
         console.log('Error: '+err);
     }
})
```

Результат:

```
$ node mongo.js
Группа: Let Zeppelin
Состав:
```

```
    Jimmy Page
    Robert Plant
    John Bonham
    John Paul Jones
```

Так ведь гораздо лучше? Напишем еще метод, выводящий список групп:

```
function showBands(connection){
        connection.collection("band",function(error,bands){
        bands.find().toArray( function(error,item){
           // console.log(item);
             for(var i = 0; i < item.length;i++){
                  console.log(item[i].bid+" "+item[i].name);
             }
             connection.close();
         });
    });
}
```

Результат:

```
$ node node C:\Users\Book\mongo.js
1 Let Zeppelin
2 Pink Floyd
```

Эту JSON-ориентированность MongoDB можно использовать и в обратную сторону, организовать ввод данных посредством консоли. Для этого будем использовать процесс стандартного ввода:

```
function writeBands(connection){
    var bands = connection.collection("band");
    console.log("введите название:");
        connection.collection("band",function(error,bands){
            process.stdin.resume();
            process.stdin.setEncoding('utf8');
            console.log('Название:');
            process.stdin.once('data', function(input) {
                input = input.replace(/\r|\n/g,'');
                var newBand ={};
                    if(input != ''){
                    newBand.name = input;
                         console.log('состав');
                         var persons = [];
                         process.stdin.on('data', function(input){
                             input = input.replace(/\r|\n/g,'');
                             if(input != ''){
                                 var person ={}
                                 person.name = input;
                                 persons.push(person);
                             } else {
                                 newBand.members = persons;
                                 bands.insert(newBand);
```

```
                        connection.close();
                        process.exit();
                    }
                })

            } else{
              connection.close();
            }
        });
    });
}
```

Тут все очень просто. Мы создаем новый объект, будущий экземпляр коллекции Bands, а пока самый обычный JavaScript-объект. Посредством **process.stdin()** (в данном случае это, понятно, обозначает ввод с клавиатуры) мы заполняем поле name этого объекта, затем создаем и заполняем массив имен участников группы и перезаписываем его в поле members. После этого просто вставляем полученный объект в коллекцию. Правда, просто?Хотя осталась одна проблема, которую мы, к слову сказать, сами себе и создали. Я говорю о пользовательском идентификаторе объекта коллекции (в данном случае bid). С одной стороны, все просто, нужно лишь узнать последний присвоенный, то есть наибольший, bid. В SQL-базах данных для этого нужно сделать простой агрегатный запрос вида SELECT MAX(id) FROM table. Проблема состоит в том, что для MongoDB понятия агрегатных запросов не существует по определению. Вместо этого следует искать нужные значения вручную – это плата за документоориентированность. Для нахождения наибольшего значения нашего параметра bid в MongoDB буквально следует:

- выбрать все объекты коллекции;
- отсортировать их по полю bid;
- ограничить выборку до первого значения.

В синтаксисе запросов MongoDB данные действия будут выражены следующим образом:

```
db.bands.find({}).sort({bid: -1}).limit(1)
```

Число –1 тут задает порядок сортировки – в данном случае обратный. В нашем приложении это выразится таким JavaScript-кодом:

```
bands.find().sort('bid', -1).toArray(function(error,collect){
    newBand.bid = collect[0].bid + 1;
});
```

Обновлять данные тоже не очень сложно. Изменяем значение bid для группы Pink Floyd:

```
bands.update({name: 'Pink Floid'},
             {$set: {aid: '10'}},
             function(err, result){

             })
```

Добавляем в состав Дэвида Гилмора:

```
bands.update({name: 'Pink Floyd'},
             {$push: {members: 'David  Gilmour'}},
             function(err, result){

             })
```

С сожалением увольняем Сида Баррета (наркотики – зло!):

```
bands.update({name: 'Pink Floyd'},
             {$push: {members: 'Syd Barrett'}},
             function(err, result){

             })
```

Вернемся к основному сценарию. Если не считать жутко раздавшегося кода, то теперь все в порядке, но есть небольшая проблема – нам в наследство от предыдущих экспериментов достались объекты, уже занесенные в процессе тестов. От них надо избавиться, и на этом примере мы сейчас освоим четвертую литеру CRUD-аббревиатуры:

```
function clearBands(connection){
        connection.collection("band",function(error,bands){
    bands.find().toArray( function(error,item){
            for(var i = 0; i < item.length;i++){
                if(item[i].bid == undefined){
                   console.log(item[i].name + "will be remove");
                       bands.remove({'name': item[i].name});

                }
           }
             connection.close();
       });
    });
}
```

Рики-тики-тави: Mongoose для MongoDB

Модуль node-mongo хорошо справляется со своей работой, предоставляя простой интерфейс для простых операций. Настолько простой, что иногда просто забываешь, что пишешь на JavaScrip, а не в консоли mongo. Проблема возникает только одна – излишняя сложность кода, в котором часто непросто отследить логику работы при-

ложения. На более или менее сложных проектах сильно не хватает высокого уровня абстракции. В такой ситуации, при использовании реляционной базы данных, выход может быть найден в применении механизма ORM. Для MondoDB как для документированной базы данных предусмотрен сходный инструмент ODM – Object-Document Mapping, объектно-документное отображение). Самая удачная и популярная реализация ODM для MongoDB называется Mongoose. Она традиционно доступна в виде модуля:

```
$ npm install mongoose
```

Основы работы с Mongoose

Начало работы мало отличается от использования нативного расширения:

```
var mongoose = require('mongoose');
var db = mongoose.createConnection('mongodb://localhost/bands');
connection.on('open',function(){
    console.log("Hello Mongoose!");
})
```

Существенное отличие состоит в том, что тут мы получаем экземпляр соединения с базой данных, используя URL, где в качестве протокола будет указано «mongodb», а в качестве последней части адреса – имя базы данных (вполне возможно, еще не существующей).

Полный синтаксис команды **reateConnection()** включает еще указание порта, а также имени пользователя базы данных и его пароля, но нам сейчас это не очень важно. Интереснее разобраться с некоторыми ключевыми элементами работы Mongoose, например объектом **Schema**.

Schema – это декларативное описание сущности (например, коллекции). Тут мы можем задать все необходимые элементы коллекции, а затем с помощью другого объекта Mongoose Model создать ее экземпляр. Если Schema – это описание сущности, то Model – это и есть та самая сущность. Посмотрим, как это все выглядит на практике.

В примерах для модуля node-mongo мы заранее создали необходимые коллекции, пользуясь консольным клиентом mongo. Теперь попробуем сделать это, используя инструменты Mongoose. Сначала определим схему:

```
connection.on('open',function(){
    // console.log("Hello Mongoose!");
    var BandSchema = new mongoose.Schema( {
        bid: { type: Number, index: true, min: 1 },
        name: { type: String, match: /[a-z ]/},
        state: {type: String, default: "uk"},
```

```
        members:[{aid: Number, name: String}]
    });
})
```

Как видите, можно сразу определить индексы, задать значение по умолчанию, а также задать некоторые граничные условия. Минимальное (а при необходимости и максимальное) численное значение, шаблон для строки – все просто и удобно. Описание сложных типов данных, в данном случае массива объектов имен музыкантов, также затруднений не вызывает. Mongoose поддерживает следующие типы данных:

- String;
- Number;
- Date;
- Buffer;
- Boolean;
- Mixed;
- ObjectId;
- Array.

Тут вроде почти все интуитивно понятно. Ну разве что следует пояснить, что ObjectId – это тот самый _id.

CRUD по-мангустски

Создадим модель по имеющейся схеме:

```
var Band = db.model("Band", BandSchema);
```

На этом этапе важно осознавать, что никакой коллекции еще не создано, для этого необходимо создать и сохранить хотя бы один документ. Как только мы это сделаем, в нашей базе появится коллекция с названием «bands» – именно так, с маленькой буквы и во множественном числе.

Создаем первый документ:

```
var newBand = new Band({ bid: 1,name: "The Clash"});
```

Сохраняем его:

```
newBand.save(function (error, item) {
    if (error){
        console.log(error);
    }else{
        console.log(newBand.name+" saved");
    }
});
```

Теперь для работы с данными нам достаточно вызвать к жизни созданную модель, с которой можно обращаться как с JavaScript-объектом (это, собственно, и есть JavaScript-объект!). Вот так можно сделать выборку:

```
db.on('open',function(){
     console.log("Hello Mongoose!");

     var Band = db.model("Band");
     Band.find(function (err, bands) {
        console.log(bands);
     })
});
```

Результат:

```
$ node mongoose.js
[ { bid: 10,
    name: 'The Clash',
    _id: 52c539a6b3f2a29014000002,
    __v: 0,
    members: [],
    state: 'uk' } ]
```

А таким образом мы можем обновить запись – добавить состав группы:

```
db.on('open',function(){
    var persons = [
                   { name: "Joe Strummer"},
                   { name: "Mick Jones"},
                   { name: "Paul Simonon"},
                   { name: "Nicky \"Topper\" Headon"},
                  ];
     var Band = db.model("Band", BandSchema);
     Band.findOne({name: 'The Clash'},function (err, band) {
         band.members = persons;
         band.save();
     })
});
```

А так – удалить документ из коллекции:

```
Band.findOne({name: 'The Clash'},function (err, band) {
        band.remove(function (err, product) {
           Band.findOne({name: 'The Clash'},
                        function (err, product) {
                            console.log(product) // null
                        })
        })
        band.save();
})
```

Возможности mongoose, конечно, не ограничиваются CRUD-операциями. Мы можем расширить схему методами, которые немедленно станут доступны в модели:

```
BandSchema.methods.showBand = function () {
        console.log("Группа: "+this.name);
    console.log("Состав: ");
    var persons = this.members

    for(i=0;i < persons.length;i++){
        console.log(i+"     "+persons[i].name);
    }
}
```

Теперь работаем с моделью:

```
db.on('open',function(){
    var Band = db.model("Band", BandSchema);
    var newBand = new Band({ bid: 11,name: "Sex Pistols"});
    var persons = [
                    { name: "Steve Jones"},
                    { name: "Paul Cook"},
                    { name: "Glen Matlock"},
                    { name: "Johnny Rotten"},
                  ];
    newBand.members = persons;
    newBand.save(function (error, item) {
        if (error){
            console.log(error);
        } else {
            newBand.showBand();
        }
    });
});
```

Результат:

```
$node mongoose.js
Группа: Sex Pistols
Состав:
     Steve Jones
     Paul Cook
     Glen Matlock
     Johnny Rotten
```

Еще существуют статические методы. В данном случае термин «статический» обозначает то, что метод может быть определен для всей модели, вне конкретной записи:

```
BandSchema.statics.findByName = function (name, cb) {
  this.find({ name: new RegExp(name, 'i') }, cb);
}
```

Применение:

```
db.on('open',function(){

    var Band = db.model("Band", BandSchema);
        Band.findByName('The Clash', function (err, band) {
        console.log(band);
    });
});
```

Сеттеры, гетттеры и прочие приятные вещи

Все, кто знаком с ООП, — вероятнее всего, знаком и с акцессор-методами. Для остальных (я что-то сомневаюсь, что они вообще существуют) поясню: это методы доступа к полям и свойствам объекта, непосредственный доступ к которым должен быть контролируем. Прежде всего это set – метод, выполняемый при присваивании значения:

```
BandSchema.path("name").set( function( name ) {
    return name.charAt(0).toUpperCase() + name.slice(1);
});
```

Смысл очень прост. Теперь при добавлении нового члена в коллекцию bands первая буква названия группы будет автоматически переведена в верхний регистр:

```
db.on('open',function(){
    var Band = db.model("Band", BandSchema);
    var newBand = new Band({ bid: 12,name: "buzzcocks"});
    newBand.save(function (error, item) {
        newBand.showBand();
    });
});
```

Результат:

```
$ node mongoose.js
Группа: Buzzcocks
Состав:
```

Того же самого результата можно достигнуть, сразу записав этот сеттер при создании схемы:

```
var BandSchema = new mongoose.Schema( {
        bid: { type: Number, index: true, min: 1 },
        name: { type: String,
                match: /[a-z ]/,
                set: function(name) {
                        if (name == '') {
                            return;
```

```
                    }
                    return name.charAt(0).toUpperCase() +
                        name.slice(1);
                    }
                },
        state: {type: String, default: "uk"},
        members:[{aid: Number, name: String}]
    });
```

Тут мы заодно подстраховались от аварии при пустом значении поля name.

Точно так же можно использовать геттеры:

```
BandSchema.path("members").get( function( members ) {
    if(!members[0]){
        return [{name : "Состав неизвестен"}];
     }
    return members;
});
```

Таким образом, при получении значения поля members пустой массив будет заменяться поясняющей надписью.

Пожалуй, на этом про mongoose все. За бортом нашего краткого обзора остались такие замечательные вещи, как индексы, виртуальные поля, смешанный тип данных и еще несколько интересных возможностей, о которых можно узнать из документации на сайте проекта [8].

Переходим на сторону клиента

Разумеется, Node.js – платформа серверная, но клиентскую сторону веб-разработки мы оставить без внимания никак не можем. Ведь Node.js тесно работает с технологиями «для браузера» – JavaScript-шаблонизаторами и CSS-препроцессорами.

Мыслим шаблонами

Если вы веб-разработчик, думаю, что объяснять, зачем веб-приложению нужен шаблонный движок (Template Engine), будет напрасной тратой времени. Этот механизм используется почти в любом, сколько-нибудь крупном веб-проекте, экономя затраты на верстку и позволяя в той или иной степени отделить код, отвечающий за бизнес-логику, от предоставления HTML-разметки.

В мире веб-разработки придумано великое множество шаблонизаторов для различных технологий. Это Blitz, Smarty для PHP, eRuby для Ruby, Mako для Python, Apache Velocity для Java и C#, Scalate для Scala, CTPP для C, C++, Perl, PHP, Python и т. д. Для JavaScript их создано тоже немало, в последнее время – и для серверной стороны.

Честно говоря, иногда просто хочется протестовать против той легкости, с которой Node/js позволяет создавать расширяющие модули. Это действительно иногда создает проблему. Вот сейчас нам нужен шаблонизатор. Есть ли среди готовых модулей что-нибудь подходящее? Есть. И немало, и выбрать что-то одно довольно непросто. Например, у нас в наличии есть клон известного .NET-шаблонизатора bliss, универсальный javaScript-шаблонизатор dust, «Django-style»-шаблонизатор Swig, Twing – продукт, пришедший из мира php-разработки, и еще много проектов, некоторые из них вполне достойны внимания. Я решил остановиться на модулях, пользующихся наибольшей популярностью. Пусть это и спорный критерий качества, но он, по крайней мере, доступен, и другого у нас нет. Итак, начнем ваять шаблоны.

Mustache – усатый и нелогичный

Шаблонизатор Mustache сами разработчики характеризуют как logic-less, язык шаблонов с минимальным использованием логических конструкций. В этом подходе есть свои преимущества: код шаблонов получается очень простым для понимания и (самое главное) повторного использования.

Простой... ну действительно, Mustache прост прямо по-пролетарски. Вот пример его использования в браузере (как и большинство популярных javaScript-шаблонизаторов, он родился задолго до Node.js):

```
<!DOCTYPE html>
<html>
  <head>
      <title>mustache - test</title>
```

```
        <script type="text/javascript" src="mustache.js"></script>
        <script type="text/javascript" >
         var data = {
            header : "mustache test",
            content: "Строка",
          };
         var template ="<h1>{{header}}</h1><p>{{content}}</p>";
         </script>
  </head>
  <body>
    <script>
     document.write( Mustache.to_html(template, data));
    </script>
  </body>
</html>
```

Результат – на рис. 26.

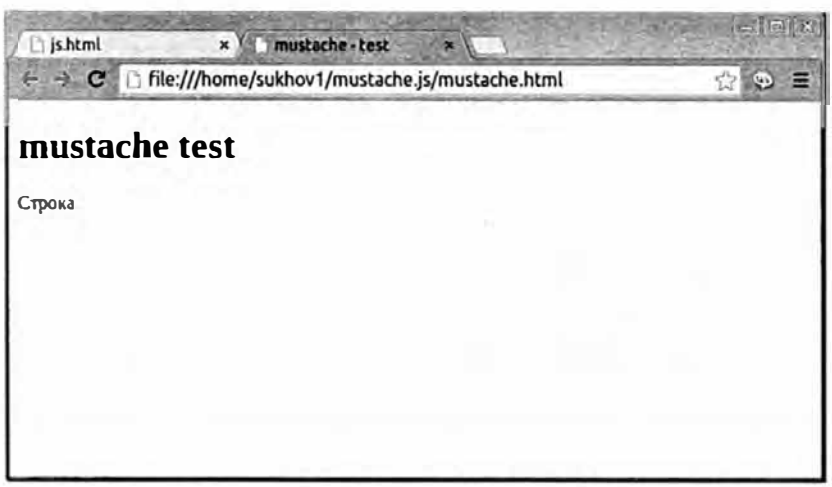

Рис. 26 ❖ Рендеринг mustache-шаблона

Это, конечно, не самая удачная клиент-сайт-конструкция, но она демонстрирует принцип работы Mustache. В данном случае все всем просто.

"<h1>{{header}}</h1> {{content}} " – это шаблон, в двойных фигурных скобках (в «усиках») обозначены имена полей объекта date, которые туда подставляются. Вот пример чуть более сложного шаблона:

```
<h1>{{header}}</h1>
<p>{{{content}}}</p>
```

```
<ul>
{{#authors}}
    <li>{{#accent}} {{.}} {{/accent}}</li>
{{/authors}}
{{^authors}}
    <li>anonymous</li>
{{/authors}}
</ul>
{{#bug}}
{{/bug}}
{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}

{{#empty}}
  <p>The list is empty.</p>
{{/empty}}
```

Всего в *Mustache* используются четыре типа элементов разметки: переменная, секция, комментарий и тег подключения вложенного шаблона.

Переменная просто выводит данные, замещая себя внутри фигурных скобок, причем с экранированием HTML-сущностей – {{header}} или без оного – {{{content}}}. Разница – в количестве скобок.

Элемент (или в терминах *Mustache* – тег) – секция несколько сложнее. Его работа зависит от типа подставляемых данных. Это парный тег – открывающий элемент предваряется диезом и закрывающимся слэшем:

```
{{#person}}... {{/person}}
```

Вот простой пример работы рендеринга секции. Шаблон:

```
{{#person}}
  <b>{{name}}</b> {{years}}</br>
{{/person}}
```

Данные:

```
{
"person": [
    { "name": "Joe Strummer", "years": "1976-86" },
    { "name": "Mick Jones", "years": "1976-83"  },
    { "name": "Paul Simonon", "years": "1976-86"  },
```

```
    { "name": "Nicky \Topper\" Headon ", "years": "1977-82"  },
  ]
}
```

Результат:

```
<b>Joe Strummer</b>1976-86<br>
<b>Mick Jones</b>1976-83<br>
<b>Paul Simonon</b>1976-86<br>
<b>Nicky "Topper" Headon </b>1977-82<br>
```

При обработке простых списков можно использовать постановочный символ «.».

Шаблон:

```
{{#pistols}}
<li>{{.}}</li>
{{/pistols}}
```

Данные:

```
{
  "pistols": ["Steve Jones", "Paul Cook, "Glen Matlock", "Johnny Rotten", "Sid
Vicious"]
}
```

Результат:

```
<li>Steve Jones</li>
<li>Paul Cook</li>
<li>Glen Matlock</li>
<li>Johnny Rotten</li>
<li>Sid Vicious</li>
```

Конструкция вида

```
{{^authors}}
    <li>anonymous</li>
{{/authors}}
```

добавляет к секции значение «по умолчанию», подставляемое в случае нахождения в данных пустого списка.

Если имени секции соответствует функция, то для подстановки будет использован результат ее выполнения. Пример данных:

```
var data ={
 "name": "John",
 "wrapped": function() {
     return function(text) {
       return "<b>" + render(text) + "</b>"
     }
   }
}
```

Шаблон:

```
{{#wrapped}}
  Hello {{name}}!
{{/wrapped}}
```

Результат:

```
<b>Hello John!</b>
```

Последняя конструкция потенциально дает нам инструмент просто фантастической гибкости, но при этом мы теряем самое главное качество Mustache – простоту. Хотя говорят, что для некоторых верстальщиков лямбда-выражения – не помеха.

Еще одна функция секции – работа в качестве условного оператора. Да-да, логика в Mustache хоть и less, но все же присутствует. В таком шаблоне:

```
{{#smocking}}
    Yes!
{{/smocking}}
```

мы увидим результатом рендеринга строчку «Yes», только если данные будут такими:

```
{
  "smocking": true
}
```

Комментарии оформляются в виде тега с восклицательным знаком:

```
<h1>Today{{! ignore me }}.</h1>
```

Результат:

```
<h1>Today.</h1>
```

Подключение дополнительного шаблона вызывается с помощью тега с угловой скобкой. Например, {{>content}}. Если в текущем контексте присутствует поле с таким названием, то оно будет передано в качестве контекста для подключаемого шаблона. Словом, тут тоже нет никаких сложностей:

Базовый шаблон:

```
<p>Users</p>
{{#names}}
  {{> user}}
{{/names}}
```

Дополнительный шаблон:

```
<li>{{name}}</li>
```

Результирующий шаблон:

```
<p>Users</p>
{{#names}}
  <li>{{name}}</li>
{{/names}}
```

Все можно использовать для оформления выводимого контента в браузере, а что насчет Node.js? Да тут все еще проще. Небольшой пример (файл mustache.js):

```
var sys = require('sys');
var mustache  = require("mustache");
var view = {
  name: "John",
  sName: "Lennon"
};
var template = "<p>Person: <p><b>{{name}} {{sName}}</b></p>";
var html = mustache.to_html(template, view);
sys.puts(html);
```

Запускаем и видим результат:

```
$ node mustache.js
<p>Person: <p><b>John Lennon</b></p>
```

Собственно, на этом и закончим знакомство с Mustache. Остается еще добавить, что шаблонизатор в нагрузочных тестах показывает просто замечательную производительность и... а что нам, в сущности, еще надо от такого рода инструмента? Очень много... гибкости, адаптации, понимания DOM-модели HTML. В простых проектах подход Mustache почти идеален, для сложных, многовариантных решений хочется чего-нибудь поинтереснее.

И варианты есть!

EJS – пришелец из мира RoR

Название шаблонизатора EJS расшифровывается как Embedded JavaScript. Это прямой родственник eRuby – Embedded Ruby.

Как следует из названия, EJS, как и Mustache, – это хм... ну, встраивающий шаблонизатор. Как удачнее перевести слово «embedded» в данном контексте, я не знаю. А означает это то, что данные подставляются в шаблон с помощью специальных тегов внутри документа. EJS – довольно простая система шаблонов. Вот пример шаблона EJS:

```
<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) {%>
```

```
    <li><%= link_to(supplies[i], 'supplies/'+supplies[i]) %></li>
<% } %>
</ul>
<%= img_tag('maid.jpg') %>
```

При наличии значений используемых переменных в результате рендеринга этот код будет выглядеть так:

```
<h1>Cleaning supplies</h1>
<ul>

        <li><a href="supplies/mop">mop</a></li>

        <li><a href="supplies/broom">broom</a></li>

        <li><a href="supplies/duster">duster</a></li>

</ul>
<img src="maid.jpg" alt="">
```

Видите, все действительно очень просто и доступно для понимания среднестатистического верстальщика (не говоря уже о существах высшего порядка –программистах).

Синтаксис EJS-шаблонов

Данные подставляются в шаблон в контейнер из скобок вида <% ... %> , при этом может быть два варианта подстановки:

- <%= /* code */ %> – подстановка кода с экранированием HTML;
- <%- /* code */ %> – подстановка кода без экранирования HTML.

EJS полностью поддерживает синтаксис JavaScript, поэтому в шаблонах допустимы любые языковые конструкции – циклы, ветвления, функции обратного вызова:

```
<% if (names.length) { %>
<ul>
<% names.forEach(function(name){ %>
<li><%= name %></li>
<% }) %>
</ul>
<% } %>
```

Помощники (Helpers)

Если вы работали с любым распространенным веб-фреймворком, то концепция хелперов вам должна быть знакома. Это действительно «помощники», берущие на себя рутинные операции в создании пред-

ставлений. В нашем случае (да и в большинстве случаев) речь идет о шаблонах.

Хелпер **link_to()** мы уже использовали в самом первом примере. Он, как нетрудно догадаться, применяется для создания гиперссылки по заданным параметрам. Там же мы применили хелпер **img_tag()**, назначение которого также очевидно.

Несколько полезных хелперов существенно облегчают создание HTML-форм:

```
<%= form_tag('/cleaning/add_supply') %>
        <%= text_field_tag('login') %>
        <%= hidden_field_tag('bid', 5) %>
        <%= submit_tag('Submit') %>
        <%= password_field_tag('pass', 7) %>
        <%= text_area_tag('message', 'Some text') %>
<%= form_tag_end() %>
```

Эта конструкция после рендеринга преобразуется в следующую форму:

```
<form action="/cleaning/add_supply">
  <input id="login" type="text" name="login" value="">
  <input id="bid" type="hidden" name="bid" value="5">
  <input type="submit" value="Submit">
  <input id="pass" type="password" name="pass" value="7">
  <textarea id="message" name="message">
    Some text
  </textarea>
</form>
```

Еще несколько хелперов отвечают за отображение гиперлинков различного назначения, выбора даты времени, просто для отображения произвольных тегов. Этот арсенал очень полезен, но... не для использования в Node.js. К сожалению, вместе с Node применяется ограниченный набор инструкций EJS, но и он вполне эффективен.

EJS и Node.js

Как и Mustache, EJS был разработан для использования на стороне клиента в браузере, но с появлением Node.js был быстро адаптирован для работы на стороне сервера, и теперь Node.js-версия шаблонизатора доступна в виде модуля:

```
$ npm install ejs
```

Использовать EJS можно следующим образом:

```
ejs = require('ejs');
var person = {name: "John"};
```

```
var str = "<% if (user) { %><h2><%= user.name %></h2><% } %>";
var html = ejs.render(str, {user: person});
console.log(html);
```

Результат:

```
$ node ejs
<h2>John</h2>
```

Все работает, но, конечно, логичнее шаблон разместить в отдельном файле. Создадим файл **template.ejs**:

```
<h1><%= title %></h1>
Состав:
<% function member(member){ %>
  <li><strong><%= member.name %></strong> с <%= member.start %>
  <%
    if(member.end) {
       %>по <%= member.end %>
  <% } %></li>
<% } %>

<ul>
  <% members.map(member); %>
</ul>
```

Основной сценарий будет таким:

```
ejs = require('ejs');
fs = require('fs');
var template = "template.ejs";
var str = fs.readFileSync(template, 'utf8');

var title ="The Rolling Stones";
var members = [];

members.push({ name: 'Mick Jagger', start: 1962 });
members.push({ name: 'Keith Richards', start: 1962 });
members.push({ name: 'Charlie Watts', start: 1962  });
members.push({ name: 'Brian Jones', start: 1962, end: 1969 });
members.push({ name: 'Bill Wyman', start: 1962, end: 1993 });
members.push({ name: 'Mick Taylor', start: 1969, end: 1974 });
members.push({ name: 'Ronnie Wood', start: 1974 });

var ret = ejs.render(str, {
  band: title,
  members: members
});

console.log(ret);
```

Результат:

```
$ node ejs
<h1>The Rolling Stones</h1>
```

```
Состав:
<ul>
  <li><strong>Mick Jagger</strong> с 1962  </li>
  <li><strong>Keith Richards</strong> с 1962  </li>
  <li><strong>Charlie Watts</strong> с 1962  </li>
  <li><strong>Brian Jones</strong> с 1962  по 1969</li>
  <li><strong>Bill Wyman</strong> с 1962  по 1993</li>
  <li><strong>Mick Taylor</strong> с 1969  по 1974</li>
  <li><strong>Ronnie Wood</strong> с 1974  </li>
</ul>
```

Так уже лучше, но наиболее эффективное использование шаблонизатора – это отдача браузеру готовой, отрендеренной HTML-странички. Давайте сейчас попробуем реализовать с помощью EJS нечто подобное. Для этого традиционно соорудим небольшой веб-сервер:

```
var ejs = require('ejs');
// var fs = require('fs');
var http = require('http')
var template = "template.ejs";
http.createServer(function (req, res) {
    res.writeHead(200, {'content-type': 'text/html'});
    res.end();
}).listen(8080);
console.log('Server running on 8080');
```

Добавим данные:

```
var http = require('http')
var template = "template.ejs";
    var members = [ "Pete Quaife", "Dave Davies", "Ray Davies",      "Mick
Avory"];
    var band = 'Kinks';
    var title = 'Bands';
```

Теперь создадим шаблон – полноценную html-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <title><%= title %></title>
    </head>
    <body>
        <div>
          <h3>Группа:<%= band %></h3>
          Состав:
          <% if (names.length) { %>
            <ul>
            <% names.forEach(function(name){ %>
              <li><%= name %></li>
            <% }) %>
```

```
            </ul>
<% } %>
</div>
</body>
```

Осталось только распарить шаблон, связав его с данными:

```
http.createServer(function (req, res) {
    res.writeHead(200, {'content-type': 'text/html'});
    ejs.renderFile(template,
                   {title : title, band: band, names : members},
                   function(err, result) {
                       if (!err) {
                           res.end(result);
                       } else {
                           res.end('An error occurred');
                           console.log(err);
                       }
    });
}).listen(8080);
```

Здесь мы применяем еще один метод EJS – **renderFile()**, позволяющий препроводить рендеринг прямо на основе файла шаблона. Теперь, после запуска скрипта, по url http://localhost:8080/ в браузер попадет следующая разметка:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bands</title>
  </head>
  <body>
    <div>
       <h3>Группа:Kinks</h3>
       Состав:
      <ul>
        <li>Pete Quaife</li>
        <li>Dave Davies</li>
        <li>Ray Davies</li>
        <li>Mick Avory</li>
      </ul>
    </div>
</body>
```

Фильтры

Одним из крайне удобных инструментов, являющихся частью шаблонизатора EJS, является механизм фильтров, упрощающих генерацию HTML-разметки. Фильтры в EJS не просто осуществляют

фильтрацию контента, они управляют им, привнося логику. Небольшой пример:

```
ejs = require('ejs');
var members = [
      { name: 'John', born: 1940 },
      { name: 'Paul', born: 1942},
      { name: 'George', born: 1943},
       { name: 'Ringo', born: 1940}
    ];
var str = "<p><%=: members | map:'name' | join:'-' %></p>";
var html = ejs.render(str, {
    members: members
});
console.log(html);
```

Результат будет следующим:

```
$ node ejs.js
<p>John-Paul-George-Ringo</p>
```

Тут мы применили два фильтра. Первый, map, принимает в качестве параметра массив и возвращает новый, применив метод **Array.map**. Второй, join, преобразует массив в строку с заданным разделителем. Результат всех преобразований и подставляется в шаблон.

Что еще можно сделать с помощью фильтров? Ну, например, отсортировать битлов по имени:

```
var str = "<p><%=: members | map:'name'| sort | join:'-' %></p>";
```

Результат:

```
$ node ejs
<p>George-John-Paul-Ringo</p>
```

Тут применен фильтр sort, не требующий особых комментариев. А вот так можно отсортировать исходный массив по полю 'born':

```
var str = "<p><%=: members |sort_by:'born'| map:'name'| join:'-' %></p>";
```

Результат:

```
C:\Users\Geol\node\ejs>node ejs
<p>John-Ringo-Paul-George</p>
```

(То, что Джон младше Ринго, я в курсе, но, к сожалению, фильтры EJS еще не очень умные, а был указан только год рождения.)

Другие фильтры, предназначенные для работы с массивами:

- first – возвращает первый элемент массива;
- last – возвращает последний элемент массива;

- size – возвращает Array.legnth;
- reverse – возвращает реверсированный массив.

Не остался обделенным другой распространенный тип данных – строка:

- **capitalize** – возвращает строку с первым символом в верхнем регистре;
- **downcase** – возвращает строку со всеми символами в нижнем регистре;
- **upcase** – возвращает строку со всеми символами в верхнем регистре;
- **truncate**:n – возвращает подстроку;
- **truncate_words:n** – возвращает результат работы **String.split** и затем **String.splice**;
- **replace:pattern,substitution** – заменяет заданный шаблон (pattern) новым содержимым (substitution);
- **prepend:val** – добавляет подстроку val к началу строки;
- **append:val** – добавляет подстроку val в конец строки.

И это еще не все. Специальные фильтры существуют для чисел:

- **plus:n** – возвращает сумму двух чисел;
- **minus:n** – возвращает разность двух чисел;
- **times:n** – возвращает произведение двух чисел;
- **divided_by:n** – возвращает a/b, –

и объектов:

- **get:'prop'** – возвращает значения свойства объекта.

Фильтры делают EJS очень гибким и удобным для интерпретации данных и отображения контента. А что нам еще нужно от шаблонизатора? Хм... да если подумать, то еще много чего. Например, «понимания» DOM и HTML, гибкой логики, иерархии вложенных шаблонов, работы с CSS... Напридумывать требований можно много, и не все они будут неразумными. Проблема в том, что реализовать эти требования в рамках embedded-шаблонизатора часто довольно трудно. Возможно, тут требуется другой подход.

Jade – нечто нефритовое

Jade – это шаблонизатор, реализованный на Javascript специально для платформы Node.js. Его создатели взяли за основу идеи **Haml** (XHTML Abstraction Markup Language) – компилируемого языка разметки для упрощённой генерации XHTML. Среди его возможностей – отличная читабельность «глазами», работа с блоками,

экранирование XSS-опасных участков кода, поддержка HTML5, кэширование, различные фильтры, комбинирование динамических и статических CSS-классов и многое другое.

Начинаем работу с Jade

Jade – это модуль node.js, поэтому начало будет традиционным:

```
$ npm install jade
```

И чтобы сразу войти в курс дела, соберем небольшой пример:

```
var jade = require('jade');
var jadeString ="p some text"
var html = jade.render(jadeString);
console.log(html)
```

После его запуска в консоли мы получим следующее:

```
$ node jade.js
<p>some text</p>
```

Я думаю, принцип понятен – символ перед текстом превратился в одноименный HTML-тег.

Теперь пример с отдельным файлом-шаблоном. Файл **template.jade**:

```
html
  body
    h1 My Site
    p Welcome to my site!
```

Файл **jade.js**:

```
var jade = require('jade');
var html = jade.renderFile('filename.jade');
console.log(html);
```

Результат:

```
$ node jade.js
<html><body><h1>My Site</h1><p>Welcome to my site</p></body></html>
```

Еще один метод из Jade API, compile, позволяет скомпилировать функцию шаблонизации в переменную. Не понятно? Ну примерно так:

```
var fn = jade.compile('H1 Hi jade!');
console.log(fn.call());
```

Получаем:

```
$ node jade2.js
<H1>Hi jade!</H1>
```

Разумеется, результат работы приложения, использующего jade, обычно не предназначен для вывода в консоль, поэтому соорудим небольшой веб-сервер:

```
var jade = require('jade');
var http = require('http');

var html = jade.renderFile('filename.jade');

http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/html'});
   response.end(html);
}).listen(8080);
```

Шаблон чуть-чуть усложним:

```
doctype 5
html(lang="ru")
  head
    title= 'Stylus page'
    script(type='text/javascript').
      if (foo) {
         bar(1 + 5)
      }
  body
    h1 Jade - шаблонизатор для Node.js
    #container.col
        p Get on it!
        // основной текст
        //- Комментарии только для шаблона
        p.
          Jade - экономный и простой
          язык шаблонов с мощным функционалом.
```

Теперь, после запуска приложения, результат мы можем наблюдать в браузере (рис. 27, http://localhost:8080/). Разметка полученной html-страницы будет следующей:

```
<!DOCTYPE html>
<html lang="ru">
    <head>
        <title></title>
        <script type="text/javascript">
            if (foo) {
               bar(1 + 5)
            }
        </script>
    </head>
    <body>
```

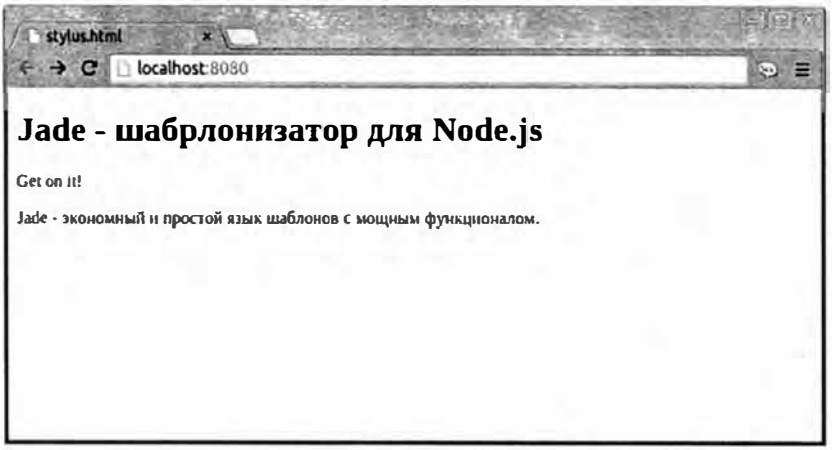

Рис. 27 ❖ Работа Jade

```
    <h1>Jade - шаблонизатор для Node.js</h1>
    <div id="container" class="col">
        <p>Get on it!</p>
        <!-- основной текст -->
        <p>Jade - экономный и простой
язык шаблонов с мощным функционалом.</p>
    </div>
    </body>
</html>
```

И тут уже требуется больше пояснений, хотя в целом все довольно просто.

Основные принципы таковы:

- тег обозначается начальным словом;
- вложенность тегов формируется отступами;
- атрибуты тегов задаются в скобках (разделяются запятыми, при значении undefined или null атрибут *не* добавляется);
- данные, переданные шаблону, вставляются с помощью конструкции #{};
- знак равенства означает: вставить содержимое переменной.

Для атрибутов, которые на самом деле и не атрибуты вовсе, то есть class и id jade, использует, соответственно, символы «.» и «#».

Первой строчкой шаблона задается тип HTML-документа (doctype). Значение 5 соответствует стандарту HTML5 (<!DOCTYPE html>). Другие значения: mobile, basic, xml, default.

Отдельно следует пояснить использование комментариев. Обычные комментарии в стиле JavaScript преобразуются в блочные, как и положено в html. Комментарии, начинающиеся символом //-, принадлежат только шаблону и в результирующую разметку не входят.

Многострочные комментарии задаются отступами:

```
body
  //
    #main
      h1 Закомментированная разметка
      p текст
  h1 Hello
```

Результат:

```
<!--<div id="main">
      <h1>Закомментированная разметка</h1>
      <p>текст</p>
   </div>-->
<h1>Hello</h1>
```

Есть возможность использовать условные комментарии:

```
body
  //if IE
      h3 О Господи, только не IE!
  //- Комментарии только для шаблона
```

даст:

```
<body>
    <!--[if IE]>
      <h3>О Господи, только не IE!</h3>
    <![endif]-->
```

Еще один момент – для тега предоставляется «синтаксический сахар», позволяющий опускать имя тега, что и сделано в нашем шаблоне:

```
    h1 Jade - шаблонизатор для Node.js
    #container.col
        p Get on it!
```

Теперь пример шаблона html-формы:

```
form(mnethod="GET", action="/action").ourForma#33
    input(type="text", value="name")
    input(type="submit", value="OK")
    input(type="checkbox", checked)\
```

Результат:

```
<form mnethod="GET" action="/action" id="33" class="ourForma">
    <input type="text" value="name"/>
```

```
    <input type="submit" value="OK"/>
    <input type="checkbox" checked />
</form>
```

Да, я упомянул про переменные, не продемонстрировав их использования. Сейчас исправлюсь. Но сначала разберемся с передачей данных. Вставим в шаблон строчку:

```
h3 Hello #{name}
```

А в **jade.js** немного... то есть радикально, изменим рендеринг шаблона, используя полный синтаксис, и вообще, давайте чуть перепишем это все, приблизив к нормальному node-приложению:

```
var jade = require('jade');
var http = require('http');
var youAreUsingJade = 1;

http.createServer(functionrequest, response){
    jade.renderFile('filename.jade',
                    { name: 'Саша' },
                    function (err, html) {
                      console.log(err);
                      response.writeHead(200,
                                  {'Content-Type': 'text/html'});
                                response.end(html);
                });


}).listen(8080);
```

Тут мы передаем методу renderFile объект – аргумент и функцию обратного вызова в качестве второго и третьего аргументов.

Результатом подстановки должна стать такая разметка:

```
<h3>Hello Саша</h3>
```

Еще пример, с передачей на рендеринг массива значений (вот тут появляются переменные):

```
girls = ['Нина', 'Зина', 'Наташа'];
http.createServer(function (request, response) {
    jade.renderFile('filename .jade', {
         documents: girls
    },
```

Шаблон:

```
hl Девочки:
ul
  - for (var d in documents)
    li=documents[d]
```

Результат (рис. 28):

```
<h1>Девочки:</h1>
<ul>
    <li>Нина</li>
    <li>Зина</li>
    <li>Наташа</li>
</ul>
```

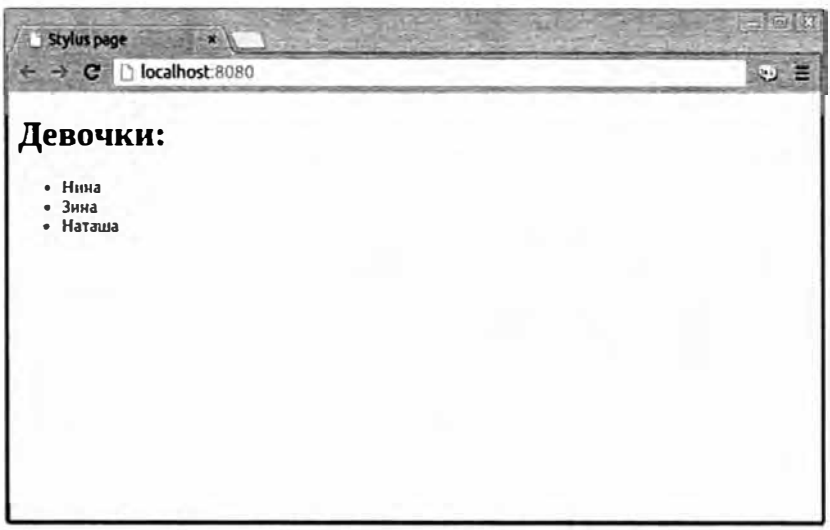

Рис. 28 ❖ Обрабатываем списки

Впрочем, не для того мы поместили парсинг шаблона в кэлбэк вебсервера, чтобы пользоваться «захардкодженными» (то есть имеющими значения, жестко прописанные в коде) переменными. Внесем некоторые изменения в первый пример:

```
var jade = require('jade');
var http = require('http');
var url = require('url');
var youAreUsingJade = 1;
var body = "";

http.createServer(function (request, response) {
    var requestQuery = url.parse(decodeURI(request.url),
                                 true).query;
    jade.renderFile('filename.jade',
                { name: requestQuery.name },
```

```
                function (err, html) {
                  response.writeHead(200,
                              {'Content-Type': 'text/html'});
                  response.end(html);
              });
}).listen(8080);
```

Теперь мы получаем имя из параметров объекта requets, и запрос браузером url вида http://localhost:8080/?name=Маша даст требуемое.

```
<h1>Hello Маша</h1>
```

Еще несколько правил.

Все содержимое тега до следующего тега должно в общем случае укладываться в одну строку. Следующий шаблон даст не совсем желаемый результат:

```
doctype 5
html(lang="en")
    body
        p foo bar foo
          test test test
          test test
          finish.
```

После рендеринга:

```
<!DOCTYPE html>
<html lang="en">
    <body>
       <p>foo bar foo
           <test>test test</test>
           <test>test</test>
           <finish>.</finish>
       </p>
    </body>
</html>
```

На задуманный абзац с небольшим фрагментом текста это мало похоже, правда? Поскольку текст действительно неудобно верстать в одну строчку, разработчиками был предложен выход: если после селектора тега следует символ точки – это указание на то, что последующее содержимое является текстом, и вот в этом случае допускается его размещение в несколько строк:

```
doctype 5
html(lang="en")
    body
        p.
          foo bar foo
```

```
        test
        test test
        finish.
```

Вот теперь все в порядке:

```
<!DOCTYPE html>
<html lang="en">
    <body>
        <p>foo bar foo
              test test test
         test
        finish. </p>
    </body>
</html>
```

Include – собираем шаблон по частям

Как и любой продвинутый шаблонный движок, Jade предоставляет возможность собирать композитный шаблон из разных самостоятельных (возможно, совместно используемых) частей. Проще всего это делается при помощи инструкции include. Например, так:

- **index.jade**:

```
doctype 5
html(lang="en")
    include header
    body
       include footer
```

- **header.jade**:

```
head
    title my jade page
    meta(charset="utf-8")
```

- **footer.jade**:

```
div.footer
    p Copyright(c) my
```

Итог:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>my jade page</title>
        <meta charset="utf-8">
    </head>
    <body>
         <div id="footer">
             <p>Copyright(c) my</p>
        </div>
```

```
    </body>
</html>
```

Важное замечание: подключать с помощью инструкции include можно не только шаблоны jade, но и просто html-файлы, а также вообще файлы любого типа. Нужно только указывать расширение.

Примеси

Примесь (mixin) – это элемент языка программирования (обычно класс или модуль), реализующий какое-либо чётко обозначенное поведение, которое используется для уточнения поведения других классов. В объектно-ориентированных языках программирования является способом реализации классов с измененным поведением, отличным от используемых принципов. Преимуществом примесей является то, что, повышая повторную используемость, этот метод избегает многих проблем множественного наследования.

Ну, это теория. А что делают примеси в шаблонах? Да, в общем, то же самое. Конкретнее лучше показать на примере. В начале шаблона делаем следующую запись:

```
doctype 5
mixin list
  ul
    li первое
    li второе
    li компот
```

Далее в нужном месте шаблона пишем следующий код:

```
  body
    h2 Обед
    mixin list
```

И после рендеринга шаблона получаем такую разметку:

```
<h2>Обед</h2>
    <ul>
        <li>первое</li>
        <li>второе</li>
        <li>компот</li>
    </ul>
```

Это самый простой случай применения примесей. Теперь попробуем соорудить что-нибудь посложнее, использовав подстановку и переменные:

```
mixin newsItem(title, short)
  div.newsItem
    div.newseText
      b #{title}:
      p
        i #{short}
```

Как можно догадаться, это шаблон для вывода списка новостей – заголовка и краткого содержания новости. В теле шаблона новости теперь могут быть добавлены следующим образом:

```
  body
    mixin newsItem('вести с полей', 'Рекордный урожай зерновых в совхозе "Большое
дышло"')
    mixin newsItem('Стабильности нет', 'Террористы опять захватили самолет')
    mixin newsItem('Дружбе - крепнуть', 'Прием в Кремле товарища Зрика Хоннеркера')
```

Результат:

```
<body>
    <div class="newsItem">
        <div class="newseText"><b>вести с полей: </b>
            <p>
                <i>Рекордный урожай зерновых в совхозе
                   &amp;quot;Большое дышло&amp;quot;</i>
            </p>
        </div>
    </div>
    <div class="newsItem">
    <div class="newseText">
.....................................................
```

CSS-препроцессоры – решение проблем стиля

Я думаю, любому веб-разработчику приходилось бороться с CSS (*Cascading Style Sheets* – каскадные таблицы стилей. Ну, вдруг кто не знает). Пусть ты трижды фронт-энд-разработчик, пусть в фирме работает целый штат html-верстальщиков, но время от времени сражения со стилями имеют место быть.

В помощь несчастным разработчикам сейчас созданы различные CSS-фреймворки (Bootstrap, Blueprint и др.), а также расширения CSS (Sass/ LESS/ SCSS), написанные для увеличения уровня абстракции CSS-кода, а значит, и для упрощения разработки. Из всех этих многообразных модернизаций в Node.js-разработке прочные позиции заняли CSS-препроцессоры LESS и Stylus. О них и пойдет речь ниже.

LESS – больше, чем Sass

Первым распространенным CSS-препроцессором стал Sass – Syntactically Awesome Stylesheets (синтаксически умопомрачительные таблицы стилей?).

Sass – это метаязык на основе CSS, разработанный для увеличения уровня абстракции и упрощения синтаксиса. Препроцессор Sass был создан в 2007 году, он написан на языке Ruby и вообще генетически происходил из мира RoR-разработки.

LESS (Leaner CSS) – это CSS-препроцессор/язык, разработанный Алексом Силлером (Alexis Sellier) как развитие идей Sass. Первая версия LESS была написана на Ruby, однако в последующих версиях было решено отказаться от этого языка программирования в пользу JavaScript.

LESS – это честная надстройка над CSS, то есть любые валидные CSS валидным же LESS-кодом (естественно, обратное утверждение не верно). Это его ключевое отличие от Sass – LESS разработан с целью быть как можно ближе к CSS.

LESS привносит в CSS следующие расширения:

- вложенные блоки;
- переменные;
- операторы;
- функции;
- примеси (миксины).

Вложенные блоки

Если честно, это не совсем точный перевод оригинального термина «Nested Rules», но другого у меня нет. Впрочем, не беда – все станет понятно из простого примера:

```
#header {
  h1 {
    font-size: 26px;
    font-weight: bold;
  }
  p { font-size: 12px;
    a { text-decoration: none;
      &:hover { border-width: 1px }
    }
  }
}
```

Результат работы препроцессора следующий:

```
#header h1 {
  font-size: 26px;
  font-weight: bold;
}
#header p {
  font-size: 12px;
}
#header p a {
  text-decoration: none;
}
#header p a:hover {
  border-width: 1px;
}
```

Преимущество такого подхода в том, что при задании стилей вложенных элементов нет необходимости указывать стили для каждого элемента. Еще пример кода nested rules:

```
#header         { color: black;
  .navigation   { font-size: 12px }
  .logo         { width: 300px;
    &:hover     { text-decoration: none }
  }
}
```

Результат:

```
#header {
  color: black;
}
#header .navigation {
  font-size: 12px;
```

```
}
#header .logo {
  width: 300px;
}
#header .logo:hover {
  text-decoration: none;
}
```

Переменные

С переменными все просто, их применение мало отличается от использования переменных в JavaScript или, например, в PHP. Синтаксически они выделяются знаком @ перед названием:

```
@main-color: #5B83AD;
@light-color: (@ main-color + #111);
#header { color: @ main-color; }
#messageBox { color: @light-color; }
```

Этот LESS-код будет оттранслирован препроцессором в следующую CSS:

```
#header {
  color: #5b83ad;
}
#messageBox {
  color: #6c94be;
}
```

Преимущество, например, в том, что изменить цветовую схему сайта теперь можно, поменяв значение одной переменой.

Правила области видимости переменных сходны с переменными php. Если переменная определена в начале LESS-файла, то она будет доступна для любого преследующего кода. Если переменная определена по CSS-правилу, она предназначена для локального использования и не будет доступна вне этого правила. Вот код, демонстрирующий работу с областями видимости переменных в LESS:

```
@var: #CCCCCC;
.class1 {
  @var: #000000;
  .class {
    @var: #FF0000;
    color: @var;
    @var: #0000FF;
  }
  color: @var;
}
.messageBox{ color: @var; }
```

Результат:

```
.class1 {
  color: #000000;
}
.class1 .class {
  color: #0000ff;
}
.messageBox {
  color: #cccccc;
}
```

Для более сложных конструкций есть возможность назначать перемеренные именам переменных. Вот так:

```
@myfont: Helvetica, Arial;
@var: 'myfont';
h1 {
    font-family: @@var;
}
```

Результат:

```
h1 {
  font-family: Helvetica, Arial;
}
```

Операции и функции

Преимущества использования переменных становятся очевидны, если учесть возможность применения к ним арифметических операций. Вот пример:

```
.messageBox{
    @var: 3px;
    border:@var solid #00ff00;
    padding: @var * 3;
    margin: @var + 20;
}
```

Результат:

```
.messageBox {
  border: 3px solid #00ff00;
  padding: 9px;
  margin: 23px;
}
```

Опять же – можно поменять размеры элемента, сохранив все пропорции, поменяв значение одной переменной в одном месте.

Особо интересная возможность – выполнять те же действия со значениями цветов (мы это уже делали в первом примере для переменных):

```
@color: #5B83AD;
div {
  color: @color;
}
div button {
   background: @color + #222222;
   border: 2px solid @color - #111111;
}}
```

Результат:

```
div {
  color: #5b83ad;
}
div button {
  background: #7da5cf;
  border: 2px solid #4a729c;
}
```

Теперь при изменении световой схемы (переменной @color) само дизайнерское решение останется неизменным – светлая кнопка с чуть затемненной границей.

В помощь разработчику в LESS присутствуют встроенные функции, позволяющие экономить код:

```
#header {
  color: @color;
  border-color: desaturate(@color, 10%);
}
```

дает:

```
#header {
  color: #5b83ad;
  border-color: #6783a1;
}
```

Одна из встроенных функций, позволяющих манипулировать цветом на канальном уровне (LESS оперирует HSL – hue, saturation, lightness – цветовой моделью).

Еще пример применения функций:

```
@base: #f04615;
@width: 0.5;

div {
```

```
  width: percentage(0.5); // returns `50%`
  color: saturate(@base, 5%);
  background-color: spin(lighten(@base, 25%), 8);
}
```

Результат работы LESS:

```
div {
  width: 50%;
  color: #f6430f;
  background-color: #f8b38d;
}
```

Таких встроенных функций в LESS довольно много, с их списком можно ознакомиться на официальном сайте проекта (http://lesscss.org/#reference).

Примеси

С этой синтаксической конструкцией мы уже сталкивались, когда говорили о движке шаблонов Jade. Примеси (Mixins) в LESS – правила многоразового использования, которые можно добавить к любому элементу, как правило. Давайте посмотрим, как с помощью этого механизма можно решить проблему, ставшую в последние годы настоящим кошмаром веб-разработчика, – vendor prefix. Если среди читателей есть такие счастливчики, которые еще с ней ни разу не сталкивались, поясню. При современной скорости развития веб-стандартов реализация всяческих нововведений в различных браузерах здорово отличается, и часто, чтобы безопасно применить какое-либо новое свойство (CSS- или DOM-объект), до его окончательной стандартизации, при его использовании к имени добавляется префикс, обозначающий целевой браузерный движок: -webkit, -moz, -o, -ie. Говорить, что свойство bordert-radius – это что-то новое, язык не поворачивается, но вот так еще совсем недавно выглядело его безопасное применение:

```
.messageBox{
   -webkit-border-radius: 5px;
   -moz-border-radius: 5px;
   -o-border-radius: 5px;
}
.articleBox{
   -webkit-border-radius: 5px;
   -moz-border-radius: 5px;
   -o-border-radius: 5px;
}
```

И так каждый раз для каждого закругленного элемента. С использованием примеси этот код можно приписать так:

```
.rounded {
    -webkit-border-radius: 5px;
       -moz-border-radius: 5px;
         -o-border-radius: 5px;
}
.messageBox {
    .rounded;
}

.articleBox {
    .rounded;
}
```

Правда, так удобнее? Что интересно, .rounded – это самый обычный, полноправный CSS-селектор, то есть в качестве примеси можно использовать любой CSS-элемент!

Но это еще не все, в LESS есть такие интересные конструкции, как параметризированные примеси (Parametric Mixins). Модернизируем предыдущий пример:

```
.rounded(@r:5px) {
    -webkit-border-radius: @r;
       -moz-border-radius: @r;
         -o-border-radius: @r;
}
.messageBox {
    .rounded;
}

.articleBox {
    .rounded(3px);
}
```

Результат будет следующим:

```
.messageBox {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  -o-border-radius: 5px;
}
.articleBox {
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  -o-border-radius: 3px;
}
```

С помощью ключевого слова arguments; можно оперировать с целым набором праметров:

```
.myborder(@width: 1px,  @color: #000000){
    border:@arguments;
}
div {
    .myborder(2px);
}
botton {
    .myborder( 1px,#FF0000)
}
```

Результат:

```
div {
  border: 2px #000000;
}
botton {
  border: 1px #ff0000;
}
```

Расширения

Расширения (extend) – это своего рода инверсия примесей. Опять же, это легче пояснить примером:

```
@link-color:        #428bca;
.link {
  color: @link-color;
}
a:extend(.link) {
  font-weight: bold;
}
```

Это будет интерпретировано следующим образом:

```
.link, a  {
  color: #428bca;
}
a {
  font-weight: bold;
}
```

Принцип очень простой – мы не внедряем свойства, а как бы делегируем их.

Работаем с LESS в Node.js

При неиспользовании на стороне клиента LESS-препроцессор просто подключается к веб-странице как обычный Javascript-файл (less.js), и преобразование LESS-кода в CSS будет идти «на лету». Для того чтобы начать работу с LESS на платформе Node.js, нужно сначала установить соответствующий модуль:

```
$ npm install less
```

Теперь проверим его работу простеньким сценарием:

```
var less = require('less');

less.render('.class { width: (1 + 1) }', function (e, css) {
    console.log(css);
});
```

Результат вполне удовлетворителен:

```
$ node less.js
.class {
  width: 2;
}
```

Теперь организуем все более основательно. Сначала создадим файл LESS-шаблона (**style.less**):

```
@link-color: #428bca;
.link {
  color: @link-color;
}
a:extend(.link) {
  font-weight: bold;
}
a {
  &:extend(.link);
  font-weight: bold;
}
```

Сам сценарий (**less.js**):

```
Ivar less = require('less');
fs = require('fs');
var template = "style.less";
var str = fs.readFileSync(template, 'utf8');
var parser = new(less.Parser)({
    paths: ['.', './less'],
    filename: 'style.less'
});

parser.parse(str, function (error, tree) {
    if (error) {
            console.log(error);
        } else {
            var cssMin = tree.toCSS({ compress: true });
            console.log(cssMin);
        }
});
```

И результат его работы:

```
$ node less.js
.link,a,a{color:#428bca}a{font-weight:bold}a{font-weight:bold}
```

Некоторые пояснения. В первом примере мы выполнили рендеринг LESS. Это не всегда хорошо, особенно когда работаешь с шаблонами сложнее, чем в одну строку: если в код вкралась ошибка, вычислить ее местонахождение будет очень непросто. Во втором примере мы сначала создаем объект – less.Parser, затем задаем его свойства (paths – это путь до папки с импортируемыми .less, у нас их нет, но вдруг?).

Методом parser.parse мы разбираем шаблон, отдавая функции обратного вызова объект tree – результат парсинга. Это действительно древообразная структура селекторов и правил примерно такого вида:

```
{ selectors: null,
  rules:
   [ { name: '@link-color',
       value: [Object],
       important: '',
       merge: undefined,
       index: 0,
       currentFileInfo: [Object],
       inline: false,
       variable: true },
     { selectors: [Object],
       rules: [Object],
       _lookups: {},
       strictImports: undefined },
     { selectors: [Object],
       rules: [Object],
       _lookups: {},
       strictImports: undefined },
     { selectors: [Object],
       rules: [Object],
       _lookups: {},
       strictImports: undefined } ],
  _lookups: {},
  strictImports: undefined,
  root: true,
  firstRoot: true,
  toCSS: [Function] }
```

К нему мы применяем метод toCSS, в аргументе которого указываем, что нам нужен минимизированный код (то есть с вырезанными пробелами, табуляциями и переводами строк, это делается для уменьшения «веса» итоговой html-страницы).

На этом про LESS хватит, перейдем к следующему инструменту.

Stylus

Stylus – это препроцессор, созданный под влиянием технических решений Sass и LESS. Он имеет ряд преимуществ, позволяющих органично встраивать его в Node.js-веб-проекты:

- изначально написан на языке javascript;
- обладает простым демократичным синтаксисом, сходным с языком популярного Node.js-шаблонизатора Jade;
- наследование, примеси, функции, операторы, итераторы и прочие функции языка высокого уровня.

Возьмем CSS и отсечем лишнее

На официальном сайте проекта приводится пример самой банальной CSS-разметки (опять border-radius и опять vendor prefixes):

```
body {
  font: 12px Helvetica, Arial, sans-serif;
}
a.button {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}
```

из которого предлагается постепенно удалить все лишнее. Ну а что здесь лишнее? По мнению создателей Stylus, это:

- фигурные скобки (braces);
- знак точки с запятой в конце CSS-инструкций (colons);
- двоеточия (colons).

Убираем! Получилось следующее:

```
body
  font 12px Helvetica, Arial, sans-serif

a.button
  -webkit-border-radius 5px
  -moz-border-radius 5px
  border-radius 5px
```

Тут все ясно, и это уже готовый Stylus-шаблон. Синтаксис его здорово напоминает Jade. Основные синтаксические элементы тут – отступы, они заменяют все вышеупомянутые «ненужные» пунктуационные знаки. В остальном синтаксис в целом совпадает с обычными CSS, хотя есть и отличия:

```
textarea
input
    background-color #fff
    &:hover
        background-color cyan
```

Этот шаблон преобразуется в следующую CSS:

```
textarea, input {
  background-color: #fff;
}
textarea:hover, input:hover {
  background-color: #0ff;
}
```

Тут селекторы, к которым применяется одинаковый набор стилей, перечисляются построчно, друг за другом. Литерал «&» означает ссылку на родительский элемент (в данном случае на textarea и, в соответствии с предыдущим правилом, на input).

Проблему вложенности (Nested Rules) Stylus, благодаря отсутствию синтаксических излишеств, решает даже изящнее, чем LESS:

```
/*
  This is Nested!
 */
body
  padding 40px
  p
    text-align center
    text-decoration underline
    font-size 40px
    > a
      color blue
      &:hover
        color red
```

Эту конструкцию Stylus преобразует в такой CSS-код:

```
/*
  This is Nested!
 */
body {
  padding: 40px;
}
body p {
  text-align: center;
  text-decoration: underline;
  font-size: 40px;
}
body p > a {
```

```
  color: #00f;
}
body p > a:hover {
  color: #f00;
}
```

Обратите внимание: комментарий остался нетронутым.

И тут примеси

Давайте вернемся к первому примеру и еще над ним поработаем, зачем останавливаться? Вот этот вендор-префикс-безобразие

```
  -webkit-border-radius 5px
  -moz-border-radius 5px
     border-radius 5px
```

при использовании CSS пришлось бы указывать везде, где необходимы закругленные уголки. Stylus предлагает другое решение:

```
border-radius(n)
  -webkit-border-radius n
  -moz-border-radius n
  border-radius n

body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius(5px)
```

Да-да, это опять использование механизма «примесей» (mixin), который мы уже успели увидеть при работе с Jade и LESS.

Строго говоря, это так называемые transparent mixins, термин, который некоторые русскоязычные авторы переводят как «прозрачные миксины». Суть от этого не меняется, это возможность вызвать функцию как обычное CSS-свойство.

Как и в LESS, можно использовать встроенную переменную arguments:

```
border-radius()
  -webkit-border-radius arguments
  -moz-border-radius arguments
  border-radius arguments

body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius 5px
```

И параметры:

```
border-radius(n=5px)
  -webkit-border-radius n
  -moz-border-radius n
  border-radius n

.messagebox
  border-radius()

.button
  border-radius(3px)
```

Результат:

```
.messagebox {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}
.button {
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  border-radius: 3px;
}
```

Функции и переменные

Использование переменных в Stylus элементарно до полного безобразия:

```
fonts = helvetica, arial, sans-serif
body
  font 12px fonts

p.tip
     font 10px fonts
```

Результат, как CSS:

```
body {
  font: 12px helvetica, arial, sans-serif;
}
p.tip {
  font: 10px helvetica, arial, sans-serif;
}
```

В отличие от LESS (и Sass), имена переменных в Stylus как синтаксические не выделяются. Во всем остальном они более чем полноценны, их можно применять даже при определении других переменных:

```
borderWith = 2px
boxBorder = borderWith solid #FFCC00

div
  border boxBorder
```

Результат компиляции:

```
div {
  border: 2px solid #fc0;
}
```

Переменные можно определять прямо при объявлении стилей. Вот такая конструкция:

```
.Box
   width: w = 150px
   height: h = 80px
   margin-left: -(w / 2)
   margin-top: -(h / 2)
```

будет совершенно корректно интерпретирована:

```
.Box {
  width: 150px;
  height: 80px;
  margin-left: -75px;
  margin-top: -40px;
}
```

И это тоже хороший способ вносить изменения в CSS, сохраняя необходимые пропорции. А вот еще один:

```
.Box
   width: 150px
   height: 80px
   margin-left: -(@width / 2)
   margin-top: -(@height / 2)
```

Результат будет таким же, как и в предыдущем примере. Добавляя префикс @ к названию свойства, мы «забираем» его ранее объявленное значение. Удивительная гибкость, не правда ли? Еще большей можно достигнуть, используя переменные совместно с примесями:

```
position()
    position: arguments
    z-index: 1 unless @z-index

.Box1
    z-index: 20
    position: absolute
```

```
.Box2
    position: absolute
```

Результат:

```
.Box1 {
  z-index: 20;
  position: absolute;
}
.Box2 {
  position: absolute;
  z-index: 1;
}
```

Что касается функций, то тут захочется сразу начать с примера:

```
invmargin(n)
  margin (- n)

.messageBox
  invmargin(5px)
```

Результат компиляции будет следующим:

```
body {
  margin: -5px;
}
```

Здорово, правда? Но это не функция, это образец той самой «прозрачной миксины». Я привел данный код для иллюстрации близости этих понятий. Еще раз – Stylus позволяет вызвать функцию как обычное CSS-свойство.

Сами функции тут задаются максимально просто:

```
add(a, b)
  a + b

times(a, b)
  a * b

.messageBox
   width: add(150px, 20)
   height: add(80px,20)
   margin-top: times(@height, 0.1)
```

Результат:

```
.messageBox {
  width: 170px;
  height: 100px;
  margin-top: 10px;
}
```

Ключевое слово return разработчики языка прогнозируемо посчитали лишним, возвращается последняя строчка тела функции. Допустимо несколько возвращаемых значений:

```
margins(n)
  n*15px n*10px n*15px n*10px

.messageBox
   margin: margins(1)
.alertBox
   margin-right: margins(2)[0]
```

Результат:

```
.messageBox {
  margin: 15px 10px 15px 10px;
}
.alertBox {
  margin-right: 30px;
}
```

Можно определять аргументы по умолчанию и указывать в качестве аргумента результат работы другой функции:

```
add(a, b)
  a + b

times(a = 2, b = add(a,4))
  a * b

.messageBox
   margin-top: times(4)
   margin-left: times(4,2)
   margin-right: times()
```

Результат:

```
.messageBox {
  margin-top: 32;
  margin-left: 8;
  margin-right: 12;
}
```

Можно использовать функции с именованными параметрами (то есть с возможностью при вызове функции указывать имена параметров, не озадачиваясь порядком аргументов):

```
subtract(a, b)
  a - b

.messageBox
   margin: subtract(b: 10, a: 25)
```

Результат:

```
.messageBox {
  margin: 15;
}
```

И конечно, в теле функции можно применять условные конструкции:

```
compare(a, b)
  if a > b
    higher
  else if a < b
    lower
  else
    equal
```

Тут, кажется, все ясно без пояснений. Что касается операторов, то с полным их списком (которому могут позавидовать многие языки общего назначения) можно ознакомиться на официальном сайте проекта (http://learnboost.github.io/stylus/docs/operators.html).

А еще в арсенале препроцессора цеплый ряд встроенных функций самого различного назначения – работа с цветами, с численными значениями, с литералами, с массивами, с объектами. Посмотреть весь их список можно здесь: http://learnboost.github.io/stylus/docs/bifs.html.

А еще Stylus – это хэш-массивы, итераторы, работа с url и многое другое. Впрочем, обо всех его возможностях можно узнать из документации [15]. Давайте лучше опробуем Stylus в деле.

Stylus и Node.js

Для начала установим модуль:

```
$ npm install stylus
```

Теперь создадим stylus-шаблон – **my.styl** (.styl – традиционное расширение для stylus-шаблонов):

```
body
  font 12px Arial, sans-serif

div.message_box
  -webkit-border-radius 5px
  -moz-border-radius 5px
     border-radius 5px
```

Далее соберем полигон для stylus:

```
var stylus = require('stylus');
fs = require('fs');
```

```
var template = "my.styl";
var str = fs.readFileSync(template, 'utf8');

stylus.render(str,  function(err, css){
if (err) throw err;
  console.log(css);
});
```

Запускаем сценарий и наблюдаем результат:

```
$ node stylus
body {
  font: 12px Arial, sans-serif;
}
div.message_box {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}
```

Все работает.

Попробуем теперь подсунуть стилусу шаблон с «примесью»:

```
// Define a mixin
border-radius()
  -webkit-border-radius arguments
  -moz-border-radius arguments
  border-radius arguments

body
  padding 40px
  p
    text-align center
    text-decoration underline
    font-size 40px
    background blue
    color white
    border-radius 10
```

Результат:

```
$ node stylus
body {
  padding: 40px;
}
body p {
  text-align: center;
  text-decoration: underline;
  font-size: 40px;
  background: #00f;
  color: #fff;
```

```
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  border-radius: 10px;
}
```

Кстати, некоторые верстальщики используют Node.js и модуль stylus для генерации файлов .css из stylus-шаблонов. Мы не верстальщики, нам этот модуль понадобится для интеграции в будущее веб-приложение, к созданию которого мы скоро приступим. Буквально в следующей части книги.

Поднимаем разработку на новый уровень

Ну вот, мы знаем про Node.js почти все. Умеем обращаться с файлами и процессами, обращаться к базам данных и NoSQL-хранилищам, использовать сторонние модули и писать собственные, работать с вебсокетами и эмитировать события. Что еще нам нужно для эффективной разработки? Да, собственно, ничего больше, если мы, конечно, не собираемся создавать сложные, многокомпонентные, расширяемые проекты. Но вот беда – мы собираемся. Нет? Ну тогда бросайте книгу, не желаю больше с вами разговаривать. Ну а если – да, тогда вперед!

Чего нам не хватает?

Рассмотрим (с некоторыми сращиваниями) код веб-сервера, обеспечивающего работу нашего веб-сайта, созданного во второй главе:

```
http.createServer(function onRequest(request, response) {
  var pathname = url.parse(request.url).path;
  if(pathname =='/'){
     pathname = '/index.html';
  }
var extname = path.extname(pathname);
    fs.readFile(pathname, 'utf8', function(err, data) {
        if (err){
              console.log('Could not find or open file '+pathname +' for
reading\n');
         } else {
              response.writeHead(200, {'Content-Type': mimeType});
              response.end(data);
        }
  })
 }
}).listen(8080);
```

Конечно, этот пример написан только для демонстрации возможностей Node.js, а никак не для использования и не для дальнейшей разработки. Но что конкретно нас не устаивает в этом вполне рабочем примере? Тут можно вспомнить очень многое – отсутствие поддержки сессий, кэширования, возможности работы с cookies, невозможность компрессии и т. д. На самом деле отсутствие этих и других современных возможностей – это всего лишь следствие самой глав-

ной проблемы – наше решение с трудом поддаётся модификации и совсем не масштабируемое. Это неудивительно – оно очень низкоуровневое, и если для определенного класса задач такого подхода достаточно, то современная веб-разработка требует более высокого уровня абстракции.

На наше счастье, на настоящий момент создано уже достаточно средств для ведения разработки на платформе Node.js «по-взрослому», и сейчас мы рассмотрим самые популярные и удачные из этих решений.

Connect – middleware framework для node.js

Что такое middleware? (и зачем все это?)

Честно говоря, ответить на вопрос, вынесенный в заголовок, не очень просто. Нет, назвать программные продукты в той или иной мере относящимися к этому классу решений труда не составляет, но вот дать определение... Да что там определение, просто точный перевод термина вызывает затруднение. Вот то, что я встречал в русскоязычных публикациях: «промежуточное программное обеспечение», «программное обеспечение среднего слоя», «подпрограммное обеспечение», «межплатформенное программное обеспечение». Самым точным мне представляется перевод «связующее программное обеспечение». Он хоть и не дословен, но передает самую суть этого термина, ведь middleware – это программное обеспечение, обеспечивающее связь других программных компонентов.

При этом под «программными компонентами» могут пониматься очень, очень разные вещи. Например, middleware может быть назван слой программного обеспечения для взаимодействия между клиентскими частями программы и базой данных, для взаимодействия системных и прикладных программных компонентов, для взаимодействия прикладного программного обеспечения с сетевыми компонентами.

Я уже вас окончательно запутал? На самом деле не все так сложно, просто не стоит сильно вдаваться в формализм. В качестве middleware может выступить, например, сервисная шина, SOAP-сервер, координирующий работу веб-сервисов, CMS-система – да, в общем, любое ПО, выполняющее задачу связывания. А решать такую задачу приходится при построении любого, хоть сколько-нибудь сложного веб-проекта. И приложения на платформе Node.js – тут не исключение.

Connect – это классический middleware (в значении «связывающий») фреймворк, содержащий в себе множество инструментов, необходимых при разработке веб-приложения. В его арсенале – средства для ведения логов, маршрутизации, работы с сессиями, файлами cookies и еще почти полтора десятка так называемых «связывающих

программ» (bundled middleware – связывающее программное обеспечение, далее СПО), среди которых профайлер, сервер раздачи статического контента, средства для обработки http-запросов и много других нужных инструментов. В полном соответствии с концепцией «связывания» connect легко расширяем сторонними приложениями (только на официальном сайте фреймворка их около сотни).

Connect на практике

Установим модуль connect и приступим к работе:

```
$ npm install connect
```

Сначала соорудим что-нибудь простое, чтобы опробовать работоспособность модуля. Что может быть проще http-сервера на Node.js? Вперед!

```
var connect = require('connect');
var http = require('http');

var app = connect()
    .use(function(req, res, next) {
        console.log("И раз!");
        next();
    })
    .use(function(req, res, next) {
        console.log("И два!");
        next();
    })
    .use(function(req, res, next) {
        console.log("И три!");
        res.end("hello connect!");
    });

http.createServer(app).listen(3000);
```

Зайдем на созданный сервер браузером. Результат:

```
$ node connect1
И раз!
И два!
И три!
```

Принцип понятен? Отлично! Теперь будем разбираться, не спеша. Попробуем построить что-нибудь, действительно полезное:

```
var http = require('http');
var connect = require('connect');
```

```
var app = connect()
  .use(connect.logger())
  .use(connect.favicon(__dirname + 'public/favicon.ico'))
  .use(connect.static(__dirname + '/public',{ \maxAge: oneDay}))
  .use(function(req, res){
    res.end('hello world\n');
  })

http.createServer(app).listen(3000);
```

Тут после подключения необходимых модулей мы создаем объект app – экземпляр положения, использующего connect. Затем с помощью метода use мы связываем приложение с необходимыми «связывающими программами» и со слушателем connect-запросов. Теперь, запустив эту программу и сделав несколько запросов браузером на запущенный север (url: http://localhost:3000), мы можем наблюдать в консоли следующую картину:

```
$ node connect.js
127.0.0.1 - - [Sun, 05 Jan 2014 20:14:26 GMT] "GET /user.html HTTP/1.1" 200 - "-
" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0"
127.0.0.1 - - [Sun, 05 Jan 2014 20:14:37 GMT] "GET / HTTP/1.1" 200 - "-" "Mozill
a/5.0 (Windows NT 6.1; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0"
127.0.0.1 - - [Sun, 05 Jan 2014 20:15:35 GMT] "GET /user.html HTTP/1.1" 200 - "-
" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0"
```

Это работает программа **connect.logger**, логирующая все запросы и по умолчанию записывающая их в поток ***stdout***. Впрочем, умолчания можно исправить, как и любая связывающая программа, **connect.logger** имеет множество опций, задающих настройки. Вот как, например, можно изменить формат логов:

```
var app = connect()
  .use(connect.logger('dev'))
  .use(connect.favicon(__dirname + 'public/favicon.ico'))
  .use(connect.static(__dirname + '/public, { maxAge: oneDay }
'))
```

Результат:

```
$ node connect.js
GET / 200 12ms
GET / 200 2ms
GET / 200 1ms
GET /user.html 200 1ms
```

Простенькая программа **connect.favicon** устанавливает эту самую favicon для сайта.

Статический сайт одной строчкой (почти)

Программа **connect.static** – это тот самый сервер раздачи статического контента. Работает он очень просто (по крайней мере, в этом примере). Если мы сейчас поместим файл user.html в папку public в директории нашего приложения (как мы помним, предопределённая переменная __dirname указывает на текущую папку), то он будет отдаваться по http-запросу вида http://localhost:3000/index.html. Программа принимает несколько аргументов:

- **maxAgep** – время жизни браузерного кэша в миллисекундах. По умолчанию 0;
- **hidden** – разрешение перемещать скрытые файлы. По умолчанию false;
- **redirect** – перенаправление к перемещению на «/» (корень папки), когда указанный путь является папкой. По умолчанию true;
- **index** – имя индексного файла сайта, по умолчанию 'index.html'.

Сейчас мы можем очень здорово переписать наш HTTP-сервер, созданный во второй главе, правда?

Это не очень сложно. Перенесём все файлы (html-, css-, javaScript-файлы и файлы изображений) в корень папки public/, и... и все! никаких изменений в коде делать не надо. Модуль сам позаботится о mime и относительных путях. Хотя одну деталь в коде все же стоит изменить:

```
.use(connect.static(__dirname + '/public'))
.use(function(req, res){
  res.writeHead(404);
  res.end('hello world\n');
})
```

Тут надо понимать механизм работы цепочки вызовов connect. Сначала модуль static будет пытаться найти запрошенный ресурс в папке public/ (ну или в той, которую вы ему указали). Если таковой найдется – все в порядке, он будет передан в браузер, и работа программы завершится. Если нет – по цепочке будет вызван следующий элемент связывающего ПО, который в данном случае сгенерирует и отправит HTTP-заголовок 404 (страница не найдена). Удобно?

Совершенствуем сайт

Какие еще возможности дает Connect? Множество, и, естественно, все их мы рассматривать не будем, только наиболее значимые. Давайте посмотрим на наш веб-сайт и решим, что ему не хватает. Ох,

немало! Например, механизма сеансов работы (сессий), без которых невозможно нормальное взаимодействие с клиентом, а авторизация и аутентификация теряют всякий смысл.

Собственно, реализовать механизм сессий, задействовать cookie и переменные окружения, имея доступ к http-заголовкам, не очень сложно. Ну, по крайней мере, любому веб-программисту (если вы это читаете, вы же имеете отношение к веб-разработке, правда?). Но connect своими связывающими программами здорово в этом помогает, беря на себя всю рутинную работу. Знакомьтесь – **connect.session()**.

Это связывающее программное обеспечение добавляет к стандартным HTTP-полям заголовков средства для определения сеанса работы (идентификатор, время жизни сеанса и необходимую передаваемую информацию – имя пользователя, корзину покупок, да что угодно). Честно говоря, сам session представляет собой реализацию сферических сеансов в вакууме, для практической их работы всегда нужно задействовать какой-нибудь механизм передачи данных сеанса. Например, те же cookie. **connect.cookieSession()** как раз так и работает. Свяжем программу еще с парой полезных компонентов СПО:

```
.use(connect.logger('dev'))
.use(connect.static(__dirname + '/public'))
.use(connect.cookieParser())
.use(connect.cookieSession({ secret: 'sekret',
                             key: 'node_site',
                             maxAge: 60 * 60 *1000 }))
```

Сначала необходимо подключить **connect.cookieParser** – СПО, предназначенное для работы с файлами cookie. Затем **connect.cookieSession**, добавляющая к обычному объекту HTTP request поле session, через которое будут доступны данные сеансы работы. **connect.session()** доступны следующие параметры (все не обязательны):

- key – имя cookie (по умолчанию connect.sess);
- secret – строка, используемая для шифрования информации в cookies;
- store – объект, используемый для хранения данных сессии. По умолчанию это оперативная память, но в рабочих приложениях это может быть файл, база данных, memcached и т. д.;
- cookie – набор параметров cookie. maxAge, время жизни в миллисекундах (в нашем примере – 1 час).

Сейчас перезапустим наш сервер и снова зайдем браузером на url http://localhost:3000/index.html. Сервер оставил у нас в браузере «печеньку» с идентификатором сеанса.

Естественно, cookies можно не только создавать, запись и чтение в них производятся следующим образом:

```
.use(connect.cookieParser())
.use(connect.cookieSession({ secret: 'tobo!',
                             key: 'node_site',
                cookie:  { maxAge: 60 * 60 * 1000 }}))
.use(function(req, res){
console.log(req.cookies);
req.session.username = 'babay';
```

В консоли при этом мы получим следующее:

```
GET /style.css 304 8ms
{ node_site: 's:j:{"username":"baybay"}.yjJYJqsyg2o5r4slKr4YW9cQbNb2g49BIwlyUyo6
xvk' }
```

Фактически в **request.session** можно записать любую переменную. Мы воспользуемся этим, организовав нечто вроде авторизации (без всяких средств безопасности и контроля доступа, сейчас не про это). Для начала введем переменную, отвечающую за авторизацию, вернее, что там, сразу начнем с проверки:

```
if(request.session.authorized != true){
    req.session.username = 'babay';
}
```

Теперь добавим еще одну программку СПО:

```
.use(connect.query())
.use(function(req, res){
```

Эта маленькая программа просто разбирает строку запроса (без нее можно спокойно обойтись, но зачем? Она помогает пять строк кода свести к одной, и это здорово!). На HTML-страницу поместим простую форму авторизации:

```
  <form name = "loginForm" action ="/login" >
  Введите имя (если не сложно):
  <input type = "text" name= "login">
  <input type = "submit" value="OK">
  </form>
```

Ну а теперь сама авторизация:

```
.use(function(req, res){
 console.log(req.url);
    if(req.session.authorized != true){
```

```
      var qr = req.query;
     var qr = req.query;
        if(qr.login != undefined){
            console.log("test");
            req.session.authorized = true;
            req.session.username = qr.login;
            res.statusCode = 302
            res.setHeader('Location','/index.html');
            res.end();
        }
         console.log("Здесь не пойми кто");
    } else {
       console.log("Здесь "+req.session.username);
    }

  })
```

Запускаем и вводим логин:

```
$ node connect

GET /favicon.ico 200 119999ms
Здесь не пойми кто

GET /favicon.ico 200 120020ms
GET /login?login=Geol 302 12ms
Здесь Geol
```

Всё здорово, но получившийся код, хоть и работает, немного не укладывается в идеологию connect-приложения – многослойную цепочку вызовов. Было бы неплохо оформить код авторизации в виде связывающей программы. Эта хорошая идея осуществима, только надо чуть подробнее разобраться, как устроены связывающие программы.

Пишем свое СПО

Для того чтобы создать свое connect-расширение, нужно соблюдать несколько правил. Их можно продемонстрировать на исходном коде какого-нибудь готового СПО, например на той же **connect.query()**:

```
module.exports = function query(options){
  return function query(req, res, next){
    if (!req.query) {
      req.query = ~req.url.indexOf('?')
        ? qs.parse(parse(req).query, options)
        : {};
    }

    next();
  };
};
```

У всех связывающие программы connect возвращают функцию, на «вход» которой подаются три аргумента – запрос, ответ и обратный вызов next – ссылка на следующую программу в цепочке. Вызов next происходит, если функция не смогла обработать поступивший запрос или обработала его не до конца, а также в случае ошибки.

Нам нет необходимости выносить код функции авторизации в отдельный модуль, следовательно, экспорт тут оформлять не нужно, главное – определить условие, при котором функция сработает:

```
function loginUser(req, res, next){
    var qr = req.query;
    if( (req.session.authorized != true) &&
         (qr.login != undefined) ){
               req.session.authorized = true;
               req.session.username = qr.login;
                res.statusCode = 302
               res.setHeader('Location','/index.html');
              res.end();
    } else {
        next();
    }
}
```

А основной код приложения станет простым и понятным:

```
.use(connect.cookieSession({ secret: 'tobo!',key: 'node_site', cookie:  { maxAge:
60 * 60 * 1000 }}))
.use(connect.query())
.use(loginUser)
.use(function(req, res){
    if(req.session.authorized != true){
         console.log("Здесь не пойми кто");
    } else {
        console.log("Здесь "+req.session.username);
    }
})
```

Теперь нужно написать код для ответственной процедуры «разлогинивания». Ее тоже логично оформить как связанную программу:

```
function logoutUser(req, res, next){
    var qr = req.query;
    if( (qr.logout == 1) ){
               req.session.authorized = false;
               req.session = null;
                res.statusCode = 302
               res.setHeader('Location','/index.html');
              res.end();
    } else {
        next();
    }
}
```

Подключим ее в цепочку:

```
.use(connect.query())
.use(loginUser)
.use(logoutUser)
.use(function(req, res){
    if(req.session.authorized != true){
```

Обратите внимание, порядок включения связывающего ПО в цепочку имеет значение! Нельзя включать **connect.cookieSession()** до **connect.cookieParser()**, у нас на этот момент просто нет cookie! Точно так же функции, которые мы только что написали, следует подключать после **connect.cookieParser()**, **connect.cookieSession()** и **connect.query()** (мы используем **ryquest.query()**).

Для того чтобы «разлогинирование» работало, достаточно разместить на HTML-странице ссылку вида:

```
<a href='/?logout=1' >Logout</a>
```

Все! Простая авторизация у нас работает. Правда, без пароля и прочих излишеств, но они и не всегда нужны.

Еще немного Connect

Впрочем, базовую http-аутентификацию connect включает практически одной строчкой:

```
.use(connect.basicAuth('babay', '1234'));
```

И это все! Теперь при заходе на сайт неавторизованного пользователя встретит форма basic HTTP-авторизации. Впрочем, в реальном приложении этот процесс следует сделать более разумным, используя функцию обратного вызова. Например, так:

```
  .use(connect.basicAuth(function(user, pass){
    return user =='babay' && pass == '1234';
  }))
```

Или даже так:

```
  .use(connect.basicAuth(function(user, pass, checkAuth){
    User.authenticate({ user: user, pass: pass }, checkAuth);
  }))
.............................
function  checkAuth(err, user){
    // проверяем логин и пароль, пользуясь базой данных или другим
    // хранилищем информации
}
```

Всего в «базовую поставку» модуля connect на сегодняшний день входят 23 программы – нужные и полезные. Этого достаточно? Практика показывает, что достаточно не бывает, и на сайте модуля на github.com (https://github.com/senchalabs/connect/wiki) представлен немаленький список СПО, написанного сторонними производителями. Там есть много интересного – например, **connect-memcached** – СПО, позволяющее хранить сессии на memcached-сервере, или **connect-php** – коннектор для php-приложений. Отличие этого СПО от «родного» только одно – эти программы требуют установки (с помощью npm install) и явного подключения посредством require.

В общем, работать с connect одно удовольствие, и дело тут даже не в экономии строчек кода, а в организации программы, которая становится простой и легко расширяемой. Не случайно именно этот модуль лежит в основе работы популярных Node.js-фреймворков, таких как Locomotive, SocketStream, RailwayJS, Chilly, sails.js, TrinteJS, и прежде всего фреймворка Express, работу с которым мы начнём в следующей главе.

Веб-каркас для node (node.js web-framework'и)

Фреймворк – это готовая структура программного приложения. Это программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. Его можно определить как набор инструментов – библиотек и соглашений, предназначенный для вынесения рутинных задач в отдельные модули, которые можно использовать многократно. Главная цель фреймворка – позволить программисту сфокусироваться на задачах, уникальных для каждого проекта, вместо неоднократного изобретения множества «велосипедов». Программа на фреймворке строится из двух частей: первая, постоянная часть – каркас, не меняющийся от конфигурации к конфигурации и несущий в себе гнезда, в которых размещается вторая, переменная часть – сменные модули (или точки расширения).

Что такое web-framework?

Тут надо уточнить. Под этим термином обычно понимают web application framework (WAF) – программный каркас веб-приложений. Само собой, такой известный фреймворк, как Cocoa, или такой инструмент, как Google Web Toolkit, и даже уже описанный Connect в эту категорию не попадают. Целью WAF является среда, берущая на себя организацию построения веб-приложения – веб-службы, веб-сайта, веб-ресурса. Обычно такой продукт содержит средства маршрутизации запросов, библиотеки для работы с источниками данных (с базами данных и/или NoSQL-хранилищами, шаблонизаторы, управление сеансами работы, генерацию форм, работу с почтой и MIME-типами).

Веб-фреймворки существуют для любой популярной среды веб-разработки. Struts, Django, Ruby on Rails, Symfony, Yii, Spring MVC, Stripes, CodeIgniter, WebObjects, web2py, Catalyst, Mojolicious, Ruby on Rails, Grails, Django, Zend Framework, CakePHP, Symfony написаны для работы на Java, Python, Ruby, PHP. Что касается JavaScript, то сразу после появления Node.js стали появляться WAF и для нее. Сейчас их довольно много, и, надо сказать, среди них есть очень инте-

ресные решения. Но по меркам развития платформы довольно давно (3 года назад) один фреймворк занял лидирующее положение, став стандартом Node.js-разработки де-факто. Я говорю о WAF Express.js.

Express

Фреймворк Express.js был разработан как аналог Sinatra, веб-фреймворка для Ruby. Он довольно низкоуровневый, но вот что у него есть на настоящий момент:

- простая маршрутизация;
- помощники перенаправления (redirect helpers);
- динамические помощники представлений;
- опции представлений уровня приложения;
- ориентированность на высокую производительность;
- рендеринг шаблонов, поддержка различных шаблонных движков;
- конфигурации, быстро переключаемые под разные задачи (development, production и т. д.);
- все возможности фреймворка **Connect**;
- скрипты для быстрой генерации каркаса приложения.

Начало работы с Express

Итак, за дело. Сначала устанавливаем модуль express:

```
$ npm install -g express
```

Теперь немедленно попробуем использовать такое преимущество веб-фреймворков, как генерация каркаса приложения (да-да, наше приложение будет готово уже через минуту!). В папке проектов выполняем консольную команду:

```
$ express myApp
```

Результат должен быть следующим:

```
create : myApp
create : myApp/package.json
create : myApp/app.js
create : myApp/public
create : myApp/public/javascripts
create : myApp/public/images
create : myApp/public/stylesheets
create : myApp/public/stylesheets/style.css
create : myApp/routes
create : myApp/routes/index.js
```

```
   create : myApp/routes/user.js
   create : myApp/views
   create : myApp/views/layout.jade
   create : myApp/views/index.jade

   install dependencies:
     $ cd myApp && npm install

   run the app:
     $ node app
```

Как мы можем понять, команда создала папку и структуру нашего будущего приложения (рис. 29).

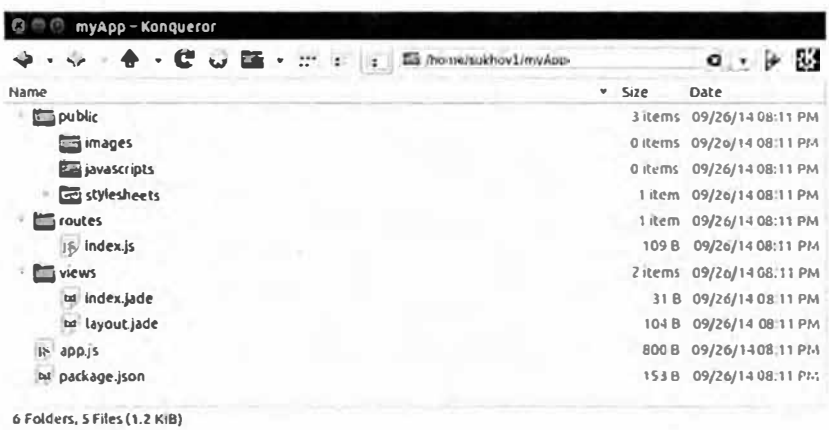

Рис. 29 ❖ Начальная структура Express-приложения

В файле **pakage.json**, сгенерированном в окне приложения, мы можем наблюдать следующее:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.7",
    "jade": "*"
  }
}
```

Оформим пакетные независимости с помощью npm:

```
cd MyApp
npm install -d
```

Теперь в нашем приложении появилась папка node_modules, в которую будут помещены все необходимые для работы модули Node.js. По умолчанию там уже установлены express и шаблонизатор jade (рис. 30).

Все! Как и было обещано, наше полноценное и совершенно бесполезное Express-приложение готово. Запустим его:

```
$ node app
```

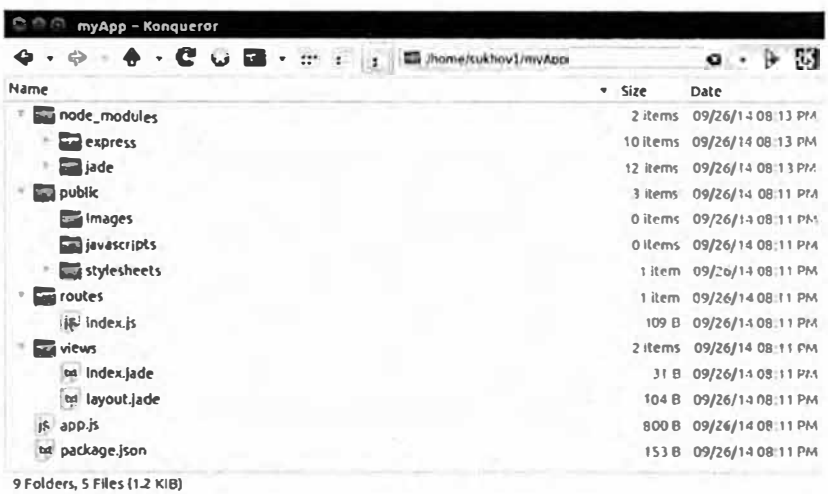

Рис. 30 ❖ Добавляем модули

Если все сделано правильно, результат будет следующим:

```
Express server listening on port 3000
```

Ну что-же, откроем в браузере адрес http://localhost:3000. Как видите, приложение работает (рис. 31).

Закат солнца вручную

Генерация каркаса – это, конечно, хорошо, но иногда эта операция приносит совсем не тот результат, который нам по какой-либо причине необходим. Можно ли при построении Express-приложения обойтись без этой стадии и сделать все «ручками»? Конечно, можно,

Рис. 31 ❖ Express работает!

и этим мы сейчас займемся, ровно потому, что так принципы и идеология фреймворка станут ближе и понятнее.

Приступим. Сначала создадим папку с будущим приложением.

```
$ mkdir band
$ cd band
```

Затем создадим там файл **pakage.json** следующего содержания:

```
{
  "name": "kakomey",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.7",
    "jade": "*",
    "stylus": "*",
    "mongoose": "*"
  }
}
```

Теперь выполним:

```
npm install
```

И npm загрузит и установит заказанные нами пакеты. Сейчас можно приступать к самому приложению, стартовым файлом которого мы назначили app.js. Создадим этот файл:

```
var express = require('express');
var app = express();
```

Собственно, на этом приложение закончено и даже работоспособно. Но, чтобы его работа хоть как-то была видна, его стоит немного дополнить:

```
var express = require('express');
var app = express();
var port = process.env.PORT || 3000;
app.get('/', function (req, res) {
  res.send('Hello, MyExpress!')
})
app.listen(port);
console.log('Listening on port ' + port);
```

Я думаю, этот код пояснений не требует. Разве что метод **app.get()** – он тут назначает функцию для прослушивания запросов. Запустим приложение:

```
$ node app
Listening on port  3000
```

Набрав сейчас в браузере url http://localhost:3000, мы можем полюбоваться текстом «Hello, MyExpress!» (рис. 32).

Рис. 32 ❖ Express, собранный «в ручную». Работоспособен

Теперь добавим нашему приложению немного интеллекта, который выразится в обработке входящих http-запросов. Это пока не полноценная маршрутизация, но уже вполне рабочее приложение:

```
var express = require('express');
var app = express();
var port = process.env.PORT || 8080;

var bands = ['Cure','Clash','Damned'];
app.get('/bands/:name?', function(req, res, next) {
    var name = req.params.name;
    for (key in bands) {
         if (bands[key] === name) {
             res.send(name + ' is Best!');
         }
     }
    next();
});

app.get('/bands/', function(req, res){
    var body = "";
    for (key in bands) {
        body += "<a href=/bands/"+bands[key]+">"+bands[key]+"</a><br />";
    }
    res.send(body);
});

app.get('/bands/*+', function(req, res){
    res.send('Unknown band!');
});
app.get('/', function (req, res) {
    res.send('Hello, MyExpress!!!')
})
```

Здесь мы создали цепочку вызовов метода **express.get()**, причем каждый последующий вызов получает управление в случае неудачи в работе предыдущего. Если это вам напоминает работу модуля connect – ничего удивительного, во-первых, у них один автор-основатель (TJ Holowaychuk), во-вторых, как уже было сказано, connect – неотъемлемая часть фреймворка.

В первый вызов попадают все запросы вида http://адрес_сайта/bands/имя_группы. Затем проверяется, есть ли это имя в заранее заданном списке групп (само имя групп вычленяется из url, после разбора **get()** регулярного выражения, в объект **request.params**). В случае удачи в бразер автора запроса будет послана чрезвычайно полезная информация (рис. 33), в противном случае управление будет передано следующему вызову метода **get()**. Обратите внимание: нам приходится делать это явным образом, используя специальный метод **next()**. Дело в том, что условие (соответствие регулярному выражению) было соблюдено, вызов прошел как успешный, и если по

Рис. 33 ❖ Цой жив!

дополнительной логике что-то не так – проброс приходится вызывать вручную.

Второй вызов сработает только при строгом соответствии url строке «http://адрес_сайта/bands/» и приведет к выводу списка групп, третий сработает при неизвестных параметрах запроса, наконец, последний, оставшийся от начального варианта кода, обработает url, указывающий на корень сайта.

Подключаем шаблонизатор

Следующим этапом подключим к нашему приложению шаблонизатор. Может быть, сейчас это не самая логически первоочередная задача, но HTML-разметка в основном файле приложения выводит меня из себя. За работу. Сначала добавим в **app.js** следующие три строчки:

```
var port = process.env.PORT || 8080;
app.set('view engine', 'jade');
app.set('view options', { layout: true });
app.set('views', __dirname + '/views');
```

Метод app.set() используется для определения различных параметров приложения. Тут мы обозначили папку для хранения шаблонов представления, в данном случае .jade-шаблонов, и подключили сам шаблонный движок. У нас здесь есть выбор, но я остановился на Jade. Настройка view options указывает на актуальность шаблона, она будет накладываться при каждом рендеринге представления.

Создадим папку /views и разместим там шаблоны представления. Сначала основной layout, **layout.jade**:

```
doctype html
html
  head
    title= title
  body
    block content
```

Теперь индексная страница **index.jade**:

```
extends layout

block content
  h1= title
  div  #{textBody}
```

Так мы создали заготовку шаблона главной страницы. (Если кто-нибудь забыл синтаксис jade – в блок body будет помещен шаблон из **index.jade**.) Далее шаблон для странички группы – **band.jade**:

```
extends layout

block content
  h2= title
  div  #{name} is Best!
```

И самый сложный шаблон – для списка групп – band_list:

```
extends layout

block content
  h2= title
  ul
    - for (var b in bandsList)
      li
          a(href='/bands/'+bandsList[b])= bandsList[b]
```

Теперь основной скрипт можно переписать более лаконично:

```
app.get('/bands/:name?', function(req, res, next) {
    var name = req.params.name;
        for (key in bands) {
        if (bands[key] === name) {
            res.render('band', {title:'Bands',name: name});
            }
        }
        next();
});

app.get('/bands/', function(req, res){
```

```
    res.render('band_list', {title:'Bands',bandsList: bands});
});

app.get('/bands/:name?', function(req, res){
    res.send('Unknown band!');
});
```

и

```
app.get('/', function (req, res) {
  res.render('index', {title:'Bands',textBody: 'Hello, MyExpress!!!'});
})

app.listen(port);
console.log('Listening on port ' + port);
```

Гораздо лучше, правда?

Задействуем статику

Следующим этапом подключим нашему приложению возможность работы со статикой, то есть с ресурсами, которые не требуют генерации. Это могут быть скрипты, css-файлы, картинки... да-да, самое главное – картинки!

Сначала создадим папку /public и там разместим отдельные папки, предназначенные для статичных ресурсов, – /scripts, /stylesheet, /images.

Затем дополним **app.js** одной строчкой, применив уже знакомый нам **connect**-модуль **static**:

```
app.use(express.favicon());
app.use(express.logger());
app.use(express.static(path.join(__dirname, 'public')));
});
```

Теперь поместим в папку /public/images логотип (**band.jpg**) и пропишем его в шаблоне (**layout.jade**):

```
body
  img(src="/images/bands.jpg")
  block content
```

Все – логотип присутствует на всех немногочисленных страницах сайта (рис. 34).

Подключаем CSS

Что дальше? Ну, неплохо бы подключить к веб-странице таблицы стилей. С одной стороны, это просто (шаблон **layout.jade**):

Рис. 34 ❖ Работа шаблонизатора

```
doctype html
html
  head
    title= title
    link(type="text/css",
           rel='stylesheet',
         href='/stylesheets/style.css')
  body
    block content
```

Но нам нужна не просто css-таблица. Нам желательно встроить в приложение css-препроцессор. И в express это совсем не сложно:

```
var http = require('http');
var path = require('path');
var stylus = require('stylus');
…….....................................
app.use(express.static(path.join(__dirname, 'public')));
});
app.use(stylus.middleware({
src: __dirname + '/views'
, dest: __dirname + '/public'
}));
```

Теперь в папку /public/stylesheets/ мы можем поместить stylus-шаблон. Например, такой (**style.styl**):

```
body
  font 14px "Lucida Grande", Arial, sans-serif
  padding 50px
  margin 20px
a
  color: #00B7FF
```

При первом запуске приложения в папке /public/stylesheets/ будет сгенерирован файл **style.css**:

```
body {
  font: 14px "Lucida Grande", Arial, sans-serif;
  padding: 50px;
  margin: 50px;
}
a {
  color: #00b7ff;
}
```

который мы заранее подключили к шаблону. Правда, теперь при каждом редактировании stylus-шаблона нам придется перезапускать приложение, но и это и все.

Ну что же, то, что у нас получилось, уже начинает напоминать нормальное приложение. Правда, пока только начинает. Давайте двигаться в этом направлении.

Разработка RESTFul-приложения на основе Express

Почему REST? Честно говоря, Express не является специализированным фреймворком для RESTFul-приложений. Да и REST – не единственная архитектура в мире веб-приложений. Но, с другой стороны, Express – самый популярный Node.js-фреймворк, а RESTFul API стал в последнее время стандартом. Посему первое наше приложение будет реализацией RESTFul API на Express framework.

Немного о REST-архитектуре

REST (*Representational State Transfer* – передача репрезентативного состояния) – это способ построения архитектуры распределенного приложения. Он был описан и популяризован в 2000 году Роем Филдингом (Roy Fielding), одним из создателей протокола HTTP.

Данные в REST должны передаваться в виде небольшого количества стандартных форматов (например, HTML, XML, JSON). Сетевой протокол (как и HTTP) должен поддерживать кэширование, не должен зависеть от сетевого слоя, не должен сохранять информацию о состоянии между парами «запрос–ответ». Утверждается, что такой подход обеспечивает масштабируемость системы и позволяет ей эволюционировать с новыми требованиями.

Антиподом REST является подход, основанный на вызове удаленных процедур (Remote Procedure Call – RPC). Подход RPC позволяет использовать небольшое количество сетевых ресурсов с большим количеством методов и сложным протоколом. При подходе REST количество методов и сложность протокола строго ограничены, из-за чего количество отдельных ресурсов может быть большим.

Приступаем к реализации RESTful API

Сначала позаботимся о загрузке необходимого функционала:

```
app.use(stylus.middleware({
src: __dirname + '/views'
, dest: __dirname + '/public'
}));
app.use(express.favicon());
app.use(express.logger());
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));
```

Как видите, мы добавили в приложение связывающие программы. Fivicon, logger, static – это наши старые знакомые по фреймворку **connect** (собственно, это и есть connect-программы).

Кроме них, мы тут задействовали еще несколько СПО. **BodyParser()** – программа, функция которой – выполнять синтаксический разбор тела входящего запроса, преобразуя нужные элементы в значения свойств объекта запроса. **MethodOverride()** позволяет эмулировать REST-вызовы (put, delete). Работа обеих программ будет продемонстрирована ниже.

В нашем приложении мы должны реализовать такой функционал, как добавление и редактирование названий групп. Другими словами, мы будем строить полнофункциональное RESTful API.

Начнем с добавления новой группы. Добавим на шаблон списка групп соответствующую ссылку:

```
a(href='/add/') Добавить группу
```

А в файле **app.js** пропишем обработчик запроса, соответствующего этому url:

```
app.get('/bands/:name?', function(req, res){
  res.send('Unknown band!');
});
app.get('/add/', function(req, res){
  res.render('add_form');
});
```

Все, что делает этот код, – выводит форму для добавления новой группы. Её шаблон (**/view/add_form.jade**) очень прост. extends layout:

```
extends layout

block content
  div
    form(method='POST' action='/add/') Добавить группу
      br
      input(type='text' name ='band_name')
      br
      input(type='submit')
  a(href='/bands/') Назад
```

Обратите внимание: мы отправляем данные методом POST. Это принципиально. По REST-идеологии запрос, совершенный через GET, не должен менять состояния сервера (это не стилистический каприз). Ответ сервера, в принципе, должен быть доступен для кэширования на определенный период времени и повторного использования без новых запросов к серверу. Поэтому на сервере мы будем слушать POST-запрос:

```
app.get('/add/', function(req, res){
  res.render('add_form');
});
app.post('/add/', function(req, res){
  bands.push(req.body.band_name);
  res.render('band_list', {title:'Bands',bandsList: bands})
});
```

Тут работает метод **bodyParser()**, позволяющий извлекать параметры непосредственно из переданной HTML-формы. В остальном – ничего сложного: мы просто добавляем в массив новое название и рендерим шаблон с обновленными данными. Осталось проверить – все должно работать (рис. 35).

Теперь предоставим пользователю редактировать название группы. Добавим ссылку для этой операции на шаблон **band_list.jade**:

Рис. 35 ❖ Движок работает

```
    li
      a(href='/bands/'+bandsList[b])= bandsList[b]
       
      a(href='/edit/'+bandsList[b]) edit
```

Маршрутизация в **app.js**:

```
app.get('/edit/:name?', function(req, res){
  res.render('edit_form',{name: req.params.name});
});
```

Форма для редактирования чуть-чуть посложнее (файл **edit_form.jade**):

```
extends layout

block content
```

```
div
  form(method='POST'  action='/edit/')
    input(type='hidden' name ='_method' value='put')
    br
    input(type='text' name ='newName' value=name)
    br
    input(type='hidden' name ='name' value=name)
    br
    input(type='submit')
a(href='/bands/') Назад
```

Обратили внимание на скрытое поле? Я, наверное, зря демонстрирую HTML-код на шаблонах jade. Это, конечно, ближе к реальности, но менее наглядно.

В откомпилированном виде форма выглядит так:

```
<form method="POST" action='/edit/'>
<input type="hidden" name="_method" value="put" action="?name=Cure" >
<br>
<input type="text" name="newName" value="Cure"><br>
<input type="hidden" name="name" value="Cure"><br>
<input type="submit">
</form>
```

Тут предоставляется поле для работы метода **methodOverride()**. Обновление, «по идее», должно проводиться HTTP-методом PUT. Почему именно PUT? Ведь все можно передать тем же POST'ом? Опять – дело принципа, причем оправданного принципа.

В чем преимущества PUT? Есть такой математический термин – идемпотентность. Он не имеет ничего общего с сексуальными расстройствами, а означает свойство операции, которое проявляется в том, что повторное действие над объектом не изменяет его. Пример такой операции – взятие модуля. Можете хоть тысячу раз повторять эту операцию, присваивая результат исходному аргументу, значение, полученное первый раз, никогда не изменится.

Так вот, согласно спецификации, HTTP-запросы типа PUT (как, впрочем, и GET) идемпотентны. Такая особенность позволяет кэшировать ответы, но нас в данном случае интересует не это. Дело в том, что метод POST не идемпотентен. Он работает как инкремент. На практике это означает создание POST-запросами с одинаковым набором парамеров дубликатов объектов. Операция же, проведенная посредством PUT-запроса, будучи повторенной несколько раз с одинаковыми аргументами, приведет к тому же результату, что и однократный запрос. Что и требуется в случае, когда мы изменяем (устанавливаем) свойства уже существующего объекта.

Я убедил? Будем использовать PUT.

Но вот незадача – HTTP-формы таким методом пользоваться не умеют. Поэтому применяется специальное поле _method, значение которого **methodOverride()** интерпретируется как название метода.

```
app.put('/edit/:name?', function(req, res){
    var name = req.params.name;
      for (key in bands) {
        if (bands[key] === name) {
            bands[key] = req.params.newName;
        }

      }
    res.render('band_list', {title:'Bands',bandsList: bands})
});
```

Теперь удаление. Добавим в шаблон band_list.jade еще одну ссылку:

```
        a(href='/edit/'+bandsList[b]) edit
         
        a(href='/del/'+bandsList[b] alt="Elfkbnm") delete
```

Вывод формы для удаления ничем не отличается от аналогичного действия для других операций:

```
app.get('/del/:name?', function(req, res){
  res.render('delete_form',{name: req.params.name});
});
```

И сама форма:

```
extends layout

block content
  div
    form(method='POST') Удалить группу?
      input(type='hidden' name ='_method' value='delete' action='?name='+name)
      br
      input(type='text' name ='band_name' value=name readonly="readonly")
      br
      input(type='submit')
  a(href='/bands/') Назад
```

Тут мы так же, посредством фокусов с **methodOverride()**, применяем «не форменный» HTTP-метод DELETE. Он идемпотентен и используется для удаления объекта с указанным URI, в случае если таковой существует. На стороне сервера обрабатываем запрос типа DELETE:

```
app.del('/del/:name?', function(req, res){
    var name = req.params.name;
```

```
        for (key in bands) {
        if (bands[key] === name) {
                delete bands[key];
           }
        }
    res.render('band_list', {title:'Bands',bandsList: bands})
});
```

Все! Рабочее JavaScript RESTFull-приложение готово! Разработка не отняла много времени, правда?

Подключаем источник данных

У нас действительно получилось функциональное и, самое главное, быстро разработанное приложение. Правда, в таком виде полезность его вызывает сомнения, но с минимальными доработками эту архитектуру можно приспособить, например, под движок хит-парада или афиши. У нас же задачи немного сложнее. Я хочу сделать нечто вроде онлайн-рок-энциклопедии, и это предполагает хранение немного расширенного объема данных и хранения персистентного. Для этого нужна база данных, и я, пользуясь правами единоличного автора, выбираю MongoDB и модуль Mongoose (собственно, это можно было предвидеть по набору пакетов).

Будем считать, что сама база у нас создана и даже содержит данные (попавшие туда при проработке предыдущих глав). Нам нужно написать коннектор, доставляющий данные приложению (**mongoose.php**):

```
var mongoose = require('mongoose');
var db = mongoose.createConnection('mongodb://localhost/bands');
    var BandSchema = new mongoose.Schema( {
        bid: { type: Number, index: true, min: 1 },
        name: { type: String, match: /[a-z0-9 ]/},
        state: {type: String, default: "uk"},
        members:[{aid: Number, name: String}]
    });
db.on('error', function (err) {
    console.log("connection error:", err.message);
});
db.once('open', function callback () {
    console.log("Connected to DB!");
});
var BandModel = db.model('Band', BandSchema);
module.exports.BandModel = BandModel;
```

Добавим конектор в основной файл сценария и создадим соответствующий объект:

```
var port = process.env.PORT || 8080;
var routes = require('./routes');
var BandModel    = require('./mongoose').BandModel;
```

И перепишем вывод списка групп в соответствии с новыми реалиями:

```
app.get('/bands/', function(req, res){
  BandModel.find(function (err, bands) {
          if (!err) {
          res.render('band_list', {title:'Bands',bandsList: bands));
       } else {
            console.log(err);
          }
       });

});
```

И чуть изменим шаблон:

```
  - for (var b in bandsList)
    li
      a(href='/bands/'+bandsList[b].bid)= bandsList[b].name
       
      a(href='/del/'+bandsList[b].bid) del
       
      a(href='/edit/'+bandsList[b].bid) edit
```

Теперь мы больше не передаем в качестве параметра название группы (это было ужасно!), вместо этого мы используем её идентификатор. Это, конечно, неизбежно ведет к модификации остального кода, но пока у нас более очевидная проблема (рис. 36) – названия групп выводятся в произвольном порядке, список не отсортирован.

Далее переписываем обработку REST-запросов. Вывод информации о группе:

```
app.get('/bands/:bid?', function(req, res, next) {
    var id = req.params.bid;
      console.log(id);

      BandModel.find({bid:id},function (err, band) {
         if (!err) {
         res.render('band', {title:'Band',band: band[0]});
      } else {
           console.log(err);
         }
         next();
      });

});
```

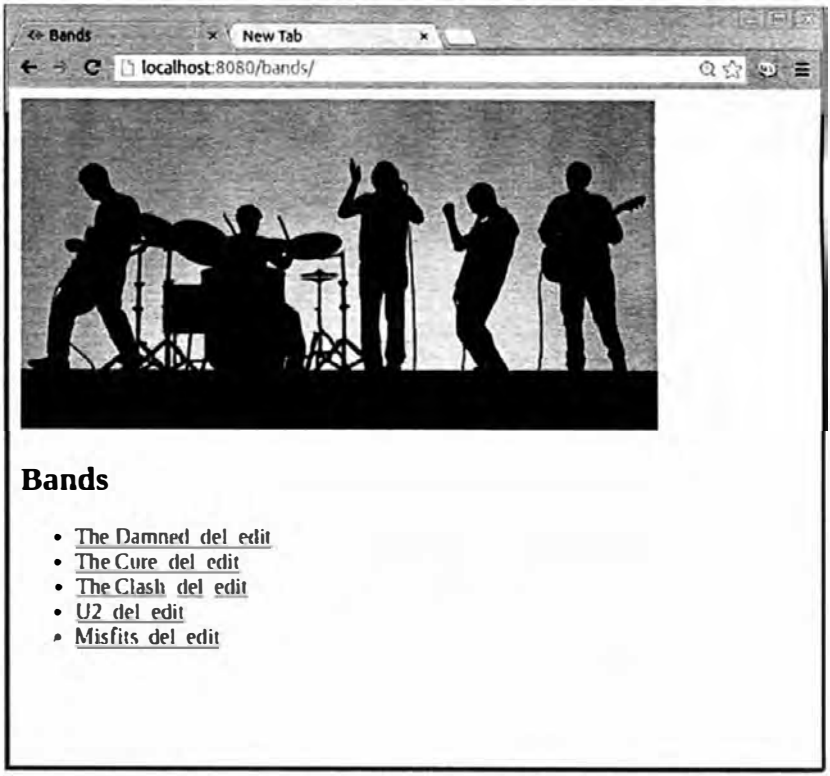

Рис. 36 ❖ Добавляем функционал

Естественно, нужен новый шаблон (как видно из схемы документа, в информацию о группе теперь входит не только название):

```
extends layout

block content
  h2= title
  h1  #{band.name}
    i  (#{band.state})
  each m in band.members
    li=m.name
```

Тут мы не можем применить конструкцию for ... in, поскольку таким перебором получим порядочное количество собственных свойств объекта. Но и так все получилось (рис. 37).

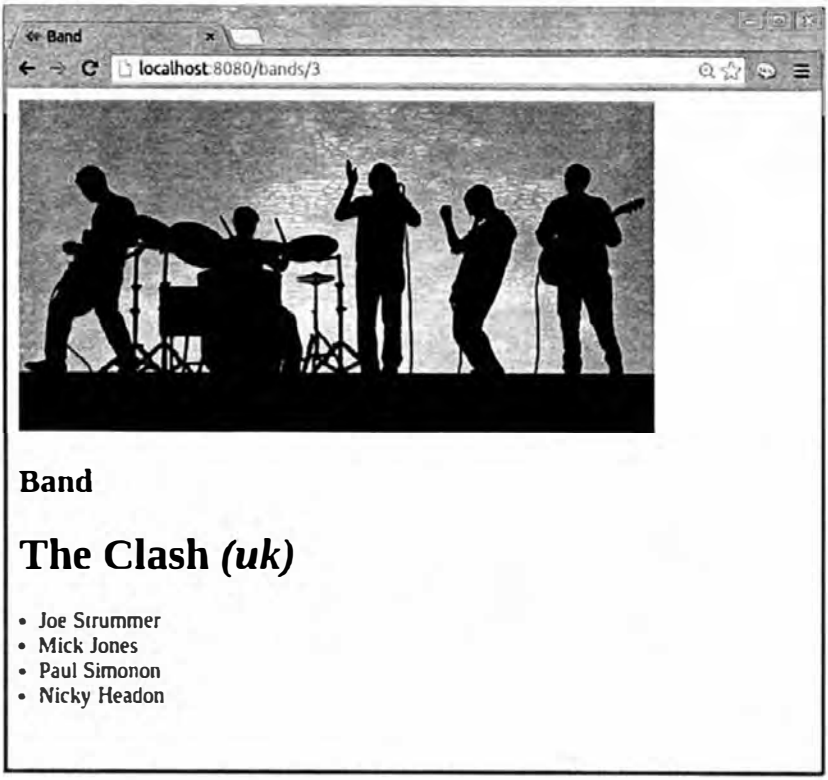

Рис. 37 ❖ White Riot!

Теперь добавление группы. Сначала новая форма (**add_form.jade**):

```
extends layout

block content
  div
    form(method='POST' action='/add/') Добавить группу
      br
      b Название
      br
      input(type='text' name ='band_name')
      br
      b Страна
      br
      input(type='text' name ='band_state')
      br
```

```
      b Состав
      br
      textarea(name ='members')
      br
      input(type='submit')
  a(href='/bands/') Назад
```

Верстка безобразная, юзабельность нулевая (имена участников группы придется вводить через запятую), но зато все просто. Теперь обработка и сохранение введенных данных:

```
app.post('/add/', function(req, res){
BandModel.find().sort({bid: -1}).findOne(function (err, band) {
    var bandMembers = [];
    var members = req.body.band_members.split(',');
        for(var i = 0;i < members.length; i++) {
           person = {
              name: members[i],
              aid: ""
           }
           console.log(person);
           bandMembers.push(person);
        }
    var newBand = {
        bid: band.bid,
        name: req.body.band_name,
        state: req.body.band_state,
        members: bandMembers
    };
    Band = new BandModel(newBand);
    Band.save(function(err,data){
        if (!err) {
            console.log("Данные сохранены");
            BandModel.find({},function (err, bands) {
                if (!err) {
                    res.render('band_list', {title:'Bands',bandsList: bands});
                } else {
                    console.log(err);
                }
            });
        } else {
                console.log(err);
        }
      })
    });
});
```

Сначала из полученных параметров мы создаем объект newBand, затем создаем на основе его новую модель и сохраняем ее в базе. По-

следнее действие – вывод списка групп, лучше вынести в отдельный метод, но реструктуризацией кода мы еще займемся в следующем разделе.

Теперь удаление группы:

```
app.get('/del/:bid?', function(req, res){
      var id = req.params.bid
      BandModel.find({bid:id},function (err, band) {
           if (!err) {
           res.render('delete_form',{band: band[0]});
           } else {
           console.log(err);
           }
      });
});
```

Форма **delete_form.jade**:

```
extends layout

block content
  div
    form(method='POST' action='/del/') Удалить группу
      input(type='hidden' name ='_method' value='delete')
      input(type='hidden' name ='bid' value=band.bid)
      br
      div #{band.name}?
      br
      input(type='submit')
  a(href='/bands/') Назад
```

И обработка delete-запроса:

```
app.del('/del/:bid?', function(req, res){
    var id = req.body.bid;
    console.log(id);
        BandModel.remove({bid:id}, function(err){
        if (!err) {
                console.log('Группа удалена');
                   // далее выводим список групп
      } else {
          console.log(err);
        }
      });
});
```

Осталось редактирование:

```
app.get('/edit/:bid?', function(req, res){
    var id = req.params.bid
    BandModel.find({bid:id},function (err, band) {
```

```
        if (!err) {

            var group = band[0];
            var persons=[];
            for(i=0; i < group.members.length; i++){
                    persons.push(group.members[i].name);
                }
              res.render('edit_form',{band: group, members: persons.join(',')});
        } else {
            console.log(err);
          }
        });
});
```

Здесь ничего нового. Форма **edit_form.jade**:

```
extends layout

block content
  div
    form(method='POST'  action='/edit/')
      input(type='hidden' name ='_method' value='put')
      input(type='hidden' name ='bid' value=band.bid)
      br
      b Название
      br
      input(type='text' name ='name' value=band.name)
      br
      b Страна
      br
      input(type='text' name ='state' value=band.state)
      br
      b Состав
      br
      textarea(name ='band_members')=members
      br
      input(type='submit')
  a(href='/bands/') Назад
```

И её обработка:

```
app.put('/edit/?', function(req, res){
  var bid = req.body.bid;
  var bandMembers = [];
  var members = req.body.members.split(',');
  members.forEach(function(item){
      person = {
             name: item,
             aid: ""
          }
      bandMembers.push(person);
```

```
  });
  var updateBand = {
      bid: bid,
      name: req.body.name,
      state: req.body.state,
      members: bandMembers
  };
  BandModel.update({bid:bid}, updateBand, function(err,data){
        if (!err) {
            console.log("Данные сохранены");
        // далее выводим список групп
        } else {
        console.log(err);
        }
  });
});
```

Все! Приложение готово!

А теперь – на три буквы (на MVC)

Вообще говоря, фреймворк Express вообще не предполагает и не диктует какую-либо определенную архитектуру приложения. Да и архитектура Model-View-Controller – не единственное удачное решение в мире веб-приложений. Но бывает, что она так и просится к внедрению, и мне кажется, с нашим REST-приложением – именно такой случай. Давайте попробуем организовать код нашего рок-справочника в соответствии с заданной парадигмой.

Немного об архитектуре MVC

Я понимаю, что среди читателей хватает специалистов, детально знакомых с тремя латинскими буквами в заголовке этого раздела, они могут смело пропустить следующие два абзаца. Для остальных небольшой архитектурный ликбез.

MVC (Model-view-controller – «модель–представление–контроллер») – это архитектура, разделяющая модель данных приложения, пользовательский интерфейс и взаимодействие с пользователем на три отдельных компонента, при этом модификация одного из компонентов оказывает минимальное воздействие на остальные.

Вот что собой представляют компоненты MVC:

Модель (*Model*) – предоставляет данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти данные доносятся до конечного потребителя.

Представление (*View*). Отвечает за отображение информации (визуализацию).

Контроллер (*Controller*). Обеспечивает связь между пользователем и положением: контролирует ввод данных пользователем, управляет потоками данных, использует модель и представление для реализации необходимого фукционала.

Очень важно то, что представление (то, что доступно пользователю приложения) целиком зависит от модели. В то же время модель не зависит от реализации ни контроллера, ни представления. Тем самым достигается назначение такого разделения: оно позволяет строить модель независимо от визуального (либо какого-нибудь другого) представления, а также создавать несколько различных представлений для одной модели.

На практике в веб-приложении это выглядит следующим образом: браузер пользователя посредством интерфейса представления отправляет запрос на сервер, контроллер обрабатывает запрос, получает необходимые данные из модели и отправляет их в представление. Представление, в свою очередь, получает данные из контроллера и визуализирует их, превращая в HTML-страничку, которую контроллер в итоге отправит пользователю.

Структурируем код

Теперь посмотрим, что мы сможем сделать с уже написанным нами кодом. Мы не собираемся жестко соблюдать все каноны концепции MVC, а просто решим с помощью нее некоторые проблемы. А какие, собственно, проблемы мы имеем? Да обычные для веб-приложения – ему в настоящем виде недостает гибкости. Для расширения функционала нам необходимо ввести новую сущность – музыканта (Artist) – и прописать операции с ним. Мы можем это сделать в рамках того же **app.js**, а схему данных прописать там же, где и схема BandSchema, но вы наверняка сами знаете, чем чреват подобный подход – работать этими файлами после введения еще пары сущностей станет очень трудно. Посему будем разбивать код на функциональные части.

Начнем с того, что создадим папку **/controllers** и разместим там пока единственный контроллер – **bands.js**. Он будет содержать все те же методы, которые обрабатывают запросы на добавление/удаление/редактирование, но теперь это будут экспортируемые функции. Листинг получится немного длинным, но разбирать методы по отдельности смысла нет – функциональность не изменилась:

```
var BandModel    = require('../mongoose').BandModel;
exports.index = function(req, res) {
   BandModel.find({},function (err, bands) {
          if (!err) {
          res.render(bands/'band_list', {title:'Bands',bandsList: bands});
        } else {
          console.log(err);
          }
        });
};
// вывод информации о группе
exports.show = function(req, res) {
    var id = req.params.bid;
    BandModel.find({bid:id},function (err, band) {
          if (!err) {
          res.render('bands/band', {title:'Bands',band: band[0]});
        } else {
          console.log(err);
          }
        });
}
// вывод формы для добавления группы
exports.addForm = function(req, res) {
    res.render('bands/add_form');
};
// добавление новой группы
exports.create = function(req, res) {
    BandModel.find().sort({bid: -1}).findOne(function (err, band) {
        var bid = band.bid + 1;
        var bandMembers = [];
        var members = req.body.band_members.split(',');
        for(var i = 0;i < members.length; i++) {
             person = {
               name: members[i],
               aid: ""
             }
            bandMembers.push(person);
      }
    var newBand = {
         bid: bid,
         name: req.body.band_name,
         state: req.body.band_state,
         members: bandMembers
    };
    var Band = new BandModel(newBand);
    Band.save(function(err,data){
        if (!err) {
          console.log("Данные сохранены");
          res.redirect('/bands/');
        } else {
```

```
            console.log(err);
        }
    });
});
// подтверждение удаления группы
exports.deleteForm = function(req, res){
    var id = req.params.bid
    BandModel.find({bid:id),function (err, band) {
        if (!err) {
                res.render('bands/delete_form',{band: band[0]});
        } else {
            console.log(err);
        }
      });
}
// удаление группы
exports.delete = function(req, res){
    var id = req.body.bid;
      BandModel.remove({bid:id}, function(err){
        if (!err) {
                console.log('Группа удалена');
                res.redirect('/bands/');
        } else {
            console.log(err);
        }
      });
}
// форма для редактирования
exports.editForm = function(req, res){
    var bid = req.params.bid
    BandModel.find({bid:bid},function (err, band) {
      if (!err) {
      var group = band[0];
        var persons=[];
        for(i=0; i < group.members.length; i++){
                persons.push(group.members[i].name);
        }
        res.render('bands/edit_form',{band: group, members: persons.join(',')});
    } else {
        console.log(err);
      }
   });
}
// сохранение
exports.edit = function(req, res){
  var bid = req.body.bid;
  var bandMembers = [];
  var members = req.body.members.split(',');
  members.forEach(function(item){
      person = {
```

```
            name: item,
            aid: ""
        }
    bandMembers.push(person);
});
var updateBand = {
    bid: bid,
    name: req.body.name,
    state: req.body.state,
    members: bandMembers
};
BandModel.update({bid:bid}, updateBand, function(err,data){
    if (!err) {
    console.log("Данные сохранены");
          res.redirect('/bands/');
  } else {
    console.log(err);
  }
});
}
```

Прежде всего бросается в глаза, что все шаблоны теперь ищутся в папке /bands. Мы действительно должны создать такую подпапку в папке /views/ и поместить туда все шаблоны для работы с группами (то есть все, кроме **layout.jade** и **index.jade**). С расширением функционала приложения там неизбежно появятся папки с шаблонами, относящиеся к другим сущностям, – /artists, /genres, /relises, но пока разберемся с существующими шаблонами – их надо слегка изменить. Как именно, я покажу на примере **band_list.jade**:

```
extends ../layout

block content
  h2= title
  ul
    - for (var b in bandsList)
      li
        a(href='/bands/'+bandsList[b].bid)= bandsList[b].name
         
        a(href='/bands/del/'+bandsList[b].bid) delete
         
        a(href='/bands/edit/'+bandsList[b].bid) edit

  a(href='/bands/add') Добавить группу
```

Во-первых, мы уточнили расположение родительского шаблона, во-вторых, изменили url'лы, дополнив их именем контроллера. Теперь адрес представляет собой имя контроллера (band) и через

слэш – адрес действия (/del, /edit, /add и т. д.). Такой же «рихтовке» следует подвергнуть и остальные шаблоны.

Теперь нам нужен некий маршрутизатор (или маппер), который свяжет url-действия с соответствующей функцией контроллера. Он будет таким (файл **approute.js**):

```
exports.route = function(app, controller) {
  var controllerObject = require('./controllers/' + controller);

  app.get('/'+controller, controllerObject.index);
  app.get('/'+controller + '/:bid', controllerObject.show);

  app.get('/'+controller + '/add', controllerObject.addForm);
  app.post('/'+controller + '/add', controllerObject.create);

  app.get('/'+controller + '/del/:bid', controllerObject.deleteForm);
    app.del('/'+controller + '/del', controllerObject.delete);

  app.get('/'+controller + '/edit/:bid', controllerObject.editForm);
    app.put('/'+controller + '/edit', controllerObject.edit);
};
```

Далее перепишем основной файл приложения (app.js):

```
var express = require('express');
var stylus = require('stylus');
var routes = require('./routes');
var map = require('./approutes');

var app = express();
var port = process.env.PORT || 8080;
var http = require('http');
var path = require('path');

app.configure(function(){
    app.set('view engine', 'jade');
    app.set('view options', { layout: true });
    app.set('views', __dirname + '/views');
    app.use(express.favicon());
    app.use(express.logger());
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
    app.use(express.static(path.join(__dirname, 'public')));
    app.use(express.errorHandler());
});
app.use(stylus.middleware({
    src: __dirname + '/views',
    dest: __dirname + '/public'
```

```
}));
app.use(function(req, res, next){
    throw new Error(req.url + ' not found');
});
app.use(function(err, req, res, next) {
    console.log(err);
    res.send(err.message);
});

var controllers = ['bands'];
controllers.forEach(function(controller) {
    map.route(app, controller);
});

http.createServer(app).listen(8080);
console.log("Express server listening on port 8080");
```

Тут мы больше убрали, чем добавили. А новых вещей здесь только две – обработка ошибок (отдельно – несуществующий url) и подключение нашего маршрутизатора и привязка url-маппинга к контроллерам. Ко всем контроллерам. Правда, у нас пока один, но исправить это несложно. А пока займемся моделями.

Сначала немного сократим наш mongoose-коннектор (mongoose.js) до следующего кода:

```
var mongoose = require('mongoose');
var db = mongoose.createConnection('mongodb://localhost/bands');
db.on('error', function (err) {
    console.log("connection error:", err.message);
});
db.once('open', function callback () {
    console.log("Connected to DB!");
});
module.exports.db = db;
```

Затем создадим папку /models и в ней файл bands.js:

```
var mongoose = require('mongoose');
var db  = require('../mongoose').db;

var BandSchema = new mongoose.Schema( {
        bid: { type: Number, index: true, min: 1 },
        name: { type: String, match: /[a-z ]/},
        state: {type: String, default: "uk"},
        members:[{aid: Number, name: String}]
});
var BandModel =  db.model('Band', BandSchema);
module.exports.BandModel = BandModel;
```

Теперь в контроллере **controllers/band.js** запрашиваем эту модель:

```
var BandModel    = require('../models/bands').BandModel;
exports.index = function(req, res) {
```

И все! Приложение должно работать, как прежде. Хм... А хотелось бы чего-то нового, иначе зачем нам потребовался рефакторинг? Сейчас мы этим новым и займёмся.

Добавляем новую сущность

Теперь разберемся с музыкантами – участниками групп. Естественно, информация о них тоже должна быть доступна, и не только для просмотра, но и для изменения. Я не буду уже так подробно показывать весь код, а просто опишу необходимые действия для введения новой сущности.

Их не очень много.

Создаем новую модель данных (файл **/models/musicians.js**):

```
var mongoose = require('mongoose');
var db  = require('../mongoose').db;

var MusicianSchema = new mongoose.Schema( {
        aid: { type: Number, index: true, min: 1 },
        name: { type: String, match: /[a-z ]/},
        birth: {type: Date},
     death: {type: Date},
     bio: {type: String},
        groups:[{bid: Number, name: String}]
    });

MusicianSchema.statics.getLastBid = function (){
    var coll =this.find({}).sort({'bid': -1}).limit(1);
  console.log(coll.bid);
}
var ArtistModel =  db.model('Musician', MusicianSchema);
module.exports.MusicianModel = MusicianModel;
```

Тут минимальный набор данных для хранения сведений об участнике группы. Как и раньше, будем считать, что коллекция Musicians в MongoDB уже создана и содержит данные.

Теперь контроллер (**controllers/musician.js**):

```
var MusicianModel    = require('../models/musician').MusicianModel;
// список музыкантов
exports.index = function(req, res) {
   MusicianModel.find({},function (err, musician) {
          if (!err) {
```

```
            res.render('musician/musician_list', {title:'Musicians',bandsList:
musicians});
        } else {
            console.log(err);
        }
    });
};
// вывод информации о музыканте
exports.show = function(req, res) {
    var id = req.params.mid;
    MusicianModel.find({mid:id},function (err, musician) {
        if (!err) {
            res.render('musicians/musician', {title:'Bands',musician:
musician[0]});
        } else {
            console.log(err);
        }
    });
}
```

Остальные функции контроллера я приводить не буду – напишете сами, принцип, по-моему, понятен. Разумеется, в папке /view/ следует создать подпапку /musicians/ для соответствующих шаблонов, а массиву controllers в основном файле приложения нужно прописать новый контроллер:

```
var controllers = ['bands', 'musicians'];
```

Если новый контроллер использует свои, оригинальные методы, в маршрутизатор approuter.js нужно добавить соответствующие строки:

```
app.get('/'+controller + '/foo', controllerObject.foo);
app.get('/'+controller + '/bar', controllerObject.bar);
```

где «foo» и «bar» – оригинальные методы.

Теперь новый функционал доступен приложению. Например, можно сделать «кликабельными» фамилии музыкантов в списке в шаблоне **band.jade**:

```
extends layout

block content
  h2= title
  h1  #{band.name}
    i  (#{band.state})
  each m in band.members
    li
      a(href='/musicions/'+m.mid)m.name
```

Структура нашего приложения в итоге будет примерно такого вида:

```
/node_modules
/public
    /images
        /band.jpg
    /javascripts
    /stylesheets
       /style.css
/routes
    /index.js
/models
    /bands.js
    /mosicuans.js
/views
    /bands
        /band.jade
        /band_list.jade
        /add_form.jade
        /delete_form.jade
        /edit_form.jade
    /musicians
        /musician.jade
        /musician_list.jade
        /add_form.jade
        /delete_form.jade
        /edit_form.jade
    /style.styl
    /layout.jade
    /index.jade
/cointrollers
    /bands.js
    /musicians.js
/mongoose.js
/approutes.js
/app.js
/package.json
```

Все просто, расширяемо и без лишних конструкций. Правда, приложение еще разрабатывать и разрабатывать. Но это уже будет сделано за пределами данной книги. Нам же предстоит освоить еще много чего интересного, ведь разрабатывать приложения на Node.js по-взрослому мы только начали.

Практика разработки приложений Node.js

Nodemon – друг разработчика

Если честно, то с моей стороны не совсем порядочно рассказывать об этом замечательном модуле так поздно. В свое оправдание могу сказать, что я и сам обратил на него внимание довольно поздно, а до этого изрядно помучился.

О чем я? Все просто. Процесс разработки на платформе Node.js, на микроуровне, заключался в следующих итерациях:

- пишем код;
- «гасим» скрипт-сервер;
- запускаем его снова;
- проверяем работу;
- пишем код;
- «гасим» скрипт-сервер
- и т. д.

Надоело? Мне да. И, слава богу, не только мне, но и программисту Реми Шарпу (Remy Sharp), создавшему nodemon.

Nodemon – это скрипт-монитор, отслеживающий изменения в файлах и перезапускающий сервер при их наличии. Все довольно просто, давайте попробуем. Сначала установим модуль:

```
$ npm install -g nodemon
```

Теперь возьмем небольшой Node.js-сценарий (server.js):

```
var http = require('http');
http.createServer(function (request, response) {
   console.log("HTTP works!");
   response.writeHead(200, {'Content-Type': 'text/html'});
   response.end('<h1>Hello!</h1>');
}).listen(8080);
```

Сейчас запустим его, но не просто так, а посредством nodemon:

```
$ nodemon server.js
9 Jun 19:52:11 - [nodemon] v1.0.15
9 Jun 19:52:11 - [nodemon] to restart at any time, enter `rs`
9 Jun 19:52:11 - [nodemon] watching: *.*
9 Jun 19:52:11 - [nodemon] starting `node server.js`
```

Пусть скрипт работает, а мы тем временем немного изменим его исходный код:

```
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.end('<h1>Hello nodemon!</h1>');
}).listen(8080);
```

В консоли немедленно последует реакция:

```
9 Jun 19:57:46 - [nodemon] restarting due to changes...
9 Jun 19:57:46 - [nodemon] starting `node server.js`
```

Nodemon уведомил нас, что перезапустил сервер с внесенными изменениями (в этом можно убедиться, открыв url http://localhost:8080 браузером). Очень мило с его стороны, правда? Теперь изменим код несколько деструктивно:

```
var http = require('http');
https.createServer(function (request, response) {
```

Мы сделали намеренную ошибку в имени переменной, посмотрим, что скажет на это nodemon:

```
9 Jun 20:06:54 - [nodemon] restarting due to changes...
9 Jun 20:06:54 - [nodemon] starting `node server.js`

/home/sukhov/node/server.js:2
https.createServer(function (request, response) {
^
ReferenceError: https is not defined
    at Object.<anonymous> (/home/sukhov/node/server.js:2:1)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Module.runMain (module.js:492:10)
    at process.startup.processNextTick.process._tickCallback

(node.js:244:9)
9 Jun 20:06:54 - [nodemon] app crashed - waiting for file changes before starting...
```

Отлично! Ошибка при перезапуске скрипта не заставила nodemon прекратить работу. Он просто ждет, когда мы ее поправим. После этого все будет, как раньше:

```
9 Jun 20:15:13 - [nodemon] restarting due to changes...
9 Jun 20:15:13 - [nodemon] starting `node server.js`
```

Вот так все просто. Если требуется перезапустить скрипт самостоятельно, не следует прерывать работу nodemon, достаточно применить команду, которую он сам подсказывает при начале работы:

```
rs

10 Jul 13:20:05 - [nodemon] starting `node server.js`
```

Если мы хотим отслеживать изменения во всех файлах приложения, то можно дать задание nodemon наблюдать за всеми файлами в указанной папке:

```
$ nodemon --watch node node/server.js
9 Jun 20:32:15 - [nodemon] v1.0.15
9 Jun 20:32:15 - [nodemon] to restart at any time, enter `rs`
9 Jun 20:32:15 - [nodemon] watching: /home/sukhov/node/**/*
9 Jun 20:32:15 - [nodemon] starting `node node/server.js`
```

По умолчанию nodemon и так отслеживает изменения всех скриптов в той папке, в которой запущен. Мы можем указать ему другую цель, задав определенную папку. Параметры работы nodemon можно указывать в качестве ключей, при запуске. Например:

```
nodemon -e js,jade,styl
```

Так мы даем указание демону мониторить файлы с расширениями js, jade, styl (сценарии JavaScript, шаблоны jade и файлы stylus):

```
nodemon --ignore lib/ --ignore tests/ --ignore param.js
```

Будут освобождены от мониторинга файлы в папках lib/,tests/ и сценарий param.js.

```
demon --delay 2500ms server.js
```

Приложение server.js будет перезапущено через 2500 миллисекунд.

Настройки nodemon можно задать в специальном json-файле nodemon.json, они могут быть как глобальными, так и локальными. Пример такого файла:

```
{
  "restartable": "rs",
  "ignore": [
    ".git",
    "*.test.js",
    "fixtures/*"
  ],
  "verbose": true,
  "execMap": {
    "js": "node --harmony"
  },
  "watch": [
    "app/",
    "node/"
```

```
  ],
  "env": {
    "NODE_ENV": "development"
  },
  "ext": "js json jade"
}
```

Тут, думаю, все понятно, verbose – режим вывода информации, true – его значение по умолчанию.

Nodemon – это модуль Node.js, соответственно, и применять его можно как модуль, управляя запуском приложений из сценария (файл **demon.js**):

```
var nodemon = require('nodemon');

nodemon({
  script: 'server .js',
  ext: 'js json'
});

nodemon.on('start', function () {
  console.log('App has started');
}).on('quit', function () {
  console.log('App has quit');
}).on('restart', function (files) {
  console.log('App restarted due to: ', files);
});
```

Результат:

```
$ node daemon
App has started
App restarted due to:  [ '/home/sukhov/node/server.js' ]
App has started
```

Под nodemon можно запускать Node-приложения, написанные на coffeeScript. Разницы при исполнении не будет никакой:

```
$ nodemon server.coffee
10 Jul 13:38:55 - [nodemon] v1.2.1
10 Jul 13:38:55 - [nodemon] to restart at any time, enter `rs`
10 Jul 13:38:55 - [nodemon] watching: *.*
10 Jul 13:38:55 - [nodemon] starting `coffee server.coffee`
Server running at http://127.0.0.1:8000/
```

Nodemon может также работать со сценариями perl и python, но об этом, как и об остальных возможностях модуля, лучше прочитать на сайте проекта https://github.com/remy/nodemon/tree/master/doc.

Отладка Node.js-приложений (debug-режим)

Возможностей для отладки Node.js-приложений достаточно. Прежде всего давайте вспомним, что основа платформы – движок V8, который имеет встроенный отладчик. «Достучаться» до него из Node.js довольно просто (буду мучить в этой главе наше Express-приложение):

```
$ node --debug app
Debugger listening on port 5858
Express server listening on port 8080
```

Как видим, V8 Debugger запущен, и мы можем общаться с ним, обращаясь к порту 5858. Если обратиться браузером по адресу http://localhost:5858/, мы поручим примерно такое текстовое сообщение:

```
Type: connectV8-Version: 3.22.24.19
Protocol-Version: 1
Embedding-Host: node v0.11.12
Content-Length: 0
```

Теперь, изучив v8 DebuggerProtocol (не слишком простой), мы можем установить TCP-соединение и заняться взрослой отладкой. Но мы этого делать не будем – у Node.js существует клиент, серьезно упрощающий работу. Собственно, мы им и воспользовались, просто в «штатном» режиме он запускается так:

```
$ node debug app
```

Результат:

```
< Debugger listening on port 5858
connecting to port 5858... ok
break in /home/geol/node/express/myApp/app.js:1
> 1 var express = require('express');
  2 var stylus = require('stylus');
  3 var routes = require('./routes');
debug>
```

Совсем другое дело! Тут все интуитивно понятно – запущена пошаговая отладка, и сейчас мы находимся на позиции первой строки программы. Первое, что приходит в голову, – набрать команду next:

```
debug> next
break in /home/geol/node/express/myApp/app.js:2
  1 var express = require('express');
> 2 var stylus = require('stylus');
```

```
  3 var routes = require('./routes');
  4 var map = require('./approutes');
debug>
```

Все, как и ожидалось. Теперь попробуем другую команду:

```
debug> step
break in module.js:373
 371
 372   function require(path) {
>373     return self.require(path);
 374   }
 375
debug>
```

Тут мы оказывались внутри метода requare. Чтобы просмотреть все команды debug-режима, наберем команду **help**:

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o), backtrace (bt), setBre
akpoint (sb), clearBreakpoint (cb),
watch, unwatch, watchers, repl, restart, kill, list, scripts, breakOnException,
breakpoints, version
```

Команд немного, и если вам знаком процесс отладки, почти все должно быть понятно. Прежде всего команды пошагового прохождения программы:

- **next, n** – шаг вперед;
- **step, s** – шаг «внутрь» конструкции (функции, модуля и т. д.);
- **out, o** – шаг «наружу» из конструкции (step out);
- **cont, c** – выполнение программы до точки останова (breakpoints, о них позже);
- **pause** – прерывание исполнения потока программы.

Для иллюстрации применения контрольных точек напишем следующий бесполезный код (debug.js):

```
console.log("Start");
var taget =  "world";
setTimeout(function () {
  debugger;
  console.log(taget);
}, 1000);
console.log("hello");
```

Тут debugger – контрольная точка, на которой исполнение программы должно приостановиться. Хотя при обычном выполнении Node.js этого оператора просто не заметит:

```
$ node debug.js
Start
hello
world
```

Теперь запустим его в режиме отладки:

```
$ node debug debug.js
< Debugger listening on port 5858
connecting to port 5858... ok
break in /home/node/debug/debug.js:1
> 1 console.log("Start");
  2 var taget =  "world";
  3 setTimeout(function () {
debug> cont
< Start
< hello
break in /home/node/debug/debug.js:4
  2 var taget =  "world";
  3 setTimeout(function () {
> 4   debugger;
  5   console.log(taget);
  6 }, 1000);
debug> cont
< world
program terminated
program terminated
```

Для работы с контрольными точками предназначены специальные команды debug-режима:

- **setBreakpoint()**, **sb()** – установить контрольную точку на текущую строку;
- **setBreakpoint(line)**, **sb(line)** – установить контрольную точку на line;
- **setBreakpoint('fn()')**, **sb(...)** – установить контрольную точку на первую строку функции;
- **setBreakpoint('script.js', 1)**, **sb(...)** – установить контрольную точку на первую строку файла **script.js**;
- **clearBreakpoint**, **cb(...)** – убрать контрольную точку.

Остальные команды debug-режима:

- **repl** – открывает **REPL()**-режим. Зачем? Например, узнать контекст:

```
break in /home/node/debug/debug.js:4
  2 var taget =  "world";
  3 setTimeout(function () {
```

```
> 4   debugger;
  5   console.log(taget);
  6 }, 1000);
debug> repl
Press Ctrl + C to leave debug repl
> console.log(taget)
< world
> 1+2
3
>
```

- **backtrace, bt** – команда обратной трассировки (вызовов списка текущих активных вызовов функции). Посмотрим ее работу, вернувшись к Express-приложению:

```
break in /home/node/express/myApp/app.js:39
 38 var prefixes = ['bands'];
>39 prefixes.forEach(function(prefix) {
 40 map.route(app, prefix);
 41 });
debug> backtrace
#0 app.js:39:10
debug> next
debug> next
break in node.js:209
 207   startup.globalConsole = function() {
 208     global.__defineGetter__('console', function() {
>209       return NativeModule.require('console');
 210     });
 211   };
debug> backtrace
#0 node.js:209:27
#1 index.js:13:2
#2 Module._compile module.js:449:26
#3 Module._extensions..js module.js:467:10
#4 Module.load module.js:349:32
#5 Module._load module.js:305:12
#6 Module.require module.js:357:17
#7 module.js:373:17
#8 index.js:3:24
#9 Module._compile module.js:449:26
debug>
```

- **run** – запуск скрипта;
- **restart** – перезапуск скрипта;
- **kill** – завершение потока исполнения скрипта.

Консольный клиент отладчика предоставляет все необходимые возможности и довольно удобен в использовании. Но есть и более удобные средства.

Node Inspector – отладка на стороне клиента

Инструмент отладки, о котором речь пойдет ниже, работает на стороне клиента, причем прямо из браузера. Правда, браузер этот должен быть непременно на движке WebKit/Blink (браузеры Chrome/Chromium, Safari, Opera 15+), так что если вы, как и я, привеженец рыжей лисы, принципами придется поступиться. И оно стоит того!

Сначала устанавливаем новый модуль – node-inspector:

```
$ npm install -g node-inspector
```

Теперь запускаем наше приложение в режиме V8 отладчика:

```
$ node --debug app
Debugger listening on port 5858
Express server listening on port 8080
Connected to DB!
```

Сейчас в другой консоли (можно открыть отдельное окно терминала) запустим node-inspector:

```
$ node-inspector
Node Inspector v0.7.4
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

Теперь осталось только делать, что нам говорят, – заходим браузером по адресу http://127.0.0.1:8080/debug?port=5858, получаем отладчик с GUI, не уступающий какой-нибудь современной IDE (рис. 38).

Что же, похоже, что с отладкой у платформы Node.js действительно все в порядке. Что там дальше, после отладки?

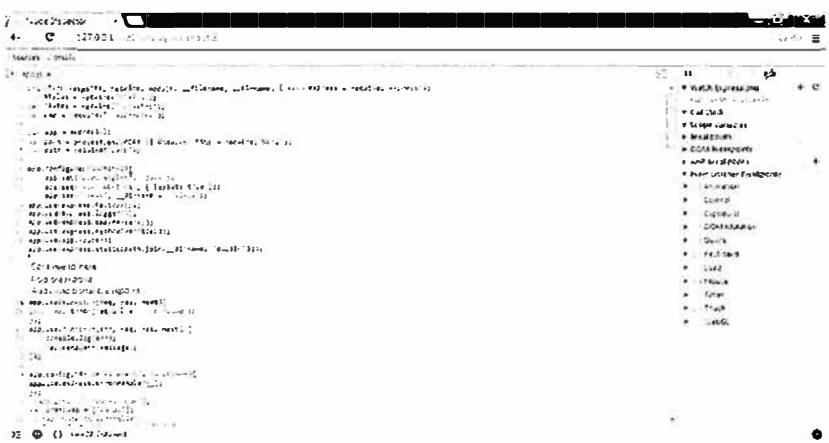

Рис. 38 ❖ Node inspector – настоящая IDE в браузере

Тестирование Node.js-приложений

Тестирование программного обеспечения вообще и веб-приложений в частности – это отдельная большая и многогранная тема. Функциональное, нагрузочное, регрессивное, модульное, интеграционное... Уже одно перечисление терминов показывает объем этой темы. Мы не будем пытаться объять почти необъятное, поговорим только об одном аспекте этого процесса – модульное, или Unit, тестирование. Именно этот тип тестирования обычно выполняется разработчиком.

Что такое Unit-тестирование?

Сначала немного определимся с методологией. Что вообще представляет собой этот процесс? Unit-тестирование (Unit testing) – это изолированная проверка каждого отдельного модуля программного кода путем запуска тестов в искусственной среде. Идея заключается в том, что, оценивая каждый модуль изолированно и подтверждая корректность его работы, точно установить проблему значительно проще, чем если бы элемент был частью системы. Для проведения Unit-тестов необходимо писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к появлению ошибок, облегчает обнаружение и устранение таких ошибок.

Что такое модуль применительно к unit-тестированию Node.js-приложений? В самом простом случае это функция, которая всегда возвращает некоторый результат для некоторого параметра или группы параметров. Причем этот простой случай наиболее желаем: цель модульного тестирования – изолировать отдельные части программы и проверить их работоспособность.

Модульные тесты – это основной инструмент разработки через тестирование (Test-Driven Development).

TDD/BDD

Разработка через тестирование (TDD, Test-Driven Development) – техника программирования, при которой модульные тесты для программы пишутся до самой программы и являются движущей (Driven) силой разработки.

Разработка через тестирование подразумевает, что перед написанием кода необходимо:

- создать тест, который будет тестировать еще не созданный код, тест, соответственно, не пройдет;
- написать код и убедиться, проверить прохождение теста;
- модифицировать код (в случае необходимости) или привести его к стандартам (рефакторинг);
- повторить.

Важный момент: такая методология ориентирована на автоматизированное тестирование (то есть использующее программные средства для выполнения тестов и проверки результатов выполнения). При этом происходит «инверсия ответственности»: от логики тестов и их качества зависит, будет ли код соответствовать техническому заданию. BDD (Behavior Driven Development) – это модификация TDD, как и в TDD, вы пишете тесты до того, как появится код, но подход при этом к ним принципиально другой. BDD ориентирован на тестирование поведения, тогда как TDD вы проверяете конструкции самого кода. Разница в следующем: при TDD-разработке мы проверяем, возвращает ли функция **saveRequestData()** значение true при получении аргументов x, y и z. При BDD мы проверяем, сохраняются ли в базе данных результаты запроса при определенном флаге. При этом нам, скорее всего, тоже надо будет тестировать функцию saveRequestData(), но, согласитесь, подход уже несколько отличается.

Мы должны думать не функциями и возвращаемыми значениями, а поведением тестируемой сущности.

TDD оперирует unit-тестами и коллекциями тестовых сценариев (test suite), BDD – спецификациями поведения (описаниями) и «обязательствами».

Ну хватит теории, давайте лучше посмотрим, какие средства для Unit-тестирования и TDD/BDD мы можем использовать в Node.js.

Accert – утвердительный модуль

Модуль Accert, входящий в ядро Node, предназначенный для написания unit-тестов, реализует концепцию тестирования на самом простом уровне – сравнивает результат операции с ожидаемым результатом (accerting – утверждение). Если мы посмотрим содержимое папки /test какого-либо модуля, то с высокой вероятностью встретим там код, написанный с применением accert. Вот, например, сценарий node_modules/curl/node_modules/router/tests/params.js:

```
var assert = require('assert');
var route = require('../index')();
var count = 0;

route.get('/{test}', function(req, res) {
  assert.equal(req.params.test, 'ok')
  assert.ok(req.url in {'/ok':1, '/ok/':1});
  count++;
});
route.get('/{a}/{b}', function(req, res) {
  assert.equal(req.params.a, 'a-ok');
  assert.equal(req.params.b, 'b-ok');
  assert.equal(req.url, '/a-ok/b-ok');
  count++;
});
route.get('/{a}/*', function(req, res) {
  assert.equal(req.params.a, 'a');
  assert.equal(req.params.wildcard, 'b/c');
  assert.equal(req.url, '/a/b/c');
  count++;
});

route({method:'GET', url:'/ok'});
route({method:'GET', url:'/ok/'});
route({method:'GET', url:'/a-ok/b-ok'});
route({method:'GET', url:'/a/b/c'});

assert.equal(count, 4);
```

Тут выполняется ряд проверок – на соответствие значению строковых параметров:

```
assert.equal(req.params.test, 'ok')
assert.equal(req.params.a, 'a');
```

На численное значение:

```
assert.equal(count, 4);
```

В целом большинство методов assert имеет следующий формат:

```
assert.утверждение(ожидаемое_значение, полученное_значение, [сообщение])
```

Если утверждение верно, ничего не происходит, если нет – выбрасывается исключение, озаглавленное текстом сообщения. Пример работы «на пальцах»:

```
test.js:
var assert = require("assert");
var util = require("util")

var arr = [1,2,3];
console.log("test1");
```

```
assert.equal(true, util.isArray(arr));
console.log("test2");
assert.notEqual(true, util.isArray(arr), "Таки массив");
```

Запускаем:

```
$ node test
test1
test2
assert.js:98
  throw new assert.AssertionError({
        ^
AssertionError:  Таки массив
    at Object.<anonymous> (C:\Users\Geol\node\express\myApp\test\test3.js:6:8)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
    at Module.load (module.js:349:32)
    at Function.Module._load (module.js:305:12)
    at Function.Module.runMain (module.js:490:10)
    at startup (node.js:124:16)
    at node.js:803:3
```

Самих методов-утверждений немного, но вполне достаточно для проверок любой сложности:

- **assert.equal()** – проверка на равенство скалярных значений (эквивалент ==);
- **assert.notEqual()** – проверка на неравенство скалярных значений (эквивалент !=);
- **assert.strictEqual()** – проверка на строгое равенство скалярных значений (эквивалент ===);
- **assert.notStrictEqual()** – проверка на строгое неравенство скалярных значений (экивалент !==);
- **assert.deepEqual()** – проверка на равенство сложных величин (массивов, объектов);
- **assert.notDeepEqual()** – проверка на неравенство сложных величин;
- **assert.ifError(value)** – проверка на значение аргумента error.

Еще два утверждения являются своеобразным «синтаксическим сахаром»:

- **assert(value, message)** – то же, что и **assert.equal(true, value, message)**;
- **assert.ok(value)** – то же, что и **assert.equal(true, value)**.

Другие методы:

- **assert.fail()** – метод возбуждает исключение. Он принимает четыре аргумента – ожидаемое значение, полученное значение, сообщение и оператор сравнения. Примерно так:

```
var assert = require("assert");
var arr = [1,2,3];
try {
    console.log('test');
    assert.fail(5, arr[0], 'Wrong number!', '==');
} catch(e) {
    console.log(e);
}
```

Результат:

```
$ node test
test
{ [AssertionError: Wrong number!]
  name: 'AssertionError',
  actual: 5,
  expected: 1,
  operator: '==',
  message: 'Wrong number!',
  generatedMessage: false }
```

И еще два утверждения:

- **assert.throws()** – блок кода выбрасывает исключение;
- **assert.doesNotThrow()** – блок кода не выбрасывает исключения.

Для пояснения их работы напишем следующий код (**test.js**):

```
var assert = require("assert");
console.log("test1");
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /value/
);
console.log("test2");
assert.notThrows(
  function() {
    throw new Error("Wrong value");
  }
);
```

Результат:

```
$ test.js:10
assert.notThrows(
       ^

TypeError: Object function ok(value, message) {
  if (!value) fail(value, true, message, '==', assert.ok);
} has no method 'notThrows'
    at Object.<anonymous> (C:\Users\Geol\node\express\myApp\test\test3.js:10:8)
    at Module._compile (module.js:449:26)
```

```
    at Object.Module._extensions..js (module.js:467:10)
    at Module.load (module.js:349:32)
    at Function.Module._load (module.js:305:12)
    at Function.Module.runMain (module.js:490:10)
    at startup (node.js:124:16)
    at node.js:803:3
```

Assert – отлично работающий модуль. Он входит в ядро Node.js и поэтому универсален. Но в то же время он слишком низкоуровневый, и удобство использования его оставляет желать лучшего. Понятно, что разработчики не могли это просто так оставить, и для модульного тестирования было создано много отличных инструментов, с некоторыми из них мы сейчас познакомимся.

Should – BDD-style тестовый фреймворк

На самом деле этот фреймворк больше похож на библиотеку утверждений – богатую библиотеку. Модуль предоставляет синтаксис для осуществления самых разнообразных проверок. Давайте установим его и посмотрим, на что он способен:

```
npm install should
```

Теперь напишем небольшой тест:

```
var should = require('should');
var fibbonacci = require('./fibbonacci.js').fibbonacci;
var band = (     name: 'The Beatles',
              members: ['John', 'Paul','George','Ringo']
      }
band.should.be.an.instanceOf(Object);
band.should.have.property('name', 'The Beatles');
band.should.have.property('members').with.lengthOf(4);

(5).should.be.exactly(5).and.be.a.Number;
```

Тут все должно пройти без ошибок:

```
$ node should.js

$
```

Мне кажется, все понятно без объяснений. Но я зануда и поэтому их дам.

Цепочки утверждений

Проверки состоят из цепочек утверждений, связанных точечной нотацией. Все утверждения – методы объекта should, их можно связывать и комбинировать. Для того чтобы эти связи были просты и

удобочитаемы, можно использовать специальные хелперы: .an, .of, .a, .and, .be, .have, .with, .is, .which:

```
band.should.be.an.instanceOf(Object).and.have.property('name', 'The Beatles');
band.members.should.be.instanceof(Array).and.have.lengthOf(4);
```

При создании цепочек надо проявлять известную осторожность. Большинство утверждений возвращают объект, но не все, например .lengthOf – число, и при комбинировании утверждений следует это учитывать.

Далее перечислены некоторые из утверждений should:

- **.ok** – утверждает, что цепочка утверждений возвращает «true» в булевом значении JavaScript:

```
true.should.be.ok;
'yay'.should.be.ok;
(1).should.be.ok;
({}).should.be.ok;
```

- **.true** – утверждает, что цепочка утвеждений строго соответствует «true»:

```
true.should.be.true;
'1'.should.not.be.true;
```

- **.false** – утверждает, что цепочка учреждений строго соответствует «false»:

```
false.should.be.false;
(0).should.not.be.false;
```

- **.eql(value)** – утверждает тождественность значений:

```
(5).should.eql(5);
({ foo: 'bar' }).should.eql({ foo: 'bar' });
[1,2,3].should.eql([1,2,3]);
[1, 2, 3].should.eql({ '0': 1, '1': 2, '2': 3 });
```

Последняя строчка примера показывает, что речь идет именно о тождестве – то есть о вычисленном значении, а не о точном равенстве.

- **.equal(value)** и **.exactly(value)** – а вот тут как раз утверждается точное равенство (как ===):

```
(4).should.equal(4);
'test'.should.equal('test');
[1,2,3].should.not.equal([1,2,3]);
(4).should.be.exactly(4);
```

- **.startWith(str) и .endWith(str)** – эти утверждения проверяют тождество начала и конца строки соответственно, с заданной подстрокой:

```
'foobar'.should.startWith('foo');
'foobar'.should.endWith('bar');
```

- **.within(from, to)** – утверждает принадлежность числа заданному диапазону:

```
(5).should.be.within(5, 10).and.within(5, 5);
```

- **.approximately(num, delta)** – утверждает приблизительное равенство (с указанной точностью) для чисел с плавающей точкой. То, чего очень не хватает во многих подобных библиотеках.

```
(99.99).should.be.approximately(100, 0.1);
```

- **.above(num) и .greaterThan(num)** – утверждают, что исходное значение больше, чем указанное:

```
(5).should.be.above(0);
(50).should.be.greaterThan.above(5);
```

- **.below(num) и .lessThan(num)** – утверждают, что исходное значение меньше, чем указанное:

```
(5).should.be.below(6);
(5).should.be.lessThan(500);
```

- **.NaN**, **.Infinity** – утверждают, что значения чисел – NaN и бесконечность соответственно:

```
(undefined + 0).should.be.NaN
(1/0).should.be.Infinity;
```

- **.type(str)** – утверждает, что данный объект имеет определенный тип:

```
band.should.be.type('object');
'test'.should.be.type('string');
```

- **.Object**, **.Number**, **.Array**, **.Boolean**, **.Function**, **.String**, **.Error** – утверждают, что данный объект соответствует конструктору указанного объекта:

```
({}).should.be.an.Object;
(1).should.be.a.Number;
[].should.be.an.Array.and.an.Object;
(true).should.be.a.Boolean;
''.should.be.a.String;
```

- **.property(name[, value])** – утверждает, что указанное свойство существует и имеет определенное значение:

```
user.should.have.property('name');
user.should.have.property('age', 15);
user.should.not.have.property('rawr');
user.should.not.have.property('age', 0);
[1, 2].should.have.property('0', 1);
```

- **.length(number)** и **.lengthOf(number)** – утверждают, что указанное свойство (length) существует и имеет определенное численное значение:

```
user.pets.should.have.length(5);
user.pets.should.have.a.lengthOf(5);
({ length: 10}).should.have.length(10);
```

- **.ownProperty(str)** и **.hasOwnProperty(str)** – утверждает, что объект имеет указанное свойство:

```
({ foo: 'bar' }).should.have.ownProperty('foo').equal('bar');
```

- **.empty** – утверждает, что аргумент не существует или имеет нулевое значение:

```
[].should.be.empty;
''.should.be.empty;
({}).should.be.empty;
(function() {
  arguments.should.be.empty;
})();
```

- **.keys()** – утверждает, что объект имеет указанный набор ключей:

```
var obj = { foo: 'bar', baz: 'raz' };
obj.should.have.keys('foo', 'baz');
obj.should.have.keys(['foo', 'baz']);
({}).should.have.keys();
```

- **.propertyByPath()** – утверждает, что свойство по указанному пути существует:

```
var obj = { foo: 'bar',  base: { a: {b:10} } };
obj.should.have.propertyByPath('base', 'a');
obj.should.have.propertyByPath('base', 'a', 'b');
```

- **.match(value)** – утверждает соответствие объекта указанному. Тут есть несколько вариантов.

Для строк используются регулярные выражения:

```
band.name.should.match(/^\w+$/);
['a', 'b', 'c'].should.match(/[a-z]/);
```

То же для строковых полей объекта:

```
({ a: 'foo', c: 'barfoo' }).should.match(/foo$/);
({ a: 'a' }).should.not.match(/^http/);
```

Для любых аргументов – функции с возвращаемым значением:

```
(5).should.match(function(n) { return n > 0; });
(5).should.not.match(function(n) { return n < 0; });
(5).should.not.match(function(it) { it.should.be.an.Array; });
(5).should.match(function(it) { return it.should.be.a.Number; });
```

Два объекта сравниваются по соответствию набора свойств:

```
({ a: 10, b: 'abc', c: { d: 10 }, d: 0 }).should
    .match({ a: 10, b: /c$/, c: function(it) { return it.should.have.
property('d', 10); }});

[10, 'abc', { d: 10 }, 0].should
    .match({ '0': 10, '1': /c$/, '2': function(it) { return it.should.
have.property('d', 10); } });
```

- **.matchEach(otherValue)** – утверждает соответствие объектов в массиве:

```
(['a', 'b', 'c']).should.matchEach(/[a-c]/);
[10, 11, 12].should.matchEach(function(it) { return it >= 10; });
[10, 11, 12].should.matchEach(function(it) { return it >= 10; });
```

- **.throw()** и **throwError()** – утверждает «выбрасывание» исключения:

```
(function(){
  throw new Error('fail');
}).should.throw();
```

Пожалуй, хватит. Остальное можно узнать из документации. Только еще две управляющие структуры для полного счастья:

- **.not** – отрицает текущее утверждение.

```
should(null).not.be.ok;
```

- **.any** – позволяет утверждения с многократными параметрами, возвращает истину, если сработает один из них (подобно родному JavaScript **array.some()**):

```
band.should.any.have.property('name', 'The Beatles','The Kinks');
```

Chai – BDD/TDD-библиотека утверждений

Chai предоставляет несколько интерфейсов, которые позволяют разработчику выбирать наиболее подходящий. Стилей всего три, и сейчас мы с ними кратко познакомимся. Для этого сначала установим этот модуль:

```
$ npm install chai
```

Chai TDD-стиль

Утверждающий стиль реализован через assert-интерфейс. То есть обычные TDD-утверждения, с обращением к объекту assert посредством точечной нотации:

```
var assert = require('chai').assert
var foo = 'bar'
var band = { name    : "Muse",
             members: [ 'Matthew', 'Christopher', 'Dominic' ] };

assert.typeOf(foo, 'string', 'foo is a string');
assert.equal(foo, 'bar', 'foo equal `bar`');
assert.lengthOf(foo, 3, 'foo`s value has a length of 3');
assert.lengthOf(band.members, 3, band.name+' has 3 members');
```

В общем, все очень похоже на «родной» node.js-модуль assert, но с некоторыми дополнительными утверждениями, «синтаксическим сахаром» и, самое главное, возможностью писать свои проверки:

```
assert('foo' !== 'bar', 'foo is not bar');
assert(Array.isArray([]), 'empty arrays are arrays');
```

Chai BDD

Chai-тесты в BDD-стиле используют два подхода, которые можно разделить по используемым интерфейсам – expect() и should().

Оба для построения утверждений используют тот же самый цепочечный принцип, но они отличаются по начальному утверждению. Expect() требует просто ссылку на ожидаемую функцию, should() – на ее выполнение.

Expect

Проверка с интерфейсом expect() будет выглядеть следующим образом:

```
var expect = require('chai').expect;
var foo = 'bar';
var band = { name    : "The Verve",
```

```
            members: [ 'Richard', 'Nick', 'Simon', 'Peter' ] };

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(band).to.have.property('members').with.length(4);
```

Expect() позволяет дополнять своими сообщениями каждое утверждение:

```
var answer = 43;
expect(answer, 'Duglas Adams say 42!').to.equal(42);
```

Результат:

```
$ node chai

/home/sukhov/node_modules/chai/lib/chai/assertion.js:106

      throw new AssertionError(msg, {

            ^

AssertionError: Duglas Adams say 42!: expected 43 to equal 42

...
```

Should

Интерфейс should() использует почти те же самые цепочки, отличие состоит в том, что утверждения распространяются на каждый объект:

```
var should = require('chai').should();
var foo = 'bar';
var band = { name    : "Thirty Seconds to Mars",
             members:[ 'Jared', 'Shannon', 'Tomo' ] };
foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.length(3);
band.should.have.property('members').with.length(3);
```

И TDD-, и особенно BDD-стили имеют свои небедные API и свои настройки, с которыми лучше подробно ознакомиться в документации. А мы сейчас рассмотрим несколько инструментов более высокого уровня.

Mocha – JavaScript тест-фреймворк

Mocha – фреймворк, позволяющий писать и запускать unit-тесты, объединять их в наборы (suite), генерировать отчеты и создавать из

тестов документацию. Он может работать как на сервере, так и на клиентской стороне (прямо на html-странице). Подробным разбором Mocha (кстати, произносится это вполне благозвучно – «Мока») и приемов работы с ним мы заниматься не будем (этот процесс может затянуться, лучше обратиться к объемной документации), просто посмотрим фреймворк в работе на небольшом примере.

Сначала все, как обычно:

```
$ npm install mocha
```

Начинаем работать с Mocha

Теперь напишем небольшой Node.js-модуль (**fibbonacci.js**):

```
function fibbonacci(n)
{
    if (n < 3) {
        return 1;
    }
    else {
        return fibbonacci(n-1) + fibbonacci(n-2);
    }
}

module.exports.fibbonacci = fibbonacci;
```

Сейчас создадим папку test и в ней файл **test.js**:

```
var assert = require('assert');
var fibbonacci = require('../fibbonacci.js').fibbonacci;
describe('ibbonacci', function(){
         it('fibbonacci check1', function(){
             assert.equal(89, fibbonacci(11));
             assert.equal(5, fibbonacci(5));
             assert.equal(55, fibbonacci(10));
          });
});
```

Конструкции тут очень простые:

- **describe** – определение набора тестов, наборы могут быть вложенными;
- **it** – определение теста внутри любого набора тестов.

Тесты в Mocha создаются внутри блока **describe()**. Отдельные тесты описываются в блоках **it()**. **describe** и **it** являются обычными вызовами функций, которым передаются два параметра. Первый – название группы или теста, второй – функция, содержащая код. Остальное делает уже знакомый нам модуль assert.

Теперь запускаем команду **mocha** без параметров (по умолчанию mocha будет запускать все тесты, расположенные в папке test):

```
$ mocha  --reporter spec


  Fibbonacci
    √ fibbonacci check1


  1 passing (35ms)
```

Все замечательно, но не интересно. Напишем еще один тест:

```
        assert.equal(5, fibbonacci(5));
        assert.equal(55, fibbonacci(10));
    });
    it('fibbonacci check2', function(){
        assert.equal(1, fibbonacci(1));
        assert.equal(0, fibbonacci(0));
    });
```

Насчет нуля и ряда Фибоначчи я в курсе, но сейчас мы просто тестируем наш скрипт:

```
$ mocha  --reporter spec
  Fibbonacci
    √ fibbonacci check
    1) fibbonacci check2


  1 passing (13ms)
  1 failing

  1) Fibbonacci fibbonacci check2:
     AssertionError: 0 == 1
      at Context.<anonymous> (C:\Users\Geol\node\express\myApp\test\test.js:11
21)
      at callFn (C:\Users\Geol\AppData\Roaming\npm\node_modules\mocha\lib\runna
le.js:223:21)
.................................
```

Теперь все хорошо – второй тест успешно (и заслуженно) провален.

Разумеется, это очень простой пример. Можно писать тесты любой сложности, используя вложенные блоки describe().

Кроме «нативного» модуля Assert, Mocha для реализации утверждений может использовать вышеописанные модули Should и Chai, а также библиотеки expect.js и better-assert.

Дополнительная возможность – можно использовать хуки-тригеры: before(), after(), beforeEach() и afterEach(). Например, так:

```
var assert = require('assert');
var fibbonacci = require('./fibbonacci.js').fibbonacci;
describe('Fibbonacci', function(){
      var testNumber;
      before(function() {
          console.log("startSuit");
           testNumber = 0;
      });
      after(function() {
          console.log("endSuit");
      });
        beforeEach(function() {
          testNumber++;
          console.log("startTest "+ testNumber);
      });
      afterEach(function() {
          console.log("endTest "+ testNumber);
      });
         it('fibbonacci check1', function(){
             ……
         });
         it('fibbonacci check2', function(){
             ……
         });
         it('fibbonacci check3', function(){
             ……
         });
});
```

Результат:

```
$ mocha  --reporter spec mocha.js
  Fibbonacci

startSuit

startTest 1

    √ fibbonacci check1

endTest 1

startTest 2

    √ fibbonacci check1

endTest 2

startTest 3
```

```
    √ fibbonacci check1

endTest 3
endSuit

  3 passing (17ms)
```

Весь код, который мы писали до того, использовал BDD-интерфейс, но то не единственный интерфейс Mocha. Специально для традиционалистов существует и интерфейс TDD, правда, код, написанный с его помощью, будет практически идентичен написанному с помощью BDD. Различие только в названии методов. Вот каким образом будет выглядеть наш пример с числами Фибоначчи, переписанный в TDD-стиле:

```
var assert = require('assert');
var fibbonacci = require('./fibbonacci.js').fibbonacci;
suite('Fibbonacci', function(){
      setup(function() {
          console.log("startSuit");
      });
      teardown(function() {
          console.log("endSuit");
      });
      test('fibbonacci check1', function(){
           assert.equal(89, fibbonacci(11));
      });
      test('fibbonacci check1', function(){
           assert.equal(0, fibbonacci(0));
      });
});
```

Кроме этого, Mocha реализует следующие интерфейсы:

Exports

Этот интерфейс достался Mocha от предшественника – фреймворка expresso. Нельзя сказать, что он очень изящный, но работать можно:

```
module.exports = {
     before: function(){
     },
     'Array': {
         '#indexOf()': {
            'should return -1 when not present': function(){
               [1,2,3].indexOf(4).should.equal(-1);
            }
         }
     }
};
```

QUnit

Этот интерфейс был инспирирован другим популярным фреймворком – Qunit. Со всеми его синтаксическими особенностями:

```
function ok(expr, msg) {
  if (!expr) throw new Error(msg);
}

suite('Array');

test('#length', function(){
  var arr = [1,2,3];
  ok(arr.length == 3);
});

test('#indexOf()', function(){
  var arr = [1,2,3];
  ok(arr.indexOf(1) == 0);
  ok(arr.indexOf(2) == 1);
  ok(arr.indexOf(3) == 2);
});

suite('String');

test('#length', function(){
  ok('foo'.length == 3);
});
```

Впрочем, все это достаточно интересно, но мы (как и большинство пользователей Mocha) продолжим с BDD-интерфейсом.

Асинхронный код

Особый случай – тестирование асинхронного кода. Тут существует очень удобный механизм передачи дополнительного параметра – функции обратного вызова, получающей управление в случае успеха теста. На практике это выглядит так:

```
describe('User', function(){
  describe('#save()', function(){
    it('should save without error', function(done){
      var user = new User('Luna');
      user.save(function(err){
        if (err) throw err;
        done();
      });
    })
  })
});
```

Асинхронность кода не мешает применять хуки:

```
describe('Connection', function(){
  var db = new Connection
    , tobi = new User('tobi')
    , loki = new User('loki')
    , jane = new User('jane');

  beforeEach(function(done){
    db.clear(function(err){
      if (err) return done(err);
      db.save([tobi, loki, jane], done);
    });
  })

  describe('#find()', function(){
    it('respond with matching records', function(done){
      db.find({ type: 'User' }, function(err, res){
        if (err) return done(err);
        res.should.have.length(3);
        done();
      })
    })
  })
})
```

В составе фреймворка Mocha еще много интересных вещей, например неплохой механизм вывода отчетов, но мы не будем задерживаться и пойдем дальше.

Jasmine – ароматный BDD-фреймворк

Жасмин – BDD-фреймворк для тестирования JavaScript-кода. Он предназначен для использования как на сервере, так и в браузере, имеет все продвинутые возможности такого рода тестовых сред и вместе с тем обладает простым и ясным синтаксисом.

Jasmine приятен тем, что поддерживает несколько типов отчетов, например **HtmlReporter**, компилирующий HTML-страницу с древовидной структурой с прогресса выполнения прохождения тестов.

Основы работы с Jasmine

Нас, понятно, сейчас не интересует браузерная сторона применения jasmine, а вот на стороне сервера мы его попробуем.

Устанавливаем фреймворк:

```
npm install jasmine-node -g
```

Опробуем его работу на примере из руководства (**testspec.js**):

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Теперь набираем в консоли команду jasmine-node:

```
$ testspec.js
.

Finished in 0.048 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

Все работает!

Для начала тестирования создадим папку spec/, именно там jasmine по умолчанию ищет тесты. Теперь поместим туда тот же тест, что мы использовали для демонстрации работы Mocha, сохраним его под именем **test1.spec.js** (именно такой формы должно быть имя jasmine-теста) и запустим jasmine:

```
$ jasmine-node spec/test1.spec.js

.F

Failures:


  1) ibbonacci fibbonacci check2

   Message:

     AssertionError: 0 == 1

   Stacktrace:

     AssertionError: 0 == 1

    at null.<anonymous> (/home/sukhov/node/https/wm/spec/test1.spec.js:11:21)

Finished in 0.007 seconds

2 tests, 1 assertion, 1 failure, 0 skipped
```

Все, что мы написали для mocha, работает и здесь, это неудивительно, синтаксис на уровне конструкций **description()** и **it()** совместим. Зато в jasmin можно обойтись без assert. Например, так (**test1.spec.js** в jasmin-стиле):

```
var fibbonacci = require('../fibbonacci.js').fibbonacci;
describe('ibbonacci', function(){
        it('fibbonacci check1', function(){
            expect(fibbonacci(11)).toEqual(89);
         expect(fibbonacci(5)).toEqual(5);
         expect(fibbonacci(10)).toEqual(55);
         });
      it('fibbonacci check2', function(){
            expect(fibbonacci(1)).toEqual(1);
         expect(fibbonacci(0)).toEqual(0);
         });
});
```

Результат выполнения теста будет точно таким же, поэтому я его опускаю. Методы Jasmine почти идентичны методам Mocha:

- **describe** – определение набора тестов (наборы могут быть вложенными друг в друга);
- **it** – определение теста внутри любого набора тестов;
- **expect** – определяет ожидания, которые проверяются в тесте.

Jasmine имеет вполне достаточный набор ожиданий для различных тестов. Вот некоторые из них:

- **.toBe()** – строгое сравнение;
- **.toEqual()** – сравнение;
- **.toBeUndefined()** – значение должно быть не определено;
- **.toBeNull()** – значение должно быть null;
- **.toBeGreaterThan(n)** – значение должно быть больше, чем n;
- **.toContain(n)** – массив/подстрока должны содержать n:

```
expect([1, 2, 3]).toContain(2)
```

- **.toBeCloseTo()** – сравнение с указанной точностью:

```
expect(3.14).toBeCloseTo(3.1, 1)
```

- **.toMatch()** – соответствие регулярному выражению.

Еще две команды предназначены для отключения действующих тестов:

- **xdescribe** – отключает набор тестов;
- **xit** – отключает конкретный тест.

Также можно использовать хуки:

```
var fibbonacci = require('../fibbonacci.js').fibbonacci;
describe('ibbonacci', function(){
      beforeEach(function() {
          console.log("start");
      });
```

```
    afterEach(function() {
        console.log("end");
    });
       it('fibbonacci check1', function(){

           expect(fibbonacci(11)).toEqual(89);
           expect(fibbonacci(5)).toEqual(5);
           expect(fibbonacci(10)).toEqual(55);
        });
      it('fibbonacci check2', function(){
           expect(fibbonacci(1)).toEqual(1);
        });
});
```

Результат:

```
$ jasmine-node spec/test1.spec.js

start

end

.start

end
.

Finished in 0.005 seconds

2 tests, 4 assertions, 0 failures, 0 skipped
```

Jasmine и асинхронный код

Тестировать асинхронный код тоже несложно. Для этого набросаем небольшой http-сервер (**server.js**):

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello Jasmine\n');
}).listen(8080, '127.0.0.1');
```

Запустим сервер, пусть он работает. Теперь пишем сам тест:

```
var request = require('request');

describe('jasmine-node', function(){
    it("should respond with Hello Jasmine", function(done) {
      request("http://localhost:8080/", function(error, response, body){
       expect(body).toEqual('Hello Jasmine');
        done();
      });
    });
});
```

Запускаем:

```
$ jasmine-node spec/test2.spec.js

F

Failures:

  1) jasmine-node should respond with Hello Jasmine

   Message:

     Expected 'Hello Jasmine

' to equal 'Hello Jasmine'.

   Stacktrace:

     Error: Expected 'Hello Jasmine

' to equal 'Hello Jasmine'.

    at Request._callback (/home/sukhov/node/https/wm/spec/test2.spec.js:6:21)

    at Request.init.self.callback
    ...

Finished in 0.018 seconds

1 test, 1 assertion, 1 failure, 0 skipped
```

Провал! Но, по крайней мере, из сообщения ясно, из-за чего – мы забыли знак перевода строки, поправим:

```
expect(body).toEqual('Hello Jasmine\n');
```

Теперь все в порядке:

```
$ jasmine-node spec/test2.spec.js

.

Finished in 0.015 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

Spies – шпионим и эмуллируем

В составе Jasmine есть замечательный механизм для эмуляции функций и объектов. Отслеживание вызова функции и параметров вызова осуществляется с помощью метода **spyOn()**. Ему передаются два параметра – объект, для которого осуществляется вызов функции, и имя функции, которую необходимо отслеживать:

```
spyOn(object, 'functionName')
```

Для того чтобы показать работу этого механизма, напишем небольшой объект в хуке **beforeEach()** (пример из руководства Jasmine):

```
describe("A spy", function() {
  var foo, bar = null;
  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      }
    };
    spyOn(foo, 'setBar');

    foo.setBar(123);
    foo.setBar(456, 'another param');
  });
```

Вот такое замыкание. Теперь пишем тесты.

При тестировании с использованием **spyOn()** можно отслеживать количество вызовов, их параметры и каждый вызов в отдельности:

```
it("tracks that the spy was called", function() {
  expect(foo.setBar).toHaveBeenCalled();
});

it("tracks its number of calls", function() {
  expect(foo.setBar.calls.length).toEqual(2);
});

it("tracks all the arguments of its calls", function() {
  expect(foo.setBar).toHaveBeenCalledWith(123);
  expect(foo.setBar).toHaveBeenCalledWith(456, 'another param');
});

it("allows access to the most recent call", function() {
  expect(foo.setBar.mostRecentCall.args[0]).toEqual(456);
});

it("allows access to other calls", function() {
  expect(foo.setBar.calls[0].args[0]).toEqual(123);
});
it("stops all execution on a function", function() {
  expect(bar).toBeNull();
});
```

Результат:

```
$ jasmine-node jaspec.js
......

Finished in 0.012 seconds
6 tests, 7 assertions, 0 failures, 0 skipped
```

Применяя различные параметры, можно вызвать оригинальную функцию, возвращать из функции определенное значение, вызывать вместо оригинальной функции указанную произвольную.

В Jasmin существует специальный механизм для создания объектов-«заглушек» (Mocks). Эта процедура осуществляется с помощью **createSpyObj()**. В качестве параметров **createSpyObj()** принимает имя объекта и массив строк, являющийся списком методов объекта-заглушки:

```
describe("Multiple spies, when created manually", function() {
  var tape;

  beforeEach(function() {
    tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);

    tape.play();
    tape.pause();
    tape.rewind(0);
  });

  it("creates spies for each requested function", function() {
    expect(tape.play).toBeDefined();
    expect(tape.pause).toBeDefined();
    expect(tape.stop).toBeDefined();
    expect(tape.rewind).toBeDefined();
  });

  it("tracks that the spies were called", function() {
    expect(tape.play).toHaveBeenCalled();
    expect(tape.pause).toHaveBeenCalled();
    expect(tape.rewind).toHaveBeenCalled();
    expect(tape.stop).not.toHaveBeenCalled();
  });

  it("tracks all the arguments of its calls", function() {
    expect(tape.rewind).toHaveBeenCalledWith(0);
  });
});
```

В этом примере, взятом из руководства, тестированию подвергается некий программный параметр, компонент, вместо которого мы средствами Jasmine подсунули свой, не очень сложный объект.

И где-то здесь мы опять остановимся. Jasmine – замечательный тестовый фреймворк, с богатым функционалом, но его изучение не укладывается в рамки нашего обзора.

Grunt – The JavaScript Task Runner

Как известно, работа JavaScrip-программиста интересна, увлекательна, наполнена яркими находками и блестящими инженерными решениями. Все это так, но, к сожалению, имеет в ней место быть и нудная рутинная работа. Например, для реализации сложной логики на странице оптимально создать несколько файлов-сценариев. Требования же минимализации нагрузки на сервер заставляют объединять их в один файл, да еще и минифицировать его.

Собственно, никаких проблем провести эту процедуру один раз нет, но ведь дальнейшее редактирование получившейся беспробельной простыни определенно представляет проблему. Приходится редактировать исходные файлы и нудно повторять процедуру сборки проекта после каждого изменения.

И это еще не все. Есть еще рутинные процедуры конвертации LESS-файлов, «сшивания» CSS-сценариев и т. п. Конечно, для этих действий созданы различные инструменты автоматизации, но они были различны для каждой процедуры, что совсем не удобно. Отличие инструмента, речь о котором пойдет ниже, – как раз в универсальности, делающей возможным провести полный комплекс действий по сборке JavaScript-проектов «за один проход». Кроме того, Grunt полностью написан на JavaScript (JavaScript-программистами для JavaScript-программистов). Он работает на Node.js и может быть использован на различных программных платформах.

Еще одно преимущество Grunt – возможность организации последовательной обработки ресурсов. Он позволяет задавать все действия в определенном порядке, что уберегает от ошибок и крайне полезно при командной разработке, заставляя каждого разработчика придерживаться единого стандарта.

Итак, Grunt – инструмент для сборки JavaScript-проектов. Это относительно недавняя разработка (2012 г.), которая очень быстро завоевала популярность. Сейчас для Grunt создано множество плагинов/расширений, часто эта платформа включается в стандартный жизненный цикл проектов. Многие мои коллеги впервые проявили интерес к платформе Node.js именно как к средству для работы Grunt.

Grunt – начало работы

Устанавливается Grunt как обычный Node.js-модуль:

```
$ npm install grunt -g
```

Затем отдельно устанавливаем средство для работы с grunt из командной строки:

```
$ npm install grunt-cli -g
```

Для проверки в консоли можно запустить новую команду:

```
$ grunt --version
grunt-cli v0.1.13
grunt v0.4.5
```

Что же, похоже, у нас все замечательно, и можно начинать использовать данный инструментарий. Перейдем в папку проекта, автоматическую сборку которого мы хотим наладить:

```
/app
    /users.js
    /bands.js
/lib
    /jquery.js
    /underscore.js
/images/
/stylus
    /main.styl
    /users.styl
    /bands.styl
/css
    /main.css
    /users.css
    /bands.css
/index.html
/app.js
/index.js
```

Типичное одностраничное веб-приложение, ничего особенного. Создадим в корне сайта файл package.json (да-да, как для обычного node.js-проекта):

```
{
    "author"    : "Geol",
    "name"      : "myProject",
    "version"   : "0.0.1",

    "devDependencies" : {
        "grunt"                 : ">= 0.4",
```

```
        "grunt-cli"              : ">= 0.1.6",
        "grunt-contrib-cssmin"   : ">=0.5.0",
        "grunt-contrib-uglify"   : ">=0.2.0",
        "grunt-contrib-concat"   : ">=0.1.3",
        "grunt-contrib-jshint"   : "~0.10.0",
        "grunt-contrib-watch"    : "*"
    }
}
```

Как вы понимаете, кроме grunt и grunt-cli, мы тут требуем наличия некоторых дополнительных модулей, имеющих префикс grunt-contrib-. Это расширения (плагины) для grunt, в «голом» виде он сам обладает весьма ограниченным функционалом. В данном случае мы хотим подключить:

- **grunt-contrib-cssmin** – минифицирует css-файлы;
- **grunt-contrib-uglify** – минифицирует javaScript-файлы;
- **grunt-contrib-concat** – позволяет соединить файлы;
- **grunt-contrib-jshint** – проверка javaScript-кода.

Что теперь? Все как обычно:

```
$ npm install
```

Тут требуется небольшое пояснение для тех, кто ранее уже работал с модулем grunt, причем с предыдущими версиями. Они могут не понять, к чему это все. Дело в том, что подобную модульность grunt обрел только в версии 0.4.0. До этого у него имелись встроенные команды для основных действий: **concat**, **lint**, **min**, **qunit**, **watch** и т. д. Теперь этот функционал разнесен по плагинам, что безусловно дает большую гибкость, но требует несколько больше начальных телодвижений. Список самых востребованных плагинов можно посмотреть здесь: https://github.com/gruntjs/grunt-contrib-jshint.

После того как все установлено, в корень нашего проекта добавим файл с именем **Gruntfile.js**. Это основной файл для конфигурации Grunt, который содержит список загружаемых задач и параметров:

```
module.exports = function(grunt) {
  grunt.initConfig(
    jshint: {},
    concat: {},
    uglify: {},
    ..........
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-uglify');
```

```
  ...........

  grunt.registerTask('default', ['jshint', 'concat', 'uglify',...]);
};
```

Сначала мы инициализируем конфигурацию **grunt**, в которой задаем настройки его модулям (предварительно установленным). Затем серией вызовов метода **grunt.loadNpmTasks()** мы их же загружаем. Потом методом **grunt.registerTask()** определяем задания, которые будут выполнены (причем в заданном порядке) сразу при запуске команды **grunt**. Нам нужно только запустить эту команду, и все будет сделано. Поехали:

```
$ grunt
>> No "jshint" targets found.
Warning: Task "jshint" failed. Use --force to continue.

Aborted due to warnings.
```

Результат не впечатляет. Дело в том, что я немного преувеличил. На самом деле тут нет волшебства, и любую задачу grunt'а надо настраивать. Сейчас мы этим займемся, попутно разобравшись с работой каждой.

Инструменты Grunt

Самый первый плагин, который мы подключили, jshint, как уже было сказано, проверяет JavaScript-код. В самом общем случае ему надо просто указать проверяемые файлы. Сделаем это:

```
jshint: {
            all: [ 'Gruntfile.js', 'index.js', 'main.js', 'app/*.js']
},
```

«Под раздачу» тут попадает и сам Gruntfile.js – пусть, хуже не будет. Теперь запустим этот плагин (задачи grunt можно запускать по отдельности):

```
$ grunt jshint
Running "jshint:all" (jshint) task

   index.js
     16 |app.set('views', __dirname + '/views')
                                              ^ Missing semicolon.

>> 1 error in 5 files
Warning: Task "jshint:all" failed. Use --force to continue.

Aborted due to warnings.
```

Вот так – grunt заметил пропущенную запятую в файле index.js (честно говоря, я ее пропустил специально для демонстрации, но не суть). Обратите внимание: программа прервала свою работу, что логически верно – ошибки надо исправлять.

Собственно, jshint – это полноценный статический анализатор кода, проверяющий JavaScript-программу в соответствии с заданными правилами. В примере выше мы ни одного параметра не указали, пустив все на самотек (то есть на заданные по умолчанию значения). Это не всегда удобно и не всегда достаточно, поэтому давайте рассмотрим некоторые из них. Параметры в jshint делятся на две группы – «запрещающие» (enforcing) и «позволяющие» (relaxing). Сначала запреты:

- **curly** – параметр обязывает использовать фигурные скобки при написании условий и циклов. JavaScript позволяет их опускать, если блок состоит только из одной строки, но в процессе разработки такие вольности чреваты ошибками;
- **eqeqeq** – параметр запрещает использовать == и != в пользу === и !==. Эти операторы при сравнении пытаются приводить операнды к одному типу, что иногда чревато необычным результатом. Последние не делают приведения, учитывая при сравнении еще и тип операнда, следовательно, являются более безопасными;
- **strict** – параметр требует, чтобы код был написан в соответствии с режимом Stict Mode (строгим режимом). Stict Mode – это нововведение стандарта EcmaScript 5. Он запрещает использование некоторых, излишне свободных конструкций JavaScript, тем самым позволяя избежать появления многих распространенных ошибок;
- **latedef** – параметр творит, по сути, ужасную для JavaScript вещь – он запрещает использование переменных до их объявления;
- **plusplus** – параметр запрещает использование унарных операторов инкремента и декремента (++ и --). Зачем? По-видимому, затесавшиеся в ряды JavaScript-разработчиков «питонисты» считают это плохим стилем программирования.

Ну, хватит запретов. Теперь то, что позволяет расслабиться. Тут надо иметь в виду, что сам по себе, без всяких параметров не позволяет разработчику пользоваться некоторыми JavaScript-вольностями, так что данная часть настроек – это ослабление поводья:

- **asi** – параметр подавляет сообщения об ошибке, вызванной пропущенной точкой с запятой;

- **boss** – параметр подавляет сообщения об ошибке в случае, когда ожидается сравнение, но его не происходит. Например:

```
if (x = 2) {
}
```

- **evil** – параметр подавляет сообщения об ошибке при использовании eval;
- **Sub** – параметр подавляет сообщения об ошибке при использовании нотации [] при обращении к свойствам объекта, когда это свойство можно получить с использованием точечной нотации.

Остается добавить, что «разрешительной» частью настроек следует пользоваться с известной осторожностью.

Плагин **concat** соединяет файлы. Тут все просто:

```
concat: {
    main: {
        src: [
            'lib/*.js',
            'app/*.js',
            'index.js'
            'main.js
        ],
        dest: 'scripts.js'
    }
},
```

То есть все javaScript-файлы из папки app/ и из папки lib/ (а это и jquery, и underscore, и, возможно, еще какие-нибудь библиотеки, которые мы будем подключать в дальнейшем), а также index.js будут склеены в один файл – scripts.js, который, единственный из внешних JavaScript-файлов, будет подключен к html-разметке. Создадим этот файл:

```
$ grunt concat
Running "concat:main" (concat) task
File scripts.js created.
Done, without errors.
```

Теперь займемся минификацией – настройкой плагина **jshint**:

```
uglify: {
    main: {
        files: {
            'scripts.min.js': 'scripts.js'
        }
    }
```

Тут мы сначала указываем файл назначения, затем файл, нуждающийся в минификации. Запустим команду:

```
$ grunt uglify
Running "uglify:main" (uglify) task
File scripts.min.js created: 167.75 kB → 107.87 kB

Done, without errors.
```

Результат не особо впечатляет, но он есть. Мы можем сконфигурировать задание чуть-чуть погибче:

```
uglify: {
         main: {
            files: {
                'scripts.min.js': '<%= concat.main.dest %>'
            }
          }
        }
```

Теперь минификации будет подвергнут результат работы над блоком main плагина concat. Все замечательно, и можно запускать утилиту без параметров:

```
$ grunt
Running "jshint:all" (jshint) task
>> 4 files lint free.

Running "concat:main" (concat) task
File scripts.js created.

Running "uglify:main" (uglify) task
File scripts.min.js created: 167.75 kB → 107.87 kB

Done, without errors.
```

Задача в минимальном приближении решена. Теперь предлагаю рассмотреть, что, собственно, из себя представляет настройка отдельного плагина. В общем случае она осуществляется следующим (декларативным) образом:

```
foo: {
    options: {
        ...
    },
    block1: {
        ...
    },
    block2: {
        src: [
            ....
        ],
        dest: 'someScripts.js'
    }
}
```

Тут foo – некая абстрактная задача. Блок options определяет её некоторые общие параметры. Например, так:

```
concat: {
  options: {
    separator: ';'
  },
```

(Здесь мы назначили общий для всей задачи разделитель файлов для плагина **concat**.)

Далее следуют блоки, смысл определения которых – запускать одну задачу для разных исходных файлов, с индивидуальным набором параметров. Сами файлы могут быть определены просто своим именем с путем от корня папки, в которой находится **Gruntfile.js**. В случае когда задача распространяется на группу файлов, их имена задаются в виде массива. Можно также использовать маску с регулярными выражениями, например:

```
libs/*.js //все файлы с расширением .js из папки libs
app/**/*.js //все файлы с расширением .js из папки app
            // и подпапок любой вложенности
```

Важно, что использование масок подразумевает произвольный порядок при обращении к файлам, поэтому к ошибкам может привести действие вроде этого: concat: {

```
        src: [
            '*.js',
            '*.css'
        ]
}
```

Еще один момент, заставляющий быть аккуратным при использовании масок, продемонстрирован ниже:

```
    jshint: {
              all: ['*.js', 'app/*.js']
      },
  concat: {
            main: {
                src: [
                    '*.js',
                    'app/*.js',
                ],
                dest: 'scripts.js'
            }
        },
```

Данная конфигурация создаст проблемы при втором запуске grunt – целевой файл задачи concat попадает под маску исходных

файлов обеих задач. Я вас предупредил – теперь для вас (в отличие от меня, в начале использования grunt) есть шанс не наступить на эти грабли.

Еще одна рутинная задача, которую почему-то часто вешают на разработчика, – оптимизация графики. И если мы не всегда можем убедить окружающих, что это не наше дело, то оптимизировать процесс с помощью grunt нам точно по силам. Для этой задачи существует специальный плагин – **grunt-contrib-imagemin**. Установим его:

```
$ npm install grunt-contrib-imagemin
```

Пропишем в package.json:

```
"grunt-contrib-imagemin": "^0.7.1"
```

и **Grandfile.js**:

```
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-imagemin');
```

И там же зададим настройки:

```
imagemin: {
    dynamic: {
        files: [{
            expand: true,
            cwd: 'images/',
            src: ['**/*.{png,jpg,gif}'],
            dest: 'images/min/'
        }]
    }
}
```

Запускаем:

```
$ grunt imagemin
Running "imagemin:dynamic" (imagemin) task
✓ images/2012-04-29 10.35.59.jpg (saved 50.16 kB - 3%)
✓ images/2012-04-29 10.36.04.jpg (saved 54.25 kB - 3%)
✓ images/2012-04-29 10.36.06.jpg (saved 54.63 kB - 3%)
✓ images/2012-04-29 10.36.23.jpg (saved 40.2 kB - 3%)
Minified 4 images (saved 199.23 kB)

Done, without errors.
```

Изображения минифицированы и сохранены в папке /min.

Думаю, с настройками все более или менее ясно: cwd – это путь к исходным файлам, src – шаблон имени файлов – картинок, предназначенных для минификации, dest – папка назначения. С файлами можно разобраться еще и в такой форме:

```
  files: {
        'dist/img.png': 'src/img.png',
        'dist/img.jpg': 'src/img.jpg',
        'dist/img.gif': 'src/img.gif'
      }
```

Параметр expand разрешает динамическое расширение.

Опционально можно задать еще несколько параметров (применимых только к определенному типу файлов):

- **optimizationLevel** (для изображений png) – уровень оптимизации изображений формата png (от 0 до 7, по умолчанию 3, что соответствует 16 trials);
- **progressive** (для изображений jpg) –преобразование без потерь (по умолчанию true);
- **interlaced** (для изображений gif) – рендеринг в Interlace-режиме (по умолчанию true).

С графикой разобрались. Какая еще рутина у нас осталась? Ну, хотя бы прекомпилировать stylus css-шаблоны в обычные .css-файлы – задача тоже однообразная и не очень интеллектуальная. Поэтому будем автоматизировать. Для этого у нас есть плагин **grunt-contrib-stylus**:

```
npm install grunt-contrib-stylus --save-dev

Gruntfile.js:
  grunt.loadNpmTasks('grunt-contrib-imagemin');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-stylus');
  ..........................................
  grunt.registerTask('default'['jshint','concat','uglify','stylus']);
```

Теперь конфигурируем задачу:

```
stylus: {
  compile: {
    files: {
      'css/main.css': 'stylus/main.styl',
      'css/users.css': 'stylus/users.styl',
      'css/bandsmain.css': 'stylus/bands.styl'
    }
  }
}
```

Тут все просто, и если вы дочитали главу до этого места, дополнительных комментариев уже не требуется. А вот небольшая модернизация будет нелишней:

```
stylus: {
  compile: {
```

```
    files: {
      'css/all.css': ['stylus/*.styl'];
    }
  }
}
```

Так мы заодно и склеим все таблицы стилей в один объединенный файл.

```
stylus: {
  compile: {
    options: {
      paths: ['path/to/import', 'another/to/import'],
      urlfunc: 'embedurl', // use embedurl('test.png') in our code to trigger
Data URI embedding
      use: [
        require('fluidity') // use stylus plugin at compile time
      ],
      import: [      //  @import 'foo', 'bar/moo', etc. into every .styl file
        'foo',       //  that is compiled. These might be findable based on values
you gave
        'bar/moo'    //  to `paths`, or a plugin you added under `use`
      ]
    },
    files: {
      'path/to/result.css': 'path/to/source.styl', // 1:1 compile
      'path/to/another.css': ['path/to/sources/*.styl', 'path/to/more/*.styl'] //
compile and concat into single file
    }
  }
}
```

Grunt watch – задача-наблюдатель

Да, grunt берет на себя большую часть рутинной работы, и это сильно облегчает жизнь разработчика. Но всегда хочется чего-то большего. Например, запускать модуль после каждого изменения исходных файлов – это тоже рутина, которую можно автоматизировать. Для этой цели у grunt есть соответствующий плагин – **grunt-contrib-watch.**

Grunt watch – это плагин, запускающий другие задачи при каждом изменении исходных файлов. Сейчас мы установим его и немного автоматизируем процесс.

```
$ npm install grunt-contrib-watch --save-dev
```

При инсталляции с флагом –save-dev в файле package.json должна появиться новая строчка:

```
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-watch": "^0.6.1"
  }
}
```

Теперь добавим в грант-файл загрузку плагина:

```
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-watch');
```

Сконфигурируем его работу:

```
    concat: {
        main: {
        }
    }
  uglify: {
          main: {
             files: {
                 ...
             }
           }
         },
  watch: {    scripts: {
       files: ['**/*.js'],
       tasks: ['concat', 'uglify']
   }
```

Запускаем команду:

```
$ grunt watch
Running "watch" task
Waiting...
```

Теперь задачи по соединению и минификации JavaScript-файлов должны выполняться автоматически при любом их изменении. Проверим это, сделав какие-нибудь изменения в файле index.js и сохранив результат работы. Через некоторое время (по умолчанию 500 миллисекунд) в консоли мы сможем наблюдать действия watch:

```
>> File "index.js" changed.
Running "concat:main" (concat) task
File scripts.js created.

Running "uglify:main" (uglify) task
File scripts.min.js created: 107.69 kB → 107.84 kB

Done, without errors.
Completed in 13.490s at Tue Jun 03 2014 00:21:31 GMT+0400 (Russian Standard Time
) - Waiting...
>> File "scripts.js" changed.
>> File "scripts.min.js" changed.
```

Теперь разработчик может точно почувствовать себя свободным от рутинной работы творцом.

У модуля watch есть опции, позволяющие тонко настроить его работу. Например, вот таким образом можно настроить выполнение задачи **generateFileManifest** при событии добавления или удаления файлов (опция event):

```
watch: {
  scripts: {
    files: '**/*.js',
    tasks: ['generateFileManifest'],
    options: {
      event: ['added', 'deleted'],
    },
  },
},
```

Событием может быть 'changed', 'added','delete' и 'all'.

Grunt connect web server

Чтобы окончательно добить разработчика своей неустанной заботой, grunt, опять же в виде contib-плагина, предоставляет статический веб-сервер, очень удобный для тестирования результата работы. Его установка ничем не примечательна:

```
$ npm install grunt-contrib-connect --save-dev
```

Пропишем загрузку модуля в Grandfile.js:

```
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-connect');
```

Теперь там же сконфигурируем задачу:

```
connect: {
    testHost: {
        options: {
            port: 8000,
            hostname: '127.0.0.1',
            base: 'C:\Users\Geol\node\grunt',
        }
    }
}
```

Параметры, которые мы задали, очевидны – порт, хост и путь к корню нашего веб-сервера. Теперь запустим модуль:

```
$ grunt connect
Running "connect:test" (connect) task
Started connect web server on http://127.0.0.1:8000

Done, without errors.
```

Хм... Нет, ну, то, что наш сервер отработал без ошибок, – это прекрасно, но как-то это все слишком быстро. Дело в том, что по умолчанию работа grant connect длится все время работы Grandfile.js и не больше – это удобно для различных сложных сценариев загрузки и тестирования различных ресурсов. Чтобы модуль работал как полноценный веб-сервер (то есть слушал указанный http-порт постоянно), необходимо добавить еще одну опцию:

```
      options: {
           port: 8000,
           hostname: '127.0.0.1',
           base: 'C:\Users\Geol\node\grunt',
    keepalive: true,
      }
   }
}
```

Теперь совсем другое дело:

```
C:\Users\Geol\node\grunt>grunt connect
Running "connect:test" (connect) task
Waiting forever...
Started connect web server on http://127.0.0.1:8000
```

Сервер работает, в чем нетрудно убедиться, зайдя браузером на url http://127.0.0.1:8000/.

На этом все. Я надеюсь, что этой главой я поспособствовал улучшению качества жизни некоторого количества веб-разработчиков.

Альтернативы JavaScript и Node.js

Как уже неоднократно говорилось, чистый JavaScript вызывает претензии у многих разработчиков. Особенно много нареканий исходит от программистов, привыкших к языкам с классической объектной моделью, но, кроме непривычного ООП, есть и другие замечания, часть из которых, как это не грустно, вполне обоснована.

CoffeeScript – зависимость с первой чашки

Underneath that awkward Java-esque patina, JavaScript has always had a gorgeous heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way.
http://coffeescript.org/

Этот язык стал, наверное, первой удачной попыткой заменить классический JavaScript. CoffeeScript – это четкий, внятный синтаксис в духе Python или Ruby, это функциональные возможности прямо из Haskell, Lisp и Erlang, это килограммы «синтаксического сахара» – считается, что в среднем для выполнения одинаковых действий на CoffeeScript требуется в 2 раза меньше строк, чем на чистом JavaScript.

Язык CoffeeScript возник в недрах сообщества Ruby On Rails. Он построен поверх JavaScript и был изначально спроектирован для компиляции в JavaScript-код, исполняемый в веб-браузере.

Компилятор, написанный на Ruby, вскоре был реализован на самом CoffeeScript, и теперь процесс компиляции очень прост.

В настоящее время CoffeeScript можно использовать как в веб-браузере, так и на платформе Node.js для создания полноценных серверных приложений.

Что представляет собой код на CoffeScript? Что этот диалект дает разработчику? Вкратце – CoffeScript:

- предоставляет простой синтаксис с меньшим количеством «знаков препинания», в том числе для выражения функций и объектов;
- использует пробелы как способ организации блоков кода;

- обеспечивает наследование на основе классов (одна из основных претензий к JavaScript – «прототипичность» языка);
- JavaScript-код, компилируемый из CoffeeScript, полностью соответствует стандарту и проходит проверку JavaScript Lint (утилита для статического анализа JavaScript-кода);
- при всем этом получавшийся в результате компиляции JavaScript-код весьма эффективен и не дает сколько-нибудь значимых проблем со снижением производительности;
- новый стандарт, ECMAScript.next, разработка которого сейчас ведется, развивается под заметным влиянием Coffee. После его принятия и повсеместного распространения поддерживающих стандарт виртуальных машин всем, наверное, будет хорошо, но пока для тех, кто не хочет/не может ждать, есть CoffeScript.

Краткий курс по кофе

Проще всего CoffeScript можно освоить, рассмотрев, как он реализует обычные JavaScript-конструкции. Приступим.

Для того чтобы изложение не было излишне унылым, рекомендую зайти на официальный сайт coffeeScript (http://coffeescript.org/) и в разделе «Try CoffeeScript» (рис. 39) попробовать скомпилировать нижеприведенные примеры в JavaScrip, пользуясь предоставленной

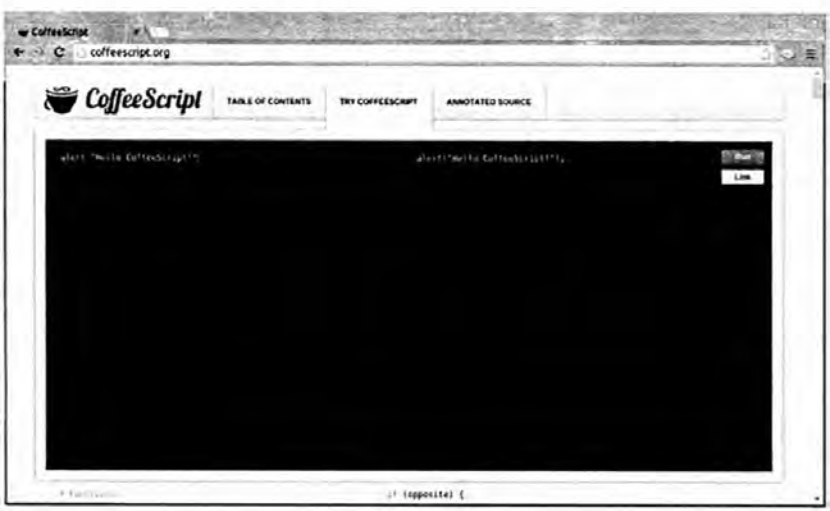

Рис. 39 ❖ Чашечку кофе?

консолью (это не online-компиляция, все будет происходить у вас в браузере без сетевых задержек).

Прежде всего для выделения блоков кода CoffeeScript использует значимые отступы. Их следует применять вместо фигурных кавычек ({}) в синтаксисе функций, объектов, циклов, условий if, switch и try/catch. Использовать точку с запятой не обязательно (хотя и не возбраняется), для разделения выражений подойдет и конец строки. Теперь сравним некоторые конструкции:

Присваивание:

JavaScript

```
var name = «John»;
var age = 32;
var  admin = true;
list = [1, 2, 3, 4, 5];
```

CoffeScript

```
name = «John»
age = 32
admin = true
list = [1, 2, 3, 4, 5]
```

Заметили разницу? (Ну, кроме отсутствия ключевого слова var, в котором CoffeScript не нуждается.) Нет? А она есть, смотрите внимательнее.

Да, а, собственно, почему в var нет необходимости? Все просто. Компилятор сам CoffeeScript заботится о том, чтобы все ваши переменные были правильно декларированы в лексической области. Вот во что превратится CoffeeScript-код справа после компиляции:

```
var admin, age, list, name;
name = "John";
age = 32;
admin = true;
list = [1, 2, 3, 4, 5];
```

Функции:

JavaScript

```
square = function(num) {
    return  num*num;
};
s = square(4);
```

CoffeScript

```
square = (num) -> num *  num
square 4
```

Тут все понятно без комментариев (ну, по крайней мере, можно догадаться). В скобках (опционально) передается список параметров, тело функции расположено за стрелкой. Пустая функция выглядит так:

```
->
```

Обратите внимание: нет необходимости добавлять слово return. Возвращаемым значением будет последнее выражение в функции.

Функциям можно задавать значения по умолчанию (у «родного» JavaScript, заметим, с этим сложности):

JavaScript

```
myfun = function(x, y) {
  if (y == null) {
    y = 2;
  }
  return x*y;
};
```

CoffeScript

```
myfun = (x, y=2) -> x*y
```

Условия:

JavaScript

```
if (age > 16) {
  accept = true;
}
```

CoffeScript

```
accept = true if  age > 16
```

Небольшой, но приятный сюрприз – в CoffeeScript есть возможность проверить переменную на значения null и undefined за один прием. Для этого предназначен оператор «?»:

JavaScript

```
if (typeof myVar !== «undefined»
   && myVar !== null) {
  alert("Yes!");
}
```

CoffeScript

```
alert «Yes!» if myVar?
```

Объекты:

JavaScript

```
user = {
  name: «John»,
  age: 38,
  daysOfLive: function(x) {
    return x * 365;
  }
};
```

CoffeScript

```
user =
  name:   «John»
  age: 38
  daysOfLive:   (x) -> x * 365
```

Тут все очень похоже на объекты в JavaScript. Можно опускать запятые, когда каждое свойство с его значением находится на своей строке.

Классы, наследование, полиморфизм, генераторы!

И это, конечно, не все. В CoffeeScript присутствует много интересных вещей, которых в JavaScript просто нет. Например, классы. Тут все очень серьезно (далее пример из официального руководства).

Создаем базовый класс:

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."
```

Запись @name – сокращение от this.name, оно определяет свойство класса name и автоматически присваивает ему значение, передаваемое в конструкторе.

Наследуем:

```
class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5
```

И ещё:

```
class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45
```

В методе **move** дочерних классов **super** вызывает родительский метод с тем же названием.

Создаем экземпляры классов:

```
sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"
```

И вот он – полиморфизм!

```
sam.move()
tom.move()
```

В какой javaScript-код превращаются эти конструкции после компиляции, вы можете посмотреть сами, я тут не буду это приводить ввиду экономии бумаги. Ну разве что самое начало:

```
var Animal;

Animal = (function() {
  function Animal(name) {
    this.name = name;
```

```
  }

  Animal.prototype.move = function(meters) {
    return alert(this.name + (" moved " + meters + "m."));
  };

  return Animal;

})();
```

Словом, тем, кто твердо не принимает прототипную модель ООП, лучше этого не видеть.

Еще одной, может, менее значительной, но не менее полезной, является возможность работы с диапазонами (range). Если вы не работали с функцией range в Python или с аналогичными конструкциями в Ruby, все станет ясно из простого примера. Допустим, мы хотим получить массив квадратов чисел от 1 до 10. В CoffeeScript достаточно написать:

```
more = (num * num for num in [10..1])
```

В чистом JavaScript это много более объемный код:

```
var more, num;

more = (function() {
  var _i, _results;
  _results = [];
  for (num = _i = 1; _i <= 10; num = ++_i) {
    _results.push(num * num);
  }
  return _results;
})();
```

Такая конструкция CoffeScript называется генератором. Генераторы упростят жизнь там, где вы раньше использовали циклы each/forEach или map.

Надеюсь, вышеизложенными примерами я если не убедил использовать CoffeScript, то, по крайней мере, привлек внимание к этому языку.

CoffeScript и Node.js

Изначально CoffeScript разрабатывался для применения в браузерах, но node.js-программисты довольно быстро взяли его на вооружение. Расширение для платформы доступно в виде модуля и устанавливается обычным способом:

```
npm install coffee-scrip
```

После установки мы получаем доступ к команде coffee, являющейся интерфейсом компилятора CoffeeScript (в данном случае работающего на движке V8). coffee предоставляет интерактивный REPL-интерфейс для работы в командной строке. С помощью этого инструмента можно выполнять скрипты и компилировать файлы .coffee в файлы js. Для проверки попробуем запустить компилятор с ключом, требующим вывода версии продукта:

```
$ coffee -vCoffeeScript version 1.6.3
```

Как видите, все в порядке.

Команда coffee принимает следующие основные опции (краткий и полный синтаксис):

- **-c, --compile** – компилирует скрипт .coffee в .js-файл JavaScript с тем же именем.

  Сразу же попробуем это в деле. Создадим файл **start.coffee** со следующим содержанием:

  ```
  square = (num) -> num *  num
  alert '2*2='+ square 4
  ```

  Теперь запустим компилятор:

  ```
  $ coffee -c start.coffee
  ```

  Сейчас в той же папке должен появиться файл **start.js**, содержащий следующий Javascript-код:

  ```
  // Generated by CoffeeScript 1.6.3
  (function() {
    var square;

    square = function(num) {
      return num * num;
    };

    alert('2*2=' + square(4));

  }).call(this);
  ```

  Хм... я бы на JavaScript написал это чуть по-другому, но, по крайней мере, все работает. Идем дальше.

- **-p, --print** – компилирует скрипт .coffee в JavaScript и выводит напрямую в консоль (то есть в STDOUT, что бывает очень полезно при отладке).
- **-s, --s** – компилятор, запущенный с этим ключом, принимает CoffeeScript в STDIN и возвращает JavaScript в STDOUT. Например, так:

```
cat src/myscript.coffee | coffee -sc > myscript.js
```

- **-o, --output [DIR]** – записывает весь компилированный вывод JavaScript-файлов в указанную директорию. (Используется совместно с –compile.)
- **-e, --eval** – еще один режим, полезный для отладки. С ним coffee компилирует и выводит небольшой кусок CoffeeScript прямо в командную строку:

```
$ coffee -e "console.log num for num in [10..1]"
10
9
8
7
6
5
4
3
2
1
```

И еще один интересный ключ:

- **-w, --watch** – если coffee запущен с этим ключом, он следит за изменениями файлов, автоматически перезапуская указанную команду при их обновлении. В качестве примера тут coffee следит за изменениями в файле и перекомпилирует его каждый раз, когда файл сохраняется:

```
coffee --watch --compile myscript.coffee
```

Пишем сервер на CoffeeScript

Компиляция исходных coffee-файлов в JavaScriptpt-сценарии – дело нужное. С помощью различных опций компилятора мы можем автоматизировать этот процесс до такой степени, что программисту, разрабатывающему клиентское приложение и пишущему свой код на CoffeeScript, вообще больше не нужно будет ни о чем беспокоиться. Но тут же возникает вопрос: а можем ли мы прямо на CoffeeScript написать рабочее северное приложение? Да конечно можем, ведь coffee – это интерпретатор, работающий на Node.js.

Попробуем исполнить в среде Node вот такой простенький coffee-Script-сценарий (файл **simple.coffee**):

```
for i in [0..5]
    console.log i
```

Запустим его на выполнение, естественно, командой coffe (фактически Coffee скомпилирует файл в JavaScript и сразу передаст его Node):

```
$ coffee simple.coffee
0
1
2
3
4
5
```

Вот как будет выглядеть простой HTTP-сервер, написанный на CoffeeScript:

```
http = require "http"
http.createServer (req, res)->
  setTimeout (()->
    res.writeHead 200, {'Content-type':'text-plain'}
    res.end message), 200
.listen 8000
console.log "Server running at http://127.0.0.1:8000/"
```

Запустим:

```
$ coffee server.coffee
Server running at http://127.0.0.1:8000/
```

Оно работает. На coffee. На Node.js. На V8. На... не важно.

При компиляции в JavaScript мы бы получили следующее:

```
var http;
http = require("http");
http.createServer(function(req, res) {
  return setTimeout((function() {
    res.writeHead(200, {
      'Content-type': 'text-plain'
    });
    return res.end(message);
  }), 200);
}).listen(8000);

console.log("Server running at http://127.0.0.1:8000/");
```

Если нам захочется подключить модуль, написанный на CoffeeScript (что довольно естественно), отличий от обычного JavaScript-модуля немного. Простой пример. Сам модуль (**simple.coffee**):

```
cube =
  hello: "Hello coffee module!"
  volume: (x)-> x * x * x
exports.cube = cube
```

Ничего особенного – создаем обычный объект и экспортируем его. Теперь основной файл Node.js-приложения (**with_coffee.js**):

```
var coffee = require('coffee-script');
var cube = require('./simple').cube;
```

```
console.log(cube.hello);
console.log(cube.volume(7));
```

Больше ничего не надо – просто запускаем:

```
$ node with_coffee
Hello coffee module
343
```

Надеюсь, мне удалось показать, что пользоваться CoffeeScript на платформе Node.js просто и эффективно. Но это не единственное решение для тех, кому не нравится работать с традиционным JavaScript.

TypeScript – типа Javascript от Microsoft

TypeScript – язык, позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript. Он был представлен корпорацией Microsoft в 2012 г.

Примечательна личность разработчика – это Андерс Хейлсберг (*Anders Hejlsberg*), принимавший ведущее участие в создании Turbo Pascal, Delphi и C#.

В отличие от CoffeeSript, TypeScript является обратносовместимым с JavaScript. Он компилируется в JavaScript, после чего программу на TypeScript можно выполнять в любом современном браузере. Как и CoffeeSript, его вполне можно использовать совместно с платформой Node.js.

Отличия TypeScript от JavaScript – это:

- возможность явного определения типов (статическая типизация);
- поддержка использования полноценных классов (как и традиционная объектно-ориентированная модель);
- поддержка подключения модулей.

Подобные нововведения призваны повысить скорость разработки, читаемость, рефакторинг и повторное использование кода, дать возможность осуществлять поиск ошибок на этапе разработки и компиляции, а также увеличить скорость выполнения программ.

Хотя, если честно, последний абзац смотрится и вовсе как рекламный слоган. Давайте посмотрим, что там есть на самом деле.

Хорошие новости – действительно каждая программа JavaScript является корректной программой TypeScript. Более того, компилятор TypeScript выполняет только локальное преобразование файлов и не делает никаких переопределений переменных, и не меняет их названия. Это позволяет писать код, максимально близкий к оригинальному JavaScript.

Node.js как TypeScript-компилятор

Ну, теперь за дело. TypeScript действительно нуждается в компиляции, и путей для нее ровно два – использовать Microsoft Visual Studio (от 2012) или... да! соответствующий модуль Node.js:

```
npm install -g typescript
```

Пользоваться им просто. Создаем простой файл с TypeScript-кодом (hello.ts):

```
function greeter(person: string) {
    return "Hello, " + person;
}
console.log(greeter("TypeScript"));
```

Теперь компиляция. Набираем в консоли:

```
$ tsc hello.ts
```

Результатом будет файл **hello.js**:

```
function greeter(person) {
    return "Hello, " + person;
}
console.log(greeter("TypeScript"));
```

Немного различий, правда? Ну теперь, когда у нас все работает, давайте разберемся с самим языком.

Аннотации типов

Что нам может предложить TypeScript? Прежде всего статическую типизацию, мы ее наблюдали в первом же примере:

```
function greeter(person: string){...}
```

Тип данных тут указан после имени аргумента. Если мы передадим функции неверный тип данных, при компиляции будет выдано предупреждение:

```
greeter(1);
```

Результат:

```
$ tsc hello.ts
gr.ts(4,13): error TS2082: Supplied parameters do not
match any signature of call target:
        Could not apply type 'string' to argument 1 which is of type 'number'.
gr.ts(4,13): error TS2087: Could not select overload f
or 'call' expression.
```

Можно указать тип возвращаемого значения:

```
function process(x: number, y: number): number {
    var sum = x + y;
    return sum;
}
```

А можно вообще ничего не указывать – строгая типизация не является в TypeScript обязательной. Зато функции теперь можно задавать параметры по умолчанию и указывать необязательные аргументы:

```
function process(x =5, y?: number): number {
    var sum;
    if(!y){
        sum = x;
  } else{
      sum = x + y;
  }
    return sum;
}
console.log(process(2,6)); //6
console.log(process(2)); //2
console.log(process()); //5
```

Результаты компиляции:

```
function process(x, y) {
    if (typeof x === "undefined") { x = 5; }
    var sum;
    if (!y) {
        sum = x;
    } else {
        sum = x + y;
    }
    return sum;
}
console.log(process(2, 6)); //8
console.log(process(2)); //2
console.log(process()); //5
```

Больше всего мне тут нравится, что результирующий код на JavaScript получается простым и понятным. Правда, это только на простых конструкциях, а нам пора переходить к более сложным.

Классы! настоящие классы!

Готов поспорить, ради того все и затевалось. Да, в TypeScript существуют «нормальные» классы и «нормальное» наследование. Пример TypeScript-класса:

```
class Person {
    name: string;
    surname: string;
    private id: number;
    static title = "Example";
    constructor (name: string, surname: string) {
       this.name = name;
       this.surname = surname;
    }
```

```
    setID (id) { this.id = id; }
    getFullName () { return this.name+" "+this.surname; }
}

console.log(Person.title+":"); // Example:

var User = new Person("Kirill", "Sukhov");
console.log(User.name); //  Kirill
console.log(User.getFullName()); //  Kirill Sukhov
```

Тут почти все, о чем мечтали сторонники «традиционного» ООП, – поля, методы, конструктор. Имеются и модификаторы доступа – попытка получить значение User.id или установить его значение непосредственно, а не с помощью специальных методов класса Person, потерпит неудачу (модификатор public тоже есть, но его почти всегда можно опустить).

Статические поля и свойства также подчиняются привычным законам – то есть доступны без создания экземпляра класса.

При компиляции этого кода мы получим следующую, не очень сложную JavaScript-конструкцию:

```
var Person = (function () {
    function Person(name, surname) {
        this.name = name;
        this.surname = surname;
    }
    Person.prototype.setID = function (id) {
        this.id = id;
    };
    Person.prototype.getFullName = function () {
        return this.name + " " + this.surname;
    };
    Person.title = "Example";
    return Person;
})();

console.log(Person.title + ":"); // Example:

var User = new Person("Kirill", "Sukhov");
console.log(User.name);
console.log(User.getFullName());
```

Теперь наследование. Напишем еще один класс, расширяющий предыдущий:

```
class Member extends Person {
  band: string;
  constructor(name: string, surname: string, band: string){
      super(name, surname);
    this.band = band;
```

```
  }
  getBand () { return this.band; }
}
var User = new Member("John", "Lennon", "The Beatles");
console.log(User.getFullName()); //  John Lennon
console.log(User.getBand()); // The Beatles
```

Мы добавили немного – название группы. В конструкторе мы методом **super()** вызываем родительский конструктор.

Можно переопределить родительский метод:

```
class Member extends Person {
  band: string;
  constructor(name: string, surname: string, band: string){
      super(name, surname);
    this.band = band;
  }
  getBand () { return this.band; }
  getFullName() {
    return  super.getFullName()+" From "+this.band;

  }
}
var User = new Member("John", "Lennon", "The Beatles");
console.log(User.getFullName()); //  John Lennon from  The Beatles
```

Интерфейсы

Да, кроме классов, в TypeScript существуют и эти полезные языковые конструкции. Ниже – пример простого интерфейса и использующей его функции:

```
interface Band {
  name: string;
  state?: string;
  members: any;
}

function ShowBand(band: Band) {
  console.log(band.name);
  if(band.state){
      console.log(band.state);
  }
  band.members.forEach( function(member){
          console.log(member);
     });
  }

var obj = { name: "Focus",
            state: "nl",
            members: ["Thijs", "Jan", "Martin", "Hans"]
     }
ShowBand(obj);
```

Тут интерфейс контролирует параметры объекта, передаваемого функции, как аргумент. Знак вопроса после параметра id указывает на его необязательность. При компиляции кода в JavaScript интерфейс исчезает, он свое дело сделал:

```
function ShowBand(band) {
    console.log(band.name);
    if (band.state) {
        console.log(band.state);
    }
    band.members.forEach(function (member) {
        console.log(member);
    });
}

var obj = {
    name: "Focus",
    state: "nl",
    members: ["Thijs", "Jan", "Martin", "Hans"]
};
ShowBand(obj);
```

Модули

Модули в TypeScript организованы по стандартам CommonJS и EcmaScript6. Их задача – инкапсуляция бизнес-логики в отдельной конструкции со своим пространством имен... хотя что я объясняю, что такое модули в конце книги про Node.js? Лучше покажу на практике, как они реализованы в TypeScript.

```
module Say {
  export function Hello(text: string) {
  return "Hello " + text;
  }
}
console.log(Say.Hello("Module"));
```

Ничего не напоминает? Хотя чего, собственно, напоминать, это не сходство, это тот же самый механизм, который мы используем в Node.js. При компиляции получаем следующее:

```
var Say;
(function (Say) {
    function Hello(text) {
        return "Hello " + text;
    }
    Say.Hello = Hello;
})(Say || (Say = {}));
console.log(Say.Hello("Module"));
```

Что еще?

Кроме разобранных нами языковых конструкций, TypeScript еще много чем может порадовать разработчика. Например, реализацией примесей (mixins), arrow-синтаксисом из EcmaScript 6, типом данных Generic. Очень динамично развивающаяся технология, поддерживаемая технологическим гигантом, – даже если вы сторонник чистого JavaScript, не стоит обделять вниманием этот проект.

Dart – дротик в спину JavaScript от Google

Естественно, «корпорация добра», разработчик движка V8 и одного из самых популярных браузеров Google Chrome, не могла пройти мимо тенденции «улучшения» JavaScript, и 12 сентября 2011 г. на конференции разработчиков Goto была проведена официальная презентация языка Google Dart.

В отличие от аналогов, Dart прямо позиционируется в качестве замены JavaScript, «страдающего от фундаментальных изъянов, которые невозможно исправить путём эволюционного развития».

Разработчики языка поставили перед собой следующие задачи:

- создать структурированный и в то же время гибкий язык для веб-программирования;
- сделать язык похожим на существующие для упрощения обучения;
- обеспечить высокую производительность получаемых программ как в браузерах, так и в иных окружениях, начиная от смартфонов и заканчивая серверами.

В настоящее время Dart-программы исполняются двумя способами: с использованием виртуальной машины (в браузере Google Chrome) и с промежуточной трансляцией (компиляцией) в JavaScript.

Самое интересное, что Dart претендует на то, чтобы стать прямым конкурентом Node.js – виртуальная машина Dart (VM Dart) является неразрывной частью языка. С помощью нее можно запускать Dart-программы в консольном режиме, и средства для работы на стороне сервера в Dart имеются в достаточном количестве.

Экосистема Dart

Давайте сначала рассмотрим, что собой представляет этот язык. Для этого имеет смысл познакомиться со специализированными инструментами, созданными Google для сопровождения технологии. На странице проекта (рис. 40) мы сразу можем видеть призыв – Downlod

Рис. 40 ❖ Dart – официальный портал проекта

Dart+Editor. Воспользуемся этим предложением и получим, скачав и распаковав архив, много интересных вещей.

Прежде всего это витуальная машина Dart (Dart VM) – серверная платформа, реализующая событийно-ориентированную, асинхронную модель исполнения, сходную с так хорошо уже знакомым нам принципом работы Node.js.

Далее Dart Editor – полноценная интегрированная среда разработки (IDE), основанная на Eclipse.

Dart Editor дополнен веб-браузером – специальной модификацией обозревателя Chromium со встроенным Dart (Dart VM) движком.

Наконец, нам теперь доступен набор Dart SDK, куда входят сам интерпретатор dart, утилита для трансляции dart-приложения в JavaScript-файл (dart2js), утилита для генерации документации (docgen) и прочие полезные инструменты.

Давайте запустим редактор, создадим новый проект (иконка в левом верхнем углу основного окна IDE, следует выбрать тип comand-line application) и напишем нашу первую dart-программу. Ничего оригинального:

```
main() {
    print("Hello Dart");
}
```

Теперь запустим ее на выполнение, кликнув по стрелке на верхней панели инструментов. Судя по появившейся строчке «Hello Dart» в консоли редактора (рис. 41), Dart работает (об особенностях самого языка – чуть далее). Того же результата мы можем достичь, сохранив вышеприведённый код в файле **hello.dart** и запустив его с помощью интерпретатора из командной строки:

```
$ dart hello.dart
Hello Dart
```

Рис. 41 ❖ Работа в Dart Editor

Теперь можно попробовать транслировать нашу программу в JavaScript:

```
$ dart2js hello/bin/hello.dart
Dart file (hello\bin\hello.dart) compiled to JavaScript: out.js
```

Сделано! Скрипт получился рабочим, но слабочитаемым, поэтому сгенерированные 14 Kb я тут приводить не буду.

Теперь стоит вспомнить, что Dart задуман в первую очередь как язык, исполняемый в браузере. Если мы создадим новый проект, при этом выбрав тип Wep Application, в качестве заготовки для приложения мы получим следующий dart-код:

```
import 'dart:html';

void main() {
  querySelector("#sample_text_id")
```

```
      ..text = "Click me!"
      ..onClick.listen(reverseText);
}

void reverseText(MouseEvent event) {
  var text = querySelector("#sample_text_id").text;
  var buffer = new StringBuffer();
  for (int i = text.length - 1; i >= 0; i--) {
    buffer.write(text[i]);
  }
  querySelector("#sample_text_id").text = buffer.toString();
}
```

Не будем его трогать, а обратим внимание, что IDE, кроме файла сценария, создала еще и html-шаблон. И его стоит рассмотреть:

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Helloweb</title>

    <script async type="application/dart" src="helloweb.dart"></script>
    <script async src="packages/browser/dart.js"></script>

    <link rel="stylesheet" href="helloweb.css">
  </head>
  <body>
    <h1>Helloweb</h1>

    <p>Hello world from Dart!</p>

    <div id="sample_container_id">
      <p id="sample_text_id">Click me!</p>
    </div>
  </body>
</html>
```

Тут сразу бросается в глаза наличие нового Internet Media Type – application/dart. Сам dart-сценарий подгружается и выполняется посредством тега <script> специфицированным данным типом. После запуска приложения в IDE откроется браузер, и мы сможем увидеть нашу программу в работе (рис. 42).

Теперь обратим внимание на строчку

```
<script async src="packages/browser/dart.js"></script>
```

Это обычный JavaScript-сценарий, который проверяет наличие в DOM браузера поддержки функции **navigator.webkitStartDart()** и

Рис. 42 ❖ Dart-веб-приложение

в случае отсутствия таковой заменяет dart-сценарий Javascript-файлом, сгенерированным dark2js. Так что получившаяся страница отработает и в Firefox (правда, не очень быстро).

Как работает Dart, мы разобрались, теперь давайте посмотрим, что собой представляет сам язык.

Знакомство с Dart

Еще раз подчеркнем, Dart – это не диалект JavaScript и не настройка над ним, это совершенно самостоятельный язык, имеющий с расширением EcmaScript 262 только одно пересечение – сферу применения. Создавали его люди, похоже, знающие и любящие технологии C++ , Java и C# и не пожелавшие отказываться от своих привычек. И это здорово, но немного непривычно для традиционного клиент-сайт-веб-программирования. Вот полюбуйтесь на Helo Word по-dart`ски:

```
main() {
    var d = "Dart";
    String w = "World";
    print("Hello ${d} ${w}");
}
```

Прежде всего бросается в глаза использование функции **main()**. Она в Dart делает именно то, что и в C/C++, – служит точкой входа в исполняемую программу, с которой начинается ее работа. Когда сценарий исполняется на странице, она начинает исполняться немедленно после загрузки DOM-модели документа.

JavaScript – язык с динамической типизацией, что здорово. В C++ или Java типизация строгая, статическая, что вообще замечательно. Но лучше всего с этим дела обстоят у Dart – типизация у него факультативная. Это обозначает то, что при объявлении переменных можно обозначать типы, а можно этого не делать, все зависит только от ваших задач:

```
var foo = "Hi"; // объявление без аннотации типа
String foo = "Hi"; // объявление с аннотацией типа String
final String foo = "Hi"; // объявление с аннотацией типа String и
                         // финализацией.
```

Ключевое слово final используется в том случае, когда значение, с которым она инициализирована, не должно далее быть изменено.

Аннотации типов могут встречаться также в объявлениях параметров функций и возвращаемых значений, а еще в определениях классов.

ООП – чтим традиции!

Создатели Dart не стали экспериментировать с ОПП. В Dart классы и объекты используются вполне традиционно, в рамках классической модели наследования:

```
class Rectangle {
    var _id;
    var name;
    final num height,
              width;
    //конструктор (Короткий синтаксис)
    Rectangle(num this.height, num this.width);
    // метод
    num area() {
      return height * width;
      }
    // метод (Короткий синтаксис)
    num perimeter() => 2*height + 2*width;
    // акцессоры
    get id => _id;
    set id(val) => _id = val;
}
class Square extends Rectangle {
  Square(num size) : super(size, size);
}
```

Пример простого класса и одного наследников. Тут есть на что обратить внимание.

Dark поддерживает короткий синтаксис обновления функций и методов.

Закрытые члены обозначаются добавлением знака подчеркивания _ в начало имени (в Dart нет ключевых слов public, protected и private. Если имя начинается с подчеркивания (_), то это приватное свойство). Ключевое слово **this** применяется строго традиционным образом, ссылаясь на конкретный экземпляр класса.

Наследование поддерживается одному классу и нескольким интерфейсам.

Использование классов тоже вполне традиционно:

```
var myRect = new Rectangle(3,4);  //создает новый экземпляр класса
                                  //Rectangle
myRect.name = "Nice Rectangle";   // присваиваем значение полю
print(myRect.perimeter());
```

Интересной особенностью классов в dart является то, что каждый из них неявно подразумевает интерфейс, который можно имплементировать. То есть переписать его методы с собственной реализацией:

```
class Shape {
  num length;
  num perimeter() => length * 4;
}

class Square implements Shape {
  num length;
  num perimeter() => length * 4;
}
class Triangle implements Shape {
  num length;
  num perimeter() => length * 3;
}
class Rectangle implements Shape {
  num length;
  num width;

  num perimeter() => length * 2+width*2;
}

num printPerimetr(Shape shape) {
    print(shape.perimeter());
}

main() {
    var square = new Square();
    square.length = 4;
    printPerimetr(square); // 16

    var triangle = new Triangle();
    triangle.length = 4;
    printPerimetr(triangle); // 12
    var rectangle = new Rectangle();
```

```
    rectangle.length = 4;
    rectangle.width = 6;
    printPerimetr(rectangle); // 22
}
```

Тут мы имплементируем интерфейс класса тремя различными способами, но функция **printPerimetr()**, ожидающая типизированного параметра (Shape), отрабатывает успешно.

Есть в Dart и абстрактные классы, с помощью которых можно реализовывать фабричный конструктор объектов:

```
abstract class iHello {
     String sayHello(String realise, String name);
     factory iHello(realise) {
         return new realise();
     }
}
class RussianHello implements iHello {
    sayHello(name) {
         return "Привет $name";
    }
}

void main() {
  iHello myHello = new iHello(); //
    var message = myHello.sayHello("Dart");
    print(message); //  Привет  Dart
}
```

Область видимости и библиотеки

Функции и классы можно организовать в библиотеки – совокупность исходных файлов, сгруппированных, исходя из логических соображений. Каждая библиотека может содержать несколько классов и функций верхнего уровня. Библиотеки могут импортировать другие, необходимые для работы файлы и библиотеки. Пример простой библиотеки приведен ниже:

```
library animals;

part "paint.dart";
class Dog {
  noise() => paint('BARK!');
}
```

И ее использование:

```
import 'animals.dart';
var fido = new Dog();

import 'animals.dart' as pets;
var fido = new pets.Dog();
```

Ключевое слово library служит для определения библиотеки.

Импортируется библиотека с помощью ключевого слова import, а part используется для ссылки на другие исходные файлы. Именно на уровне библиотеки имеют значения закрытые члены класса. Клиенты библиотеки не могут обращаться непосредственно к ним, вместо этого используются аксессоры.

Наверное, не будет сюрпризом тот факт, что библиотек для Dart уже написано великое множество, и основные из них поставляются вместе с языком, как часть Dart SDK. Их мы можем наблюдать (и, естественно, использовать) в IDE Dart Editor (рис. 43). Одними из значимых являются dart:html – библиотека для работы с DOM-моделью браузера и dart:io, содержащая классы и функции для доступа к файловой системе, сетевым сокетам, для работы с протоколом HTTP и веб-сокетами.

Рис. 43 ❖ Библиотеки Dart

Изоляторы

Любое Dart-приложение работает в однопоточном режиме. Но современные браузеры, даже на мобильных платформах, работают на многоядерных процессорах. Для использования всех ядер разработчики применяют работающие одновременно потоки с разделяемой памятью.

В Dart единицей работы является изолятор (**isolate**), а не процесс или поток. У каждого изолятора имеется собственная область памяти (этим изолятор в основном и отличается от потока), недоступная любому другому изолятору. Изоляторы могут работать параллельно, могут обмениваться сообщениями (сигналами и данными).

Механизм изоляторов доступен и на веб-странице – каждый скрипт, содержащий функцию **main()**, исполняется в отдельном изоляторе. При трансляции в JavaScript такие конструкции будут реализованы посредством технологий HTML5 (WebWorkers).

Программа на Dart может создать новый изолятор (по аналогии с **fork()**). Для создания нового изолятора с указанной точкой входа необходимо импортировать библиотеку **dart:isolate** и вызвать функцию **spawnFunction()**, передав ей имя точки входа:

```
import "dart:isolate";
void main() {
 ...
analyzeFileList(runSript);
}
```

Изоляторы хорошо использовать как для распределения логики между процессорами, так и для динамической загрузки внешнего кода. Код, не являющийся частью приложения, можно загружать и исполнять в изолированной области.

Я даже вкратце не перечислил всех особенностей новой технологии Google, но я думаю, вы можете оценить масштаб инноваций. Dart явно претендует на место универсального и всеобъемлющего языка веб-программирования. Получится ли у него его занять? Вопрос сложный. Но старт технологии очень впечатляет.

А в общем-то...

Честно потратив время на исследование различных разработок около JavaScript и изложив результаты на этих страницах, не могу удержаться и не заметить, что сама идея заменить JavaScript не кажется мне достаточно обоснованной. Достаточно простой и вместе с тем мощный язык доказал свое право на существование, а некоторые претензии к нему действительно кажутся надуманными.

Будущее уже сейчас – ECMAScript.next и Node

Последняя реализация стандарта ECMA-262 – ECMAScript 5 – утверждена в 2009 г. Надо сказать, что изменения, внесенные в спецификацию (и воплощенные затем во многих современных реализациях JavaScript), были не революционны, но очень востребованы. Поскольку утверждение стандарта практически совпало с рождением платформы Node.js, в последней все новые возможности были доступны «из коробки» (чем мы, не стесняясь, пользуемся на протяжении всей книги). А вот браузеры были менее расторопны, хотя сейчас уже можно говорить о всеобъемлющей поддержке стандарта. Давайте кратко пройдемся по новшествам ECMAScript 5, чтобы оценить, что привнесено в JavaScript спустя 10 лет, после утверждения предыдущей версии (ECMAScript 3 – утвержден в 1999 г.). Заодно если вы по каким-то причинам не знаете новых возможностей языка – самое время их освоить (лучше поздно, чем никогда).

ECMAScript 5 – уже стандарт

На самом деле причина, побудившая меня писать о ECMAScript 5, – это поведение некоторых моих коллег, веб-разработчиков, в упор не желающих замечать тех перемен, которые происходят в языке JavaScript в последние годы. Если вы держитесь в курсе – прошу меня извинить и пропустить первую часть этой главы.

Всякие строгости – Strict Mode в JavaScript

Этого точно никто не ожидал. «Стогий режим», знакомый любому Perl-программисту, теперь доступен и в JavaScript. Как и в Perl, Strict Mode накладывает ограничения на некоторые «вольности» языка, отсекает потенциальные опасности, спасает от ошибок.

Сам ECMAScript 5 обратносовместим с ECMAScript 3, со всеми его «особенностями», в том числе и с теми, которые «не рекомендуются». В строгом режиме они выбрасывают исключения или просто отключены.

Чтобы начать писать javaScript-программу в «строгом» стиле, достаточно включить в начало сценария одну строку:

```
"use strict";
```

Хотя можно и не поступать так радикально, ограничивавшись одной функцией:

```
function foo(){
"use strict"
.............
}
```

При этом на все вложенные функции Stict Mode тоже распространится. Но не наружу.

Теперь начнем нарушать правила. Будем плохими парнями!

```
"use strict";
foo = "bar";
```

Результат:

```
$ node ecma.js
foo = "bar";
    ^
ReferenceError: foo is not defined
    at Object.<anonymous> (C:\Users\Geol\node\ecma\ecma1.js:2:5)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

Ну вот, нам запретили сделать самую невинную вещь – присвоить значение необъявленной переменной. Не будь подключен режим Strict Mode, мы бы получили новую глобальную переменную, а так только ошибку. И это на самом деле очень хорошо!

Продолжим безобразничать:

```
"use strict";
var foo = { bar: true, bar: false };
```

Результат:

```
$ node ecma.js

C:\Users\Geol\node\ecma\ecma1.js:2
var foo = { bar: true, bar: false };
                       ^^^
SyntaxError: Duplicate data property in object literal not allowed in strict mode
    at Module._compile (module.js:439:25)
.....................................................................
```

Давайте посмотрим на ошибки, от которых нас (хотим мы того или нет) уберегает Strict Mode, более системно. Что именно нам теперь

нельзя? Во-первых, это синтаксические ошибки. Дублирующие имена переменных или свойств и методов объекта, некорректное использование **eval (obj.foo = eval)** или, например, **for (var eval in ...) {})** вызовет исключение SyntaxError.

Эту же ошибку вызовет почти любой вариант использования оператора **with()** – он признан устаревшим и потенциально опасным. Во-вторых, запрещены такие привычные вольности, как непрямое создание глобальных переменных (например, при попытке обратиться к переменной, которая не была объявлена). Тут будет возбуждено исключение **ReferenceError**. В-третьих, исключение **TypeError** возникает, например, когда операнд или аргумент, переданный функции, несовместим с типом, который данный оператор или функция ожидают получить. В-четверых, Srict Mode считает слова «implements», «interface», «let», «package», «private», «protected», «public», «static» и «yield» потенциально ключевыми и требует с ними соответствующего обращения. В-пятых... Хотя на этом пока хватит, просто по мере изложения остального материала я буду упоминать об исключениях, возбуждаемых в режиме Strict Mode. А сейчас посмотрим, что еще интересного принес новый стандарт.

JSON

Наконец-то формат данных JSON (JavaScript Object Notation), давно уже используемый в самых разных, иногда достаточно далеких от JavaScript областях, получил «родную» поддержку в самом языке. Она сводится к двум методам, которые, собственно, и перекрывают основные потребности работы с JSON.

Метод **JSON.parse()** преобразует сериализованную строку JSON в объект JavaScript:

```
var obj = JSON.parse('{"name":"John","surname":"Lennon" }');
console.log(obj.name );
```

Результат:

```
$ node ecma.js
John
```

Тут тоже можно использовать функции обратного вызова:

```
JSON.parse('{"name":"John","surname":"Lennon"}', function(key, value){
    console.log( key +" - "+ value);
});
```

Результат:

```
$ node ecma
name - John
surname - Lennon
~[object Object]
```

Метод JSON.stringify() преобразует объект JavaScript в сериализованную строку:

```
var obj = {name:"John",surname:"Lennon"}
var str = JSON.stringify(obj);
console.log(str);
```

Результат:

```
$ node ecma
{"name":"John","surname":"Lennon"}
```

А таким образом можно сериализовать не все, а отдельные поля объекта:

```
var obj = {name:"John",surname:"Lennon",band:"Beatles"}
var list = ["band","surname"];
var str = JSON.stringify(obj, list);
console.log(str);
```

Результат:

```
$ node ecma
{"band":"Beatles","surname":"Lennon"}
```

Ну и, конечно, функция обратного вызова:

```
var obj = {name:"John",surname:"Lennon",band:"Beatles"}
var str = JSON.stringify(obj,function(key, value){
if(key == "band"){
   return "The "+value;
}
return value;
});
console.log(str);
```

Результат:

```
$ node ecma
{"name":"John","surname":"Lennon","band":"The Beatles"}
```

Еще одно новшество стандарта ECMAScript 5 – давно реализованная в различных JavaScript-библиотеках конструкция bind(), связывающая объект (обычно DOM-объект, но на самом деле это не принципиально) с событием. Наверное, для Node.js с ее модулем event и

Event Emiter'ом это не очень важное приобретение, но для браузерного JavaScript – актуальнейшая необходимость. Хотя и в серверном коде это использовать иногда удобно:

```
var band =  { title: "The Beatles",
              setTitle: function(title){
                  this.title = title;
                   console.log(this);
              }
        };
setTimeout( band.setTitle.bind(band, "The Rolling Stones"), 1000 );
console.log(band);
```

Результат:

```
$ node ecma
{ title: 'The Beatles', setTitle: [Function] }
{ title: 'The Rolling Stones', setTitle: [Function] }
```

В EcvaScript 5 есть еще много мелких и не очень нововведений, с полным их списком можно ознакомиться, изучив описание формата [1]. Мне же хотелось бы сосредоточиться на самом существенном.

Массивы

Лично мной вот эти дополнения к возможностям языка, а именно средства работы с массивами, более интеллектуальные, чем простой цикл for, ожидались давно и с нетерпением. И EcvaScript5 не разочаровал. Посмотрим, что у нас есть теперь:

- **Array.forEach()** – последовательный обход массива. Главное отличие в использовании этой функции от аналогов, например в php, – наличие в качестве аргумента функции обратного вызова:

  ```
  var myArray = ["Федя", "Петя", "Костя"];
  myArray.forEach( function(element, index, array){
      console.log("a[" + index + "] = " + element);
  });
  ```

  Результат:

  ```
  $ node ecma.js
  a[0] = Федя
  a[1] = Петя
  a[2] = Костя
  ```

  Я думаю, что тут все понятно и объяснений не нужно. То же касается остальных некоторых приятных вещей, они просты, я просто приведу небольшие примеры использования.

- **Array.map()** – метод вызывает функцию обратного вызова для каждого элемента массива в возрастающем порядке и создаёт из них новый массив:

```
var myArray = ["Федя", "Петя", "Костя"];
var upperArray = myArray.map( function(element){
    return element.toLocaleUpperCase();
});

console.log(upperArray);
```

Результат:

```
$ node ecma.js
[ 'ФЕДЯ', 'ПЕТЯ', 'КОСТЯ' ]
```

Ну а где есть Map, там логично ждать Reduce. И метод **Array.reduce()** действительно существует:

```
var myArray = ["Федя", "Петя", "Костя"];
var result = myArray.reduce(function(x, y){
    return x + y;
},' ');
console.log(result);
var result = myArray.reduceRight(function(x, y){
    return x + y;
},' ');
console.log(result);
```

- Последним параметром метода служит начальное значение **Array.reduceRight()**, модификация **Array.reduce()**, служащая для редуцирования с конца массива (если честно, не представляю, почему вместо этого ввели специальный опциональный параметр, но это вопрос к авторам стандарта).
Результат:

```
$ node ecma.js
 ФедяПетяКостя
 КостяПетяФедя
```

- Метод **Array.filter()** также возвращает массив, «фильтруя» массив исходный, с помощью функции:

```
var myArray = ["Федя", "Петя", "Костя"];
var addArray = myArray.filter(function(value){
                    return value.length < 5 ? true : false;
                });
console.log(addArray);
```

Результат:

```
$ node ecma.js
[ 'Федя', 'Петя' ]
```

- Два «проверочных» метода **every()** и **some()** являются булевыми. Первый возвращает true, когда callback-функция вернет true для каждого элемента массива, второй же – если callback-функция вернула true хотя бы один раз:

```
var myArray = ["Федя", "Петя", "Костя"];
var result = myArray.every(function(value){
    return value.length < 5;
});
console.log(result);
var result = myArray.some(function(value){
    return value.length == 5;
})
console.log(result);
```

Результат:

```
$ node ecma.js
false
true
```

И еще три метода, при прочтении названий которых возникает только один вопрос: почему, черт возьми, их не было в языке раньше? Вот они:

- **Array.indexOf()**;
- **Array.lastIndexOf()**;
- **Array.isArray()**.

Объекты

С объектами в ECMAScript 5 все просто замечательно. Даже не так. Тем недалёким людям, кто вообще не признавал за конструкциями JavaScript гордого звания объектов, теперь придется замолчать. Судите сами.

Прежде всего контроль за расширяемостью объектов – это то, чего не хватало JavaScript, чтобы называться хоть сколько-нибудь безопасным языком. Теперь расширяемость объектов можно контролировать:

```
var obj = {};
obj.name = "John";
console.log( obj.name );
console.log( Object.isExtensible( obj ) );
Object.preventExtensions( obj );
obj.surname = "Smith";
console.log( obj.surname );
console.log( Object.isExtensible( obj ) );
```

Вывод:

```
$ node ecma.js
John
true
undefined
false
```

Нетрудно догадаться, что метод **Object.isExtensible()** предназначен для проверки расширяемости объекта, а **Object.preventExtensions()** запрещает любые расширения объекта. Попытка это сделать в режиме strict mode вызовет исключение.

Впрочем, это еще не все. Приведены в порядок свойства объектов. То есть старые свойства, дающие непосредственный доступ к своим значениям, сохранены, но появилась возможность создать свойства-«аксессоры», с геттерами, сеттерами и контролем доступа:

```
var band = {};

Object.defineProperty(band, "title", {
  value: "Kinks",
  writable: false,
  enumerable: true,
  configurable: true
});
Object.defineProperty(band, "state", {
  value: "uk",
  writable: true,
  enumerable: false,
  configurable: false,
})

console.log(band);

band.state = "Sweden";
band.title = "ABBA";
console.log(band);
console.log(band.state);

Object.defineProperty(band, "title", {writable: true});

band.title = "ABBA";
console.log(band.title);
```

Вывод:

```
$ node ecma
{ title: 'Kinks' }
{ title: 'Kinks' }
Sweden
ABBA
```

Тут мы сначала создаем «пустой» объект, затем методом **defineProperty()** добавляем ему два свойства. Метод имеет ряд аргументов, атрибутов свойства (почти все необязательны):

- **value** – значение свойства;
- **writable** – возможность изменения значения;
- **enumerable** – появление свойства в перечислении свойств объекта;
- **configurable** – конфигурируемость свойства.

Если честно, value (значение) должно присутствовать обязательно, если не применяются геттеры/сеттеры, о которых чуть позже. Атрибут writable, установленный в false, запрещает его изменять (при задействованном «строгом» режиме будет вызвано исключение). Атрибут enumerable в нашем примере бесполезен, а вот способность свойства с таким значением атрибута не быть перечисленным в конструкции

```
for (var prop in obj){
    console.log(prop);
}
```

смысл, безусловно, имеет.

Атрибут configurable, установленный в false, запрещает изменять остальные атрибуты (и себя тоже, да). Он же запрещает удалять свойство. Метод **Object.defineProperties()** позволяет определять несколько свойств за один раз:

```
var band = {};
Object.defineProperties(band, {
 "title", {
  value: "Kinks",
  writable: false,
  enumerable: true,
  configurable: true
},
 "state", {
  value: "uk",
  writable: true,
  enumerable: false,
  configurable: false,
 }
});   }
});
```

Весь набор атрибутов называется дескриптором свойства. Геттеры и сеттеры, то есть функции, вызываемые при назначении значения свойства и при его получении, тоже входят в дескриптор. Если зна-

чение свойства определяется с их помощью – value не указывается (будет ошибка):

```
var band = {};

Object.defineProperty(band, "state", {
  value: "uk",
  writable: true,
  enumerable: true,
  configurable: true,
})
Object.defineProperty(band, "title", {
  enumerable: true,
  get: function(){
      return title + " (" + band.state + ")";
  },
  set: function(value){
      title = value;
      console.log("Set value - " + value);
  }
});

console.log(band);
band.title = "Kinks";
console.log(band.title);
```

Результат:

```
$ node ecma
{ state: 'uk', title: [Getter/Setter] }
Set value - Kinks
Kinks (uk)
```

Важно понимать, что дескрипторы – это не свойства, не атрибуты, конструкции вида **band.value.writable** доступа к их свойствам не дадут. Дескрипторы не находятся в распоряжении пользователя – они не существуют как видимые атрибуты свойства, они хранятся внутри, в движке. Прочесть их значения можно с помощью специальных методов, например метода **Object.getOwnPropertyDescriptor()**:

```
var band = {};

Object.defineProperty(band, "state", {
  value: "uk",
  writable: true,
  enumerable: false,
  configurable: true,
})

console.log(Object.getOwnPropertyDescriptor(band, "state"));
```

Результат:

```
$$ node ecma
{ value: 'uk',
  writable: true,
  enumerable: false,
  configurable: true }
```

Кроме того, для объектов появились новые методы. Они не так революционны, но очень полезны.

Object.keys() – возвращает массив строк, представляющих имена всех свойств объекта (кроме тех, которые заданы с enumerable: false):

```
var band = { name: "Kinks",
             state: "uk",
           getName: function(){
               return this.name;
             }};
console.log( Object.keys(band));
```

Результат:

```
$ ecma
[ 'name', 'state', 'getName' ]
```

Object.getOwnPropertyNames() – идентичен **Object.keys()**, но возвращает имена всех свойств объекта (а не только перечислимых).

Object.seal() – «запечатывает» объект. Как это? Сейчас покажу:

```
"use strict";
var band = { name: "Kinks",
        getName: function(){
            return this.name;
        }};
band.geanre = "rock";

Object.seal( band);
band.name = "Trogs";
console.log(band);

band.state = "uk";
console.log(band);
```

Результат:

```
$ ecma
{ name: 'Trogs', getName: [Function], geanre: 'rock' }
/home/geol/node/ecma/ecma.js:12
band.state = "uk";
             ^
TypeError: Can't add property state, object is not extensible
```

```
    at Object.<anonymous> (/home/geol/node/ecma/ecma.js:12:12)
    at Module._compile (module.js:449:26)
    at Object.Module._extensions..js (module.js:467:10)
.....................................................
```

Проще говоря, применение этого метода позволяет запретить менять набор свойств объекта (добавлять новые или удалять). В то же время менять значения свойств можно – что и продемонстрировано в примере. Для полного запрета (замораживания) изменений есть другой, совсем драконовский метод **Object.freeze( )**:

```
"use strict";
var band = { name: "Kinks",
           getName: function(){
               return this.name;
           }};
band.geanre = "rock";

Object.freeze( band);
console.log(Object.isSealed(band));
console.log(Object.isFrozen(band));
band.name = "Trogs";
```

Результат:

```
$ ecma
true
true
C:\Users\Geol\node\ecma\ecma1.js:11
band.name = "Trogs";
          ^
TypeError: Cannot assign to read only property 'name' of #<Object>
    at Object.<anonymous> (/home/geol/node/ecma/ecma.js:11:11)
    at Module._compile (module.js:449:26)
...........................
```

Для чего предназначены методы **Object.isSealed()** и **Object.isFrozen()**, мне кажется, понятно. Обратите внимание: если объект «заморожен», то он считается и «запечатанным».

Осталось еще раз подчеркнуть, что все возможности, описанные выше, – это уже стандарт. И ими следует пользоваться. А вот о том, что стандартом еще не стало, мы поговорим во второй части главы.

Who's Next? ECMAScript 6

Вы, возможно, обратили внимание, что, упомянув о предыдущей версии языка, я говорил ECMAScript 3. А где четвертая версия? Она была!

Спецификация ECMAScript 4, или, как тогда называли, JavaScript 2, была представлена к утверждению еще в 2007 г. Там было много интересного – указание типов при вызове функции, реализация ООП – как классическая, так и в варианте «динамических» классов, контроль доступа, наследование [...]... Нововведения были достаточно радикальны, и именно этот факт оказался для стандарта роковым – ECMAScript 4 был отвергнут ввиду сложности реализации (довольно редкий случай в наше время). ECMAScript 4 не погиб, а в 2008 г. в немного урезанном варианте вылился в новый проект ECMAScript Harmony, а в 2009 г. была утверждена новая редакция стандарта ECMA-262 – ECMAScript 5.

С тех пор уже почти 5 лет идет непрерывное строительство светлого будущего под названием ECMAScript 6, кодовое имя ECMAScript. next. Теперь уже шестая версия стандарта планировалась к утверждению в декабре 2013 г. Затем этот срок перенесли на год, и есть надежда, что больше отсрочек не будет. Хотя... на что, собственно, надеяться? Что такого может принести нам новый стандарт, чтобы назвать его долгожданным? Давайте разберемся.

ECMAScript 6 в браузерах и в Node.js

Узнавать о новых технологиях без возможности попробовать их в деле, конечно, не очень интересно. С поддержкой этого стандарта в современный браузерах сейчас не то чтобы плохо, а скажем так – разно. В любом случае, лучше пробовать примеры из этой части в последних релизах Chrome Canary и ночных сборках Firefox. С Node.js дела обстоят лучше – начиная с версии 0.11 появилась возможность запускать платформу с флагом --harmony, дающим поддержку конструкций ECMAScript 6, правда, неполную – но полная поддержка незавершенного стандарта вряд ли возможна.

Флагов на самом деле больше, можно, допустим, с помощью --harmony_scoping подключить только блочную область видимости переменных, а с помощью --harmony_modules – поддержку модулей (обо всем этом ниже).

Да, конечно, Node.js 0.11 не является на настоящий момент стабильной версией платформы, но очень хотелось... то есть я хотел сказать, что для исследований она вполне пригодна. И давайте к ним приступим.

Вот теперь настоящие классы!

Нет, понятно, что в JavaScript/ECMAScript объекты существуют достаточно давно. Понятно, что сама модель прототипирования позво-

ляет обходиться без классов. Но все равно многим хотелось бы иметь под рукой привычные языковые структуры. И они есть:

```
class Person {
   constructor(name) {
      this.name = name;
   }

   getname() {
      return this.name;
   }
}
var user = new Person("John");
console.log(user.getName()); //  John
```

Правда, круто? А есть еще и наследование, причем классического типа:

```
class User extends Person {
   constructor(name, id) {
      super.constructor(name);
      this.id = id;
   }

   getName() {
      return super.getName() + " " + this.id;
   }
}
var user = new User("Paul", 211);
user.getName() //  Paul 211
user instanceof Person; // true
```

(Правда, множественного наследования нет, но многие, наверное, скажут – и слава богу!)

И самые настоящие статические методы:

```
class Point {
   constructor(x, y) {
      this.x = x;
      this.y = y;
   }

   static zero() {
      return new Point(0, 0);
   }
}
```

Причем старая, прототипная модель ООП в ECMAScript 6 никуда не ушла (что не может не радовать), классическое наследование – это просто еще одна возможность.

Модульная структура

Изначально в JavaScript, но с усложнением создаваемых на этой технологии приложений, потребность в этих языковых структурах появилась. Причем потребность была настолько сильной, что было создано несколько проектов модульного расширения, самыми известными из которых являются CommonJS (напомним – на этой модели основана модульная система Node.js и Asynchronous Module Definition (AMD, наиболее популярная реализация библиотеки – RequireJS). Оба подхода имеют свои особенности и свое назначение: первый ориентирован на использование на стороне сервера, второй – на асинхронную загрузку и сторону клиента.

Формат модулей ECMAScript 6 (ES6) создавался как некоторый компромисс между этими подходами, но в синтаксисе, из-за компактности и простоты, предпочтение было Node.js:

```
module Math {
   export function sum(x, y) {
      return x + y;
   }

   export var pi = 3.141593;

   // Не видна снаружи
   function internal() {
      ...
   }
}
```

Импортирование модуля:

```
import Math.{sum, pi};
alert("2π = " + sum(pi, pi));
```

Можно использовать *, чтобы импортировать всё:

```
import Math.*;
alert("2π = " + sum(pi, pi));
```

Модули можно вкладывать друг в друга:

```
module Widgets {
   module Button { ... }
   module Alert { ... }
   module TextArea { ... }
   ...
}

import Widgets.Alert.{messageBox, confirmDialog};
...
```

Модули можно подгружать из веба или через файловую систему:

```
module JSON = require('http://json.org/modules/json2.js'); // web
import JSON.*;

module File = require('io/File'); // file system
import require("bar.js").y; // file system
```

Все глобальные переменные в модуле являются глобальными только в этом модуле. Возможны циклические зависимости между модулями.

Цикл for-of

Как вы знаете, цикл for-in в JavaScript итерирует по всем полям объекта (включая наследованные). То есть итерировать по значениям массива можно, но опасно:

```
let arr = [ "blue", "green" ];
arr.notAnIndex = 123;
Array.prototype.protoProp = 456;

for(var x in arr) {
   console.log(x); // Напечатает blue, green, notAnIndex, protoProp
}
```

В ECMAScript 6 появится цикл for-of, который решит данную проблему:

```
for(var x of arr) {
   console.log(x); // Напечатает только blue, green
}
```

Функции – let it block!

В JavaScript всегда существовала проблема области видимости. Причем она настолько старая, что все уже давно с ней смирились. Состоит она в том, что все переменные, объявленные с помощью ключевого слова var, будут доступны в любом месте функции (даже если они объявлены внутри блока):

```
function local() {
   for(var i = 0;i<10;i++) {
   ...
   }

  console.log(i); // 10
}
local();
```

Это называется функциональной областью видимости. В ECMAScript 6 появится ключевое слово let, которое позволит объявлять переменные с блочной областью видимости (block scope):

```
function local() {
   if (let i=0;i<10;i++) {
     ...
   }

  console.log(i;); // ошибка!
}
local();
```

Во-первых, в новом стандарте предусмотрены параметры функций по умолчанию. Вот этого функционала точно очень не хватало. Тут ничего объяснять не надо:

```
function square(arg = 1) {
   return arg * arg;
}

square(2); //  return 4
setLevel(); // return 1
```

Я не знаю, как вы обходились без этого раньше, я – с трудом. И создатели стандарта на этом не остановились! Теперь у ECMAScript есть именованные параметры! Из примера будет все понятно:

```
function foo(id, {name, age}) {
   ...
}

foo(1, { name: 'John', age: 23 });
foo(253, { name: 'Kirill', age: 44 });
```

Тут именованным, понятно, будет второй параметр.

Arrow function

Arrow-функции (обычно употребляют не очень изящный дословный перевод – «стрелочные функции») получили свое название благодаря новому синтаксису. То есть, на первый взгляд, это действительно новый, более эффектный синтаксис для уже устоявшихся языковых конструкций. Например, вот такое выражение:

```
var arr = [ 1, 2, 3 ];
var double = arr.map(function (x) { return x * 2 });
```

можно переписать так:

```
var arr = [ 1, 2, 3 ];
let double = [ 1, 2, 3 ].map(x => x * 2);
```

Ну или совсем в общем виде, вместо

```
var ref = function(value) { return value; }
```

пишем:

```
var ref = value => value;
```

В общем, это нечто вроде аналога лямбда-выражений для JavaScript. Еще примеры:

```
var binom = (x, y)=> (x + y) * (x + y);

// то же, что и
//  binom = function(x, y) {
//    return (x + y) * (x + y);
// }

var sum = ()=> 1 + 2 + 3;

// то же, что и
//  sum = function() {
//    return 1 + 2 + 3;
// }
```

Все это хорошо и смотрится действительно красиво, но это совсем не все. Дело в том, что при использовании стрелочных функций значение специальных переменных – this, super, arguments – определяется не местом вызова, а методом логического создания. Сейчас попробую пояснить на примере. Вот обычный JavaScript-объект:

```
var getID = {
   ID: 315,
   getLogin: function (users) {
      users.forEach(function(user) {
          console.log(user + this.ID);
   });
  }
}
```

Вызовем метод getLogin:

```
getID.getLogin(['Peter', 'Paul', 'Mary']);
```

Результат:

```
Peter undefined
Paul undefined
Mary undefined
```

Что тут не так? Да все просто, переменная this в месте ее вызова указывает не на сам объект, а на очередного члена перебираемого массива. Проблема знакомая и решается введением дополнительных переменных, но в ECMAScript 6 можно поступить изящнее:

```
var getID = {
   ID: 315,
   getLogin: function (users) {
      users.forEach(user => { console.log(user + this.ID); });
   }
```

Теперь все работает, как надо:

```
Peter  315
Paul  315
Mary  315
```

Значения этих переменных – this, super и arguments – внутри стрелочных функций остаются неизменными на протяжении всего жизненного цикла функции. Это действительно означает невозможность их использования как конструктора и появление ошибки, при применении с оператором new(), что, если подумать, вполне логично.

Такое поведение переменных позволяет задавать цепочки вызовов стрелочных функций, не теряя при этом контекста:

```
intersection: function() {
        foo = [...],
        bar = [...],
        uff = [...],
        return this.arr1.reduce( (sum, s1) =>
            this.bar.reduce( (sum, s2) => (
                this.uff.reduce( (sum, s2) => {
                    sum.push( this.concatenate( s1, s2,s3 ) )
                    return sum;
                }
```

А еще Arraw-функции из-за лаконичного синтаксиса просто идеальны как передаваемые аргументы. Ну, это уже элементы функционального программирования, эту тему я развивать не буду.

Обещания

Обещания (Promise) – это еще один впечатляющий механизм будущего стандарта. Хотя почему будущего? Этим механизмом JavaScript-разработчики пользуются довольно давно. Они прямо входят в такие библиотеки, как Q, when.js, WinJS, RSVP.js, реализованы как подмножество JQuery Deffered. Обещания используют множество

Node.js-модулей. Наступающее будущее – только в том, что механизм обещаний теперь будет встроен в «нативный» JavaScript. И это, между прочим, здорово! Дело в том, что, будучи повсеместно используемыми, обещания в JavaScript-библиотеках различных реализаций сильно отличаются друг от друга как по синтаксису, так и по логике работы.

Proxy – займемся метапрограммированием на JavaScript

Прокси – это новые объекты JavaScript, для которых программист должен сам определить их поведение. Начнем с примера:

```
var p = Proxy.create({
  get: function(proxy, name) {
    return 'Hello, '+ name;
  }
});

console.log(p.World);
console.log(p.Proxy);
```

Результат:

```
$ node -harmony ecma6
Hello, World
Hello, Proxy
```

Прокси – это так называемые метаобъекты (по аналогии с метаклассами в Python). Они позволяют программисту определить поведение объекта JavaScript, расширить синтаксис, динамически создавать несуществующие методы. В примере выше мы заменили метод get() объекта p, заставив его возвращать для каждого свойства его значение с префиксом «Hello».

Вот так выглядит конструктор Proxy-объекта:

```
var proxy = Proxy.create(handler, proto);
```

Тут **handler** – объект, который определяет поведение прокси, **proto** – прототип прокси-объекта (не необязательный параметр).

handler содержит базовые ловушки (traps), перехватывающие передаваемые переменные. Они имеют следующий вид:

```
имяПерехватчика:function(передаваемые переменные)->{Тип возвращаемых данных}
```

Видов перехватчиков множество – **getPropertyDescriptor**. **GetOwnPropertyNames**, **getOwnPropertyNames** и т. д. А есть еще про-

изводные ловушки (**Derived traps**), но это уже дебри, в которые мы не полезем.

Еще одна конструкция из этой новой концепции – прокси-функция:

```
var myHandler = {
  get: function(proxy, name) {
    return 'Hello, '+ name+ '!';
  }
};
var fproxy = Proxy.createFunction(
  myHandler,
      function() { return arguments[0]; },
      function() { return arguments[1]; }
  );
console.log(fproxy(1,2));
var fp = new fproxy(1,2);
console.log(fp);
console.log(fproxy.Word);
```

Результат:

```
$ node -harmony ecma6
1
2
Hello, word
```

Конструктор прокси-функции:

```
var proxy = Proxy.createFunction(handler, callTrap, constructTrap);
```

handler и **proto** тут означают ровно то же, что и в случае с прокси-объектом.

callTrap – функция, замещающая оригинальную функцию при прямом вызове прокси-функции.

constructTrap – функция, которая будет заменять оригинальный конструктор функции при вызове через **new()** (необязательный параметр).

Прокси-функции позволяют в чистом виде «расширить» JavaScript, например создать функцию из произвольного объекта (callable object):

```
function makeCallable(obj, call, construct) {
  return Proxy.createFunction(
    new ForwardingHandler(obj),
    call,
    construct || call);
}
```

Для чего все это нужно? Вот с ответом на это замечание многие энтузиасты новой технологии справляются хуже всего. На самом деле

если такой вопрос возникает – значит, эти фокусы вам не нужны. Пока не нужны или совсем. Прокси-объекты действительно довольно специфичны, их область применения – написание абстракций виртуальных объектов и создание для них собственного API. Это может быть создание «обвертки» для существующих объектов, привнесение дополнительного функционала и вообще поведения, динамический перехват несуществующих методов. Можно создавать исполняемые экземпляры классов и виртуальные объекты, эмулирующие существующие (или несуществующие). Чтобы не создалось впечатления, что работа с прокси-объектами и вообще метапрограммирование – это лишь чистая абстракция без связи с реальностью, рекомендую посмотреть эксперименты замечательного программиста Дмитрия Сошникова на https://github.com/DmitrySoshnikov/es-laboratory. Среди них – реализация массивов с отрицательными индексами, делегирующих примесей и много чего еще интересного.

Константы

Как мы привыкли объявлять константы в JavaScrip? Естественно, задавая их имена «капсом» и надеясь, что это обстоятельство убедит коллег (да и себя) не менять их значения. Но времена меняются, и в ECMAScript 6 модификатор cons – часть стандарта:

```
const PI = 3.14;
PI = 8; // (в военное время)
console.log(PI);
```

Результат:

```
$ node -harmony ecma6
3.14
```

Генераторы

Генераторы – это функции, которые могут прерывать свое исполнение (с помощью ключевого слова **yield**) и продолжать его с того же места после вызова их метода next.

Возможно, тут нужно сказать что-то про объекты, инкапсулирующие отложенные контексты исполнения, но с моим естественно-научным образованием этого лучше даже не пытаться.

На самом деле генераторы – это очень интересный механизм, теоретически призванный покончить с извечными проблемами асинхронного программирования – Callback Hell/The Pyramid of Doom. JavaScript теперь в состоянии приостанавливать выполнение в сере-

дине тела функции-генератора и переключать контекст на исполнение чего-то другого.

Рассмотрим следующий пример:

```
function* generator() {
    var a = 2;
    while (true) {
        yield a;
        a *=a;
    }
}

function run() {
    var seq = generator();
    console.log(seq.next().value);
    console.log(seq.next().value);
    console.log(seq.next().value);
    console.log(seq.next().value);
    console.log(seq.next().value);
    console.log(seq.next().value);
}
run();
```

Результат работы подобной конструкции будет таким:

```
$ node -harmony ecma6
2
4
16
256
65536
4294967296
```

Тут пояснять почти нечего – сама функция-генератор отмечена специальным синтаксисом (*function). В отличие от обычной функции, ее вызов не начинает исполнение ее тела, а возвращает новый объект – генератор. Далее, при вызове метода **next()** этого объекта, происходит выполнение его кода до первого оператора **yield**. При срабатывании этого оператора выполнение функции прекращается, и возвращается результат, а значение переменной, служащее аргументом **yield**, становится доступным через его свойство value.

При следующем вызове метода **next()** выполнение программы будет продолжено с того же места, с полным сохранением контекста.

В генератор можно также передавать параметры – они будут подставлены в переменную – параметр **yield**:

```
function* generator() {
     var result =  yield "x";
```

```
  result += yield "y";

      return result;
}

function run() {
    var seq = generator();
    console.log(seq.next().value);
    console.log(seq.next(5).value);
 console.log(seq.next(7).value);
}
run();
```

Результат:

```
$ node -harmony ecma6
x
y
12
```

Опять же, я не буду дальше развивать тему применения генераторов, но хотел бы обратить внимание на их совершенно синхронную природу. Поэтому, если кто-то совсем запутался в бесконечных вложенных кэлбэк-вызовах, горячо их рекомендую. Как видите, JavaScript живет и развивается. Он уже почти захватил мир www, и просто страшно подумать, на что он будет способен с новыми возможностями. Хотя нет. Не страшно, скорее интересно. Посмотрим.

Заключение – что дальше?

Стоял необычно теплый май 2014 года. Разбирая почту в начале трудового дня, я натолкнулся на занятный head-hunter-спам. Предлагалась работа серверного веб-программиста на довольно неплохих условиях и с достойной компенсацией, но вот требуемый опыт работы... необходимо было не менее пяти лет трудиться, используя технологии Node.js и MongoDB. Конечно, это забавно – на вакансию подошел бы разве что сам Райан Дал, да и то только в компании с 10gen (создателями Mongo). Но если подумать, то появление таких смешных вакансий – это своего рода признак того, что Node.js – это уже не экзотика и авангард, а вполне зрелая и используемая технология.

И осознание этой несложной вещи заставило меня закончить работу над этой книгой и поставить точку. Хотя о многом хотелось бы рассказать. Платформа Node.js сейчас переживает период, который иначе как бурным развитием не назовешь. И это развитие действительно интересное. Появились средства для разработки кросс-платформенных десктопных приложений (node-webkit), так называемые веб-платформы – JavaScript-фреймворки, включающие в себя клиентскую и серверную часть (Meteor, Derby). С другой стороны, Node.js сейчас часто рассматривается как серверная часть так называемых одностраничных веб-приложений (Single-page application – SPA), где платформа работает вместе с такими продуктами, как Backbone.js, AngularJS или YUI App Framework.

В общем, эта книга – это всего лишь введение в интересный мир серверной JavaScript-разработки, в дальнейшем освоении которого я желаю читателю успехов. И меньше багов в рабочем коде.

Приложение – полезные ресурсы по платформе Node.js

1. Документация по Node.js API на сайте проекта: http://nodejs.org/api/.
2. Node.JS от А до Я – серия скринкастов Ильи Кантора: http://learn.javascript.ru/nodejs-screencast.
3. Node Packaged Modules: https://www.npmjs.org/.
4. Александр Календарев. Создание addon-модулей для Node.js: http://samag.ru/archive/more/145.
5. Сайт, посвященный технологии WebSocket: http://www.websocket.org/.
6. The WebSocket Protocol: http://tools.ietf.org/html/rfc6455.
7. Проект Socket.io: http://socket.io/.
8. Проект async: https://github.com/caolan/async.
9. ORM Sequelize: http://sequelizejs.com/.
10. Mongoose MongoDB ODM: http://mongoosejs.com/.
11. Шаблонизатор Mustache: https://github.com/janl/mustache.js.
12. Embedded JavaScript templates: https://github.com/visionmedia/ejs.
13. Jade – node template engine: http://jade-lang.com.
14. CSS-препроцессор LESS: http://lesscss.org/.
15. Препроцессор Stylus: https://github.com/LearnBoost/stylus.
16. Middleware framework Connect: https://github.com/senchalabs/connect.
17. Проект Express: http://expressjs.com/.
18. Проект nodemon: http://nodemon.io/.
19. Отладчик node-inspector: https://github.com/node-inspector/node-inspector.
20. Проект Should: https://github.com/shouldjs/should.js.
21. Проект Chai: http://chaijs.com/.
22. Проект Jasmine – Behavior-Driven JavaScript: http://jasmine.github.io/.
23. Проект Grunt – The JavaScript Task Runner: http://gruntjs.com/.
24. Проект CoffeeScript: http://coffeescript.org/.
25. Проект TypeScript: http://www.typescriptlang.org/.
26. Проект Dart: https://www.dartlang.org/.
27. ECMAScript® Language Specification (5.1 Edition): http://www.ecma-international.org/ecma-262/5.1/.
28. Таблица поддержки браузерами элементов ECMAScript6: http://kangax.github.io/compat-table/es6/.
29. ECMAScript Language Specification ECMA-262 6th Edition – DRAFT: https://geekli.st/OmShiv/links/18490.

Список литературы

1. *Баккет К.* Dart в действии. – М.: ДМК Пресс, 2013.
2. *Бейтс М.* CoffeeScript. Второе дыхание JavaScript. – М.: ДМК Пресс, 2012.
3. *Бэнкер К.* MongoDB в действии. – М.: ДМК Пресс, 2013.
4. *Лэм Ч.* Haddop в действии. – М.: ДМК Пресс, 2012.
5. *Редмонд Э., Уилсон Дж.* Семь баз данных за семь недель. – М.: ДМК Пресс, 2012.
6. *Сухов К.* HTML 5. Путеводитель по технологии. – М.: ДМК Пресс, 2012.
7. *Сухов К.* HTML 5. Путеводитель по технологии. Второе обновленное издание. – М.: ДМК Пресс, 2013.
8. *Хэррон Д.* Node.js. Разработка серверных веб-приложений на JavaScript. – М.: ДМК Пресс, 2013.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. (499) 782-38-89.
Электронный адрес: books@alians-kniga.ru.

Сухов Кирилл Константинович

Node.js. Путеводитель по технологии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 32. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

За последние несколько лет платформа Node.js стремительно повысила свой статус от экспериментальной технологии до основы для серьезных промышленных проектов. Тысячи программистов оценили возможность построения достаточно сложных, высоко нагруженных приложений на простом, элегантном и, самое главное, легковесном механизме.

Все эти скучные слова правдивы, но на самом деле не это главное. Прежде всего Node.js – это совершенно увлекательная и захватывающая вещь, с которой по-настоящему интересно работать!

Есть одна проблема – невозможно рассказывать про использование Node.js в отрыве от остальных технологий современной веб-разработки (и Highload-разработки). Я и не стал этого делать, дав обзор инструментов, без которых сегодня трудно обойтись. Прежде всего это чудесный язык JavaScript, и в книге рассказано о новинках в его последней и будущей спецификациях (EcmaScript 5 и 6). Кроме того, дается краткое введение в большинство связанных веб-технологий – от NoSQL-хранилищ данных (Memcached, MongoDB, Redis) до CSS-препроцессоров MVC JavaScript-фреймворков. Конечно, эту книгу нельзя рассматривать как полноценный учебник по MongoDB, LESS или EcmaScript 6, Dart или CoffeScript, но в ней дано основное представление об этих довольно интересных вещах, вполне достаточное для начала работы.

Для нормального восприятия книги достаточно начальных знаний языка JavaScript, общего представления об устройстве Всемирной сети и желания разобраться в самых современных веб-технологиях.

В бурной юности я трудился геологом и мне до сих пор нравится искать и находить интересные вещи. Особенно в тех технологиях, с которыми я работаю. В этой книге я хочу поделиться своими находками с читателями.

Кирилл Сухов

ISBN 978-5-97060-164-8

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф