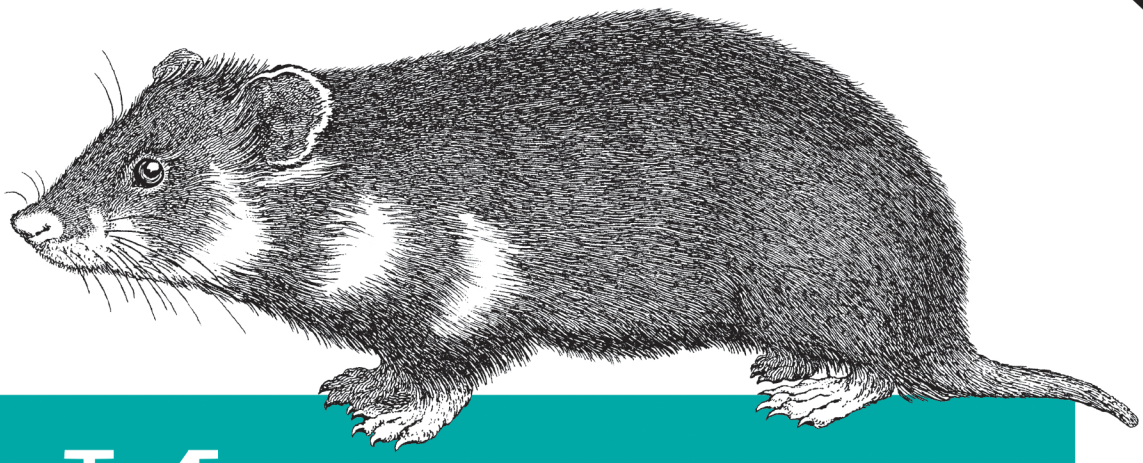
O'REILLY®

2-е ИЗДАНИЕ



Изучаем Node

Переходим на сторону сервера

Шелли Пауэрс

ПИТЕР®

SECOND EDITION

Learning Node

Moving to the Server Side

Shelley Powers

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Шелли Пауэрс

Изучаем Node

Переходим на сторону сервера

2-е ИЗДАНИЕ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2017

ББК 32.988.02-018
УДК 004.737.5
П21

Пауэрс Ш.

П21 Изучаем Node. Переходим на сторону сервера. 2-е изд., доп. и перераб. — СПб.: Питер, 2017. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02941-4

Технология Node.js всё еще молода и в то же время существует достаточно долго, чтобы крупные корпорации (LinkedIn, Yahoo! и Netflix) взяли ее на вооружение.

Эта книга посвящена Node и тем модулям, которые образуют базовую функциональность Node. Вы начнете знакомство с основ создания веб-сервера и базовых функциональностей, а затем перейдете к системе модулей, REPL, разработке приложений, проблемам безопасности, дочерним процессам, познакомитесь с новыми функциональностями, появившимися в ES6, комплексной разработкой (Express, MongoDB, Redis, AngularJS и Backbone.js), приемами разработки приложений и, наконец, с использованием Node в других областях, таких как микроконтроллеры и «интернет вещей».

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.737.5

ISBN 978-1491943120 англ.

Authorized Russian translation of the English edition of Learning Node, 2nd Edition ISBN 9781491943120 © 2016 Shelley Powers
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

ISBN 978-5-496-02941-4

Оглавление

Введение

Технология Node.js существует достаточно долго, чтобы быть принятой некоторыми крупными технологическими компаниями (LinkedIn, Yahoo! и Netflix). При этом она все еще остается достаточно молодой, чтобы вызывать определенное беспокойство у типичного корпоративного руководителя среднего звена. Когда-то она послужила движущей силой для создания более сложного языка JavaScript, а также стала единственным безопасным местом для использования нового, улучшенного сценарного языка. А поскольку долг платежом красен, передовой язык JavaScript стал движущей силой развития переработанной структуры Node.js и новой парадигмы выпуска.

Среда Node.js также переопределила то, что можно делать с JavaScript. В наши дни работодатель требует от разработчика JavaScript опыта работы в серверной среде так же часто, как опыта работы в более знакомой среде браузера. Кроме того, Node.js создает новый серверный язык, заслуживающий внимания разработчиков Ruby, C++, Java и PHP на стороне сервера, особенно если они уже владеют JavaScript.

Я считаю, что Node.js — интересная штука. По сравнению со многими другими средами начало работы с ним, создание и размещение приложения и эксперименты с новыми возможностями требуют минимальных усилий. Служебный код, необходимый для проекта Node, не такой сложный и нудный, как в других средах. Только у PHP существует настолько простая среда, но даже она требует тесной интеграции с Apache для создания приложений, ориентированных на обслуживание внешних запросов.

Впрочем, при всей простоте Node.js некоторые аспекты выяснить самостоятельно довольно сложно. Действительно, для изучения Node.js необходимо хорошее знание среды и базовых API, но вам также не обойтись без поиска и освоения этих трудноуловимых аспектов.

Для кого написана эта книга

На мой взгляд, читатели этой книги относятся к двум категориям.

В первую входят разработчики, которые создавали клиентские приложения с разными библиотеками и фреймворками и теперь хотят перенести свои навыки владения JavaScript на сторону сервера.

Ко второй категории относятся разработчики серверных приложений, которые хотят попробовать что-то новое или перейти на более современную технологию. Они работали на Java или C++, Ruby или PHP, а теперь они желают освоить язык JavaScript и объединить его со своим знанием серверной среды.

У этих двух вроде бы разных аудиторий есть одно общее знание: JavaScript (или ECMAScript, если выражаться точнее). Читатель этой книги должен очень уверенно владеть языком JavaScript. Есть и другое сходство: обеим категориям придется изучать одни и те же азы Node, включая базовый Node API.

Однако каждая категория смотрит на изучение Node со своей точки зрения и руководствуется своими навыками. Чтобы книга приносила больше пользы, я постаралась представить материал с обеих точек зрения. Например, разработчика C++ или Java может заинтересовать создание дополнений для Node на языке C++, которое вряд ли вызовет интерес у разработчика клиентских приложений. В то же время такие концепции, как прямой или обратный порядок байтов, могут быть очень знакомыми для разработчиков серверных приложений, но не для специалистов по клиентскому программированию. Я не могу глубоко изложить любую из этих перспектив, но постараюсь, чтобы материал не показался слишком сложным или скучным любому читателю.

Я совершенно точно не собираюсь заставлять вас что-либо заучивать наизусть. В книге будут представлены API базовых модулей, но я не стану описывать все объекты и все функции, потому что они документированы на сайте Node. Вместо этого я постараюсь рассмотреть важнейшие аспекты всех базовых модулей и конкретную функциональность Node, которая, на мой взгляд, особенно важна для того, чтобы вы не уступали другим разработчикам Node. Конечно, основной фактор успеха — практика, а эта книга — средство обучения. Перевернув последнюю страницу книги, вы сможете самостоятельно перейти к углубленному изучению конкретных типов функциональности, например работе с семейством технологий MEAN (Mongo-Express-Angular-Node). Материал книги станет отправной точкой, от которой вы сможете пойти в любом из интересующих вас направлений Node.

О ДОКУМЕНТАЦИИ NODE

На момент написания книги разработчики Node (а среди них и я) участвовали в обсуждении проблем, связанных с сайтом Node.js. В частности, обсуждалась проблема определения «текущей» версии Node.js, информация о которой должна отображаться при обращении к документации.

Когда я в последний раз участвовала в дискуссии, планировалось вывести на странице /docs список всех текущих версий Node.js с долгосрочной поддержкой (LTS), а также самую свежую стабильную версию и выводить индикатор версии документации Node в верхней части каждой страницы документации.

Со временем ответственные за подготовку документации хотели бы сгенерировать сводки различий в версиях API для всех страниц, но это довольно сложный проект.

На момент издания книги текущей была назначена версия Node.js 6.0.0, а от обозначения текущей ветви разработки как «стабильной» было решено отказаться. Node.js 6.0.x со временем превратится в следующую LTS-версию.

Из-за большого количества существующих версий при обращении к документации API на сайте Node.js всегда проверяйте, соответствует ли документация вашей версии Node.js. Также желательно проверить наличие новых версий, чтобы узнать о том, что вас ждет в будущем.

NODE.JS = NODE

Формально правильное название — Node.js, но им никто не пользуется. Все используют термин «Node». Точка! И в этой книге чаще всего встречается именно это название.

Структура книги

В этой книге первоочередное внимание уделено основам. Она посвящена Node и тем модулям, которые образуют базовую функциональность Node. Конечно, я упомяну ряд сторонних модулей и подробно опишу `npm`, но главная цель этой книги заключается в том, чтобы дать читателю представление о базовой функциональности Node. А когда вы уверенно встанете на ноги, вы сможете пойти в нужном направлении.

Глава 1 знакомит читателя с Node; в нее также включено краткое описание процедуры установки. Вам предоставится возможность опробовать Node на практике: сначала мы создадим веб-сервер из документации Node, а затем

более сложный сервер, код которого приведен в книге. Кроме того, будет затронута тема создания дополнений Node для читателей, программирующих на C/C++. И разве можно представить себе введение в Node без обзора истории развития среды, первой версии которой был присвоен номер 4.0 вместо 1.0?

В главе 2 рассматривается важнейшая функциональность Node: принципы обработки событий, глобальные объекты, необходимые для создания приложений Node, и асинхронный характер Node. Также будет представлен объект буфера — структуры данных, передаваемой многими сетевыми функциями Node.

Глава 3 подробно описывает систему модулей Node, а также содержит подробное описание `npm` — системы управления модулями Node от независимых разработчиков. В этой главе вы узнаете, как приложение ищет используемые модули Node и как создать собственный модуль Node. Далее для представления более сложных аспектов функциональности Node будет исследована поддержка изолированной среды (sandboxing). Просто для интереса я также представлю три популярных модуля Node от независимых разработчиков: `Async`, `Commander` и `Underscore`.

Интерактивная консоль, входящая в поставку Node и известная под названием REPL, — бесценный учебный инструмент и учебная среда. REPL посвящена отдельная глава 4. Я расскажу о том, как использовать этот инструмент, и о том, как создать вашу собственную, специализированную модификацию REPL.

Разработка приложений Node для Интернета рассматривается в главе 5. В частности, мы гораздо подробнее рассмотрим модули Node, предоставляющие средства для веб-разработки. Вы увидите, что необходимо для создания полнофункционального веб-сервера для статических файлов, и научитесь запускать приложения Node в сочетании с Apache при помощи прокси-сервера Apache.

Глава 7 посвящена сетевой поддержке, а о сетях невозможно говорить без обсуждения проблемы безопасности. Эти две темы всегда идут рука об руку — как арахисовое масло и джем, как шоколад и... что угодно. Я расскажу о поддержке TCP и UDP в Node, а также о том, как реализовать сервер HTTPS (в дополнение к серверу HTTP, создание которого рассматривалось в главе 5). Также будет описана механика, лежащая в основе цифровых сертификатов, принципы работы протокола SSL (Secure Sockets Layer) и его обновленной версии TLS (Transport Layer Security). Глава завершается описанием криптографического модуля Node и работы с хешами паролей.

Одной из наиболее приятных особенностей Node является возможность использования функциональности операционной системы при помощи дочерних процессов. Некоторые из моих любимых приложений Node — небольшие утилиты для работы со сжатыми файлами, популярное графическое приложение ImageMagick и программа для получения экранных снимков на сайтах. Конечно, они не производят впечатления на фоне модных приложений с хитроумными облачными интерфейсами, но прекрасно подходят для изучения работы с дочерними процессами. Дочерние процессы рассматриваются в главе 8.

В большинстве примеров книги используется язык JavaScript, с которым вы работаете уже не первый год. Впрочем, одной из главных причин деления Node.js/io.js и появления нового, объединенного продукта была поддержка новых версий ECMAScript, таких как ES6 (или ECMAScript 2015, если угодно). В главе 9 я расскажу о том, что в настоящее время поддерживается в Node, к каким последствиям ведет новая функциональность, а также когда и как использовать новую функциональность вместо старой. Также будут описаны некоторые тонкости, связанные с использованием новых возможностей JavaScript. Я отклонюсь от описания встроенной функциональности только один раз: когда буду описывать реализацию обещаний (promises) в очень популярном модуле Bluebird.

В главе 10 рассматриваются фреймворки и функциональность, образующие так называемую *комплексную разработку Node*. Я расскажу об Express — фреймворке, очень часто применяемом при разработке приложений Node. Также вы опробуете на практике MongoDB и Redis и познакомитесь с парой фреймворков, обеспечивающих «комплексность» в концепции комплексной разработки Node: AngularJS and Backbone.js.

После того как вы напишете код приложения Node, он отправится в эксплуатацию. В главе 11 рассматриваются инструменты и приемы разработки приложений Node и обеспечения их функционирования, включая модульное, нагрузочное и эталонное тестирование, основные методы и инструменты отладки. Кроме того, я расскажу о том, как организовать восстановление приложения после сбоя.

Глава 12 — своего рода «десерт». В ней рассматриваются некоторые возможности для переноса ваших новых навыков Node в другие области, включая мир микроконтроллеров/микрокомпьютеров как часть «Интернета вещей», и версия Node, которая не работает на базе V8.

Соглашения, используемые в этой книге

В данной книге используются следующие соглашения, связанные с типографским оформлением.

Шрифт без засечек

Служит признаком заголовков, пунктов и кнопок меню, клавиатурных комбинаций (с использованием клавиш Alt и Ctrl), а также имен файлов, расширений этих имен, путевых имен и каталогов.

Курсив

Служит признаком новых понятий.

`Моноширинный шрифт`

Служит признаком команд, переменных, атрибутов, ключей, функций, типов, классов, пространств имен, методов, модулей, свойств, параметров, значений, объектов, событий.

`Моноширинный жирный шрифт`

Для кода, который должен набираться пользователем буквально.

`Моноширинный наклонный шрифт`

Текст, который должен быть заменен значениями, предоставляемыми пользователями.

Этот элемент обозначает рекомендацию.

Этот элемент обозначает общее примечание.

Этот элемент обозначает предупреждение.

Использование примеров кода

Дополнительные материалы (примеры кодов, упражнения и т. п.) можно скачать по ссылке *https://github.com/shelleyp/LearningNode2*.

Эта книга призвана помочь вам в решении задач. По большей части вы можете использовать код из книги в своих программах и документации. Вам не нужно связываться с нами по поводу получения разрешения на это, если только вы не начнете копировать достаточно существенные фрагменты кода. Например, написание программы, в которой используется несколько фрагментов кода из этой книги, не требует разрешения. А вот продажа или распространение компакт-дисков с примерами из книг издательства O'Reilly требует разрешения. Ответы на вопросы с использованием цитат из этой книги и приведением примеров не требуют получения разрешения. А вставка существенных объемов кода примеров из этой книги в документацию потребует разрешения.

Благодарности

Я хотела бы поблагодарить людей, которые помогли мне в работе над книгой: это редактор Мэг Фоли (Meg Foley), научный редактор Итан Браун (Ethan Brown), выпускающий редактор Джиллиан Макгарви (Gillian McGarvey), корректор Джуди Макконвил (Judy McConville), иллюстратор Ребекка Панзер (Rebecca Panzer) и все остальные, кто внес свой вклад!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Среда Node

Забудьте все, что вы слышали о Node как об инструменте, предназначенном исключительно для программирования на стороне сервера. Действительно, Node в основном используется в серверных приложениях. Тем не менее Node можно установить почти на любой машине и использовать для любых целей, включая запуск приложений на вашем PC или даже на планшете, смартфоне или микрокомпьютере.

Я установила Node на своем сервере на базе Linux, а также на PC под управлением Windows 10 и на микрокомпьютере (Raspberry Pi). У меня есть планшет на базе Android, на котором я тоже планирую установить Node, я могу использовать Node для программирования микроконтроллера Arduino Uno, а в настоящее время экспериментирую со встраиванием Node в конфигурацию своего «умного дома» благодаря into IFTTT Maker Channel.

На моем PC Node используется в качестве тестовой среды для JavaScript, а также предоставляет интерфейс к ImageMagick для пакетного редактирования фотографий. Node — мой любимый инструмент для выполнения любых пакетных операций на PC или на сервере.

Да, при этом я применяю Node для обработки на стороне сервера, когда мне нужен веб-интерфейс, который работает в обход сервера Apache или предоставляет служебный серверный процесс для веб-приложения.

Суть в том, что среда Node обладает богатой функциональностью и имеет широкое распространение. Но если вы захотите поэкспериментировать с ней, придется начать с начала: с установки Node.

IFTTT — замечательный сайт, позволяющий связывать триггеры и действия различных компаний, услуг и продуктов с использованием простой логики «если-то». Каждая конечная точка в формуле представляет собой канал, включая упоминавшийся выше Maker Channel.

Установка Node

Начать установку Node проще всего со страницы Downloads проекта Node.js. С этой страницы можно загрузить двоичные файлы (заранее откомпилированные исполняемые файлы) для Windows, OS X, SunOS, Linux и ARM. Страница также предоставляет доступ к установочным программам для конкретных архитектур, которые могут очень сильно упростить процесс установки, особенно для Windows. Если ваше окружение подготовлено к сборке, загрузите исходный код и постройте Node напрямую. Для сервера Ubuntu я предпочитаю использовать именно этот вариант.

Вы также можете установить Node с использованием пакетного установщика для вашей архитектуры. Этот вариант пригодится не только для установки среды Node, но и для поддержания ее актуальности (см. раздел «LTS-версия и обновление Node», с. 29).

Если вы решите компилировать Node прямо на своей машине, вы должны настроить правильную среду сборки и установить необходимые инструменты сборки. Например, в Ubuntu (Linux) установка инструментов, необходимых для Node, выполняется следующей командой:

```
apt-get install make g++ libssl-dev git
```

При первой установке Node в разных архитектурах существуют некоторые различия. Например, в системе Windows программа установки не только устанавливает Node, но и создает окно командной строки для работы с Node на вашей машине. Node работает в режиме командной строки и не обладает графическим интерфейсом (в отличие от типичного Windows-приложения). А если вы захотите использовать Node для программирования Arduino Uno, то устанавливается как Node, так и Johnny-Five; оба приложения будут использоваться для программирования подключенного устройства.

Устанавливая Node для Windows, подтверждайте местонахождение и набор компонентов по умолчанию. Программа установки добавляет каталог Node в переменную PATH, чтобы команду node можно было вводить без указания полного пути установки Node.

Если вы устанавливаете Node на Raspberry Pi, загрузите подходящую версию для ARM (например, ARMv6 для исходной версии Raspberry Pi или ARMv7 для более новой версии Raspberry Pi 2). После загрузки извлеките двоичный файл из сжатого архива и переместите приложение в /usr/local:

```
wget https://nodejs.org/dist/v4.0.0/node-v4.0.0-linux-armv7l.tar.gz
tar -xvf node-v4.0.0-linux-armv7l.tar.gz

cd node-v4.0.0-linux-armv7l

sudo cp -R * /usr/local/
```

Вы также можете настроить среду сборки программ и построить Node напрямую.

НОВЫЕ СРЕДЫ NODE

Раз уж речь зашла о Node для Arduino и Raspberry Pi, я рассматриваю Node для нетрадиционных сред (например, «Интернета вещей») в главе 12.

Первая программа для Node

Итак, вы только что установили Node. Естественно, вам хочется опробовать новинку в деле. У программистов есть традиция: начинать программирование на новом языке с приложения «Hello, World». Приложение обычно выводит слова «Hello, World» в выходной поток (каким бы он ни был), демонстрируя тем самым, что приложение создается, запускается и может выполнять операции ввода/вывода.

Эта традиция соблюдается и для Node: приложение «Hello, World» на сайте *Node.js* включается в документацию приложения. И это первое приложение, которое мы создадим в этой книге, но с небольшими изменениями.

Простейшее приложение Hello, World

Для начала рассмотрим версию «Hello, World» из документации Node. Чтобы воссоздать приложение, создайте текстовый документ с кодом JavaScript в своем любимом текстовом редакторе. Я использую Notepad++ в системе Windows или Vim в Linux.

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

Сохраните файл под именем hello.js. Чтобы запустить приложение, откройте терминал, если вы используете OS X или Linux, или окно командной строки Node в Windows. Перейдите в каталог, в котором находится сохраненный файл, и введите следующую команду для запуска приложения:

```
node hello.js
```

Результат выводится в командной строке вызовом функции `console.log()` в приложении:

```
Server running at http://127.0.0.1:8124/
```

Теперь откройте браузер и введите адрес *http://localhost:8124/* или *http://127.0.0.1:8124* в адресной строке (или укажите свой домен, если вы разместили Node на сервере). Открывается простая веб-страница с сообщением «Hello, World» в верхней части (рис. 1.1).

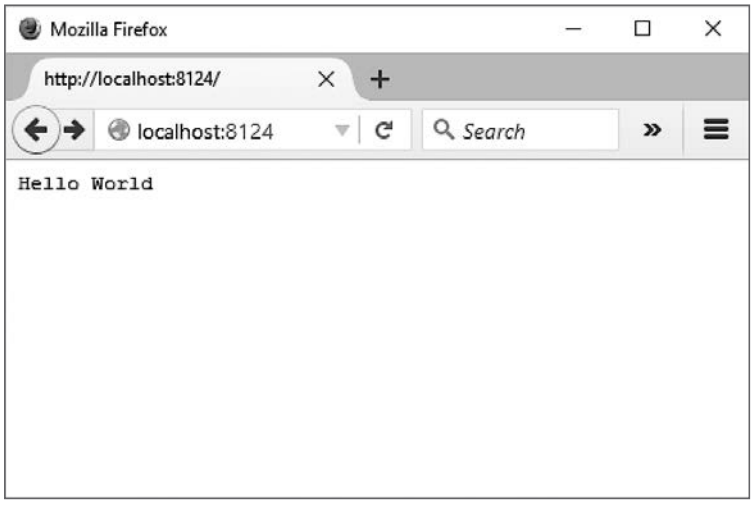


Рис. 1.1. Ваше первое приложение для Node

Если вы запускаете свое приложение в Windows, то, скорее всего, вы получите уведомление от Брандмауэра Windows (рис. 1.2). Снимите флажок Общественные сети (Public Network), установите флажок Частные сети (Private network) и щелкните на кнопке Разрешить доступ (Allow access).

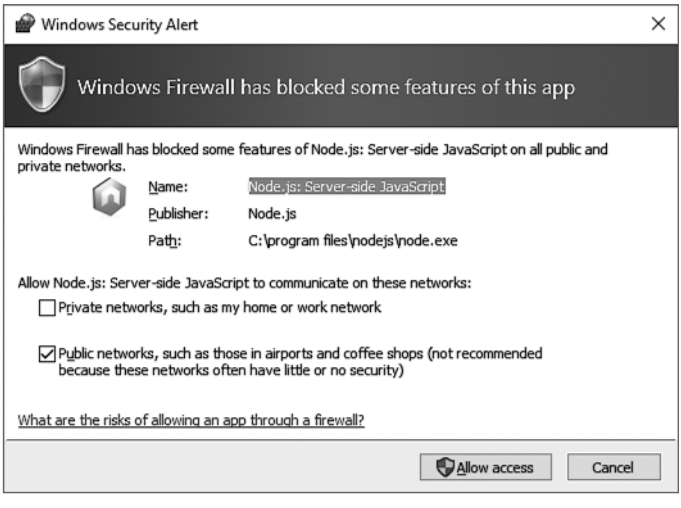


Рис. 1.2. Разрешение доступа для приложения Node в Windows

Повторять этот процесс в Windows не придется: система запоминает ваш выбор.

Чтобы завершить программу, закройте окно терминала/командной строки (далее — только терминал) или нажмите Ctrl+C. Когда вы запустили приложение, оно выполнялось в *активном* (foreground) режиме. Это означает, что во время его выполнения вы не сможете ввести в терминале никакую другую команду. Кроме того, закрытие терминала приведет к остановке процесса Node.

ЗАПУСК NODE В ФОНОВОМ РЕЖИМЕ

На этой стадии лучше запускать Node в активном режиме. Пока вы учитесь, как пользоваться этим инструментом, и не хотите, чтобы любой желающий смог получить внешний доступ к вашему приложению. А когда вы завершаете работу с ним, приложение должно завершаться. В главе 11 я покажу, как создать более устойчивую исполнительную среду Node.

Вернемся к коду программы: JavaScript создает веб-сервер, который при обращении к нему из браузера выводит веб-страницу со словами «Hello, World». Он демонстрирует несколько ключевых компонентов приложения Node.

Сначала программа включает необходимый для запуска простого сервера HTTP модуль с подходящим именем `HTTP`. Внешняя функциональность Node подключается при помощи модулей, экспортирующих определенные типы функциональности, которая может использоваться в приложении (или другом модуле). Модули очень похожи на библиотеки в других языках программирования.

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

МОДУЛИ NODE, БАЗОВЫЕ МОДУЛИ И МОДУЛЬ HTTP

Модуль HTTP входит в число базовых модулей Node, которым будет уделяться первостепенное внимание в книге. Модули Node и управление модулями рассматриваются в главе 3, а модуль HTTP описан в главе 5.

Модуль импортируется командой Node `require`, а результат присваивается локальной переменной. После импортирования локальная переменная может использоваться для создания экземпляра веб-сервера функцией `http.createServer()`. В параметрах функции встречается одна из фундаментальных конструкций Node: функция обратного вызова (callback) — листинг 1.1. Эта анонимная функция передает веб-запрос и ответ коду, который обрабатывает веб-запрос и предоставляет ответ.

Листинг 1.1. Функция обратного вызова в приложении «Hello, World»

```
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);
```

JavaScript — однопоточный (single-threaded) язык, и для имитации асинхронного выполнения в Node используется *цикл событий*, а функции *обратного вызова* вызываются при срабатывании определенного события. В листинге 1.1 функция обратного вызова вызывается при получении веб-запроса.

Сообщение `console.log()` выводится на терминал сразу же при вызове создания сервера. Программа не прекращает работу в блокирующем режиме, ожидая веб-запроса.

```
console.log('Server running at http://127.0.0.1:8124/');
```

ПОДРОБНЕЕ О ЦИКЛЕ СОБЫТИЙ И ФУНКЦИЯХ ОБРАТНОГО ВЫЗОВА

Цикл событий Node, его поддержка асинхронного программирования и функции обратного вызова более подробно рассматриваются в главе 2.

После того как сервер будет создан и получит запрос, функция обратного вызова передает браузеру простой текстовый заголовок с кодом статуса 200, выводит сообщение *Hello World* и завершает ответ.

Поздравляем! Вы только что создали свой первый веб-сервер в среде Node всего в нескольких строках кода. Конечно, пользы от такого сервера не много — если вам нужно что-то большее, чем сообщить о себе окружающему миру. В этой книге вы научитесь строить более полезные приложения Node, но прежде чем расстаться с приложением «Hello, World», мы внесем в него несколько изменений, чтобы оно стало более интересным.

«Hello,World» — новая версия

Простой вывод статического сообщения, во-первых, демонстрирует, что приложение работает, а во-вторых, показывает, как создать простой веб-сервер. Базовый пример также демонстрирует некоторые ключевые элементы всех приложений Node. Но пример получился немного пресным, хотелось бы чего-то более интересного для экспериментов. Поэтому я немного видоизменила его, чтобы у вас было более содержательное и не столь тривиальное приложение.

Обновленный код приведен в листинге 1.2. В нем я изменила базовое приложение и включила в него обработку входящего запроса. Имя выделяется из строки и используется для определения типа возвращаемого контента. Почти для любого имени будет возвращен персонализированный ответ, но если использовать в запросе параметр `name=burningbird`, вы получите изображение. Если строка запроса не используется или в нем не указано имя, переменной `name` присваивается значение `'world'`.

Листинг 1.2. Программа «Hello, World» в обновленной версии

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

http.createServer(function (req, res) {
   var name = require('url').parse(req.url, true).query.name;

   if (name === undefined) name = 'world';

   if (name == 'burningbird') {
      var file = 'phoenix5a.png';
      fs.stat(file, function (err, stat) {
         if (err) {
```

```
            console.error(err);
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end("Sorry, Burningbird isn't around right now \n");
         } else {
            var img = fs.readFileSync(file);
            res.contentType = 'image/png';
            res.contentLength = stat.size;
            res.end(img, 'binary');
         }
      });
   } else {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end('Hello ' + name + '\n');
   }
}).listen(8124);

console.log('Server running at port 8124/');
```

Результат обращения к веб-приложению с параметром `?name=burningbird` показан на рис. 1.3.

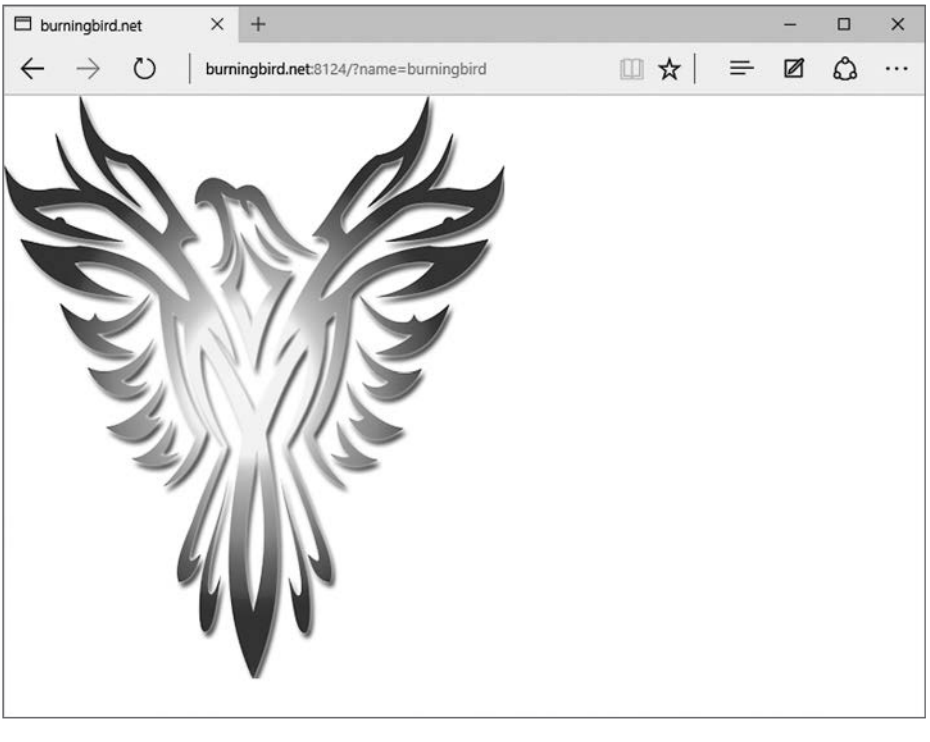


Рис. 1.3. Hello, Burningbird

Дополнительного кода не так уж много, но между базовым приложением «Hello, World» и обновленной версией существует ряд различий. В начале кода в приложение включается новый модуль с именем `fs`. Это модуль файловой системы (`File System`), с которым вы близко познакомитесь в нескольких ближайших главах. Но при этом в программе импортируется еще один модуль, но не так, как два других:

```
var name = require('url').parse(req.url, true).query.name;
```

Свойства экспортируемых модулей могут объединяться в цепочки, что позволяет импортировать модуль и использовать его функции в одной строке. Это часто происходит в модуле `URL`, единственная цель которого — предоставить инструменты для работы с URL-адресами.

Имена параметров ответа (response) и запроса (request) для удобства часто сокращаются до `res` и `req` соответственно. После разбора запроса для получения значения `name` программа сначала проверяет, равен ли параметр `undefined`. Если параметр не определен, используется значение по умолчанию `world`. Если же значение `name` существует, оно снова проверяется и сравнивается с `burningbird`. Если значения не совпадают, то приложение возвращает почти такой же ответ, как в исходном приложении, не считая того, что в возвращаемое сообщение подставляется переданное имя.

Но если параметр `name` содержит строку `burningbird`, это означает, что вместо текста придется иметь дело с изображением. Метод `fs.stat()` не только проверяет, что файл существует, но также возвращает объект с информацией о файле, включающей его размер. Значение используется для создания заголовка контента.

Если файл не существует, то приложение корректно обрабатывает ситуацию: оно выводит дружелюбное сообщение об ошибке и информацию на консоль, для чего на этот раз используется метод `console.error()`:

```
{ [Error: ENOENT: no such file or directory, stat 'phoenix5a.png']
  errno: -2,
  code: 'ENOENT',
  syscall: 'stat',
  path: 'phoenix5a.png' }
```

Если файл существует, то изображение загружается в переменную и возвращает его в ответе, изменяя значения в заголовке соответствующим образом.

Метод `fs.stat()` использует стандартный для Node паттерн функции обратного вызова с передачей значения ошибки в первом параметре. Но возможно,

часть с загрузкой изображения показалась вам несколько неожиданной. Она просто необычно выглядит и не похожа на другие функции Node, которые встречались вам в этой главе (и скорее всего, встретятся в других примерах в Интернете). Главное отличие — использование синхронной функции `readFileSync()` вместо асинхронной версии `readFile()`.

Node поддерживает как синхронную, так и асинхронную версию большинства функций файловой системы. Обычно использование синхронных операций в веб-запросах в Node считается крайне нежелательным, но такая возможность существует. Асинхронная версия того же фрагмента кода используется в примере 1.3.

Листинг 1.3. Асинхронная версия кода

```
fs.readFile(file, function(err,data) {
                res.contentType = 'image/png';
                res.contentLength = stat.size;
                res.end(data, 'binary');
              });
```

Когда какую версию стоит использовать? В некоторых обстоятельствах файловый ввод/вывод не влияет на производительность независимо от типа функции, а синхронная версия кажется более «чистой» и простой. Она также может уменьшить уровень вложенности кода — эта проблема особенно важна для системы обратного вызова Node, и она будет более подробно рассмотрена в главе 2.

Кроме того, хотя в этом примере обработка исключений не используется, с синхронными функциями можно использовать конструкцию `try…catch`. С асинхронными функциями эта более традиционная форма обработки ошибок невозможна (отсюда и передача кода ошибки в первом параметре анонимной функции обратного вызова).

Из второго примера следует одно важное замечание: ввод/вывод в Node не всегда является асинхронным.

МОДУЛИ ФАЙЛОВОЙ СИСТЕМЫ И URL, БУФЕРЫ И АСИНХРОННЫЙ ВВОД/ВЫВОД

Модуль URL более подробно рассматривается в главе 5, а модуль файловой системы — в главе 6. Впрочем, модуль файловой системы постоянно используется в книге. Буферы и асинхронная обработка рассматриваются в главе 2.

Параметры командной строки Node

В последних двух разделах Node запускается в командной строке без параметров командной строки. Я хочу кратко представить некоторые параметры, прежде чем двигаться дальше. Другие параметры будут описаны в книге там, где это будет уместно.

Чтобы получить информацию обо всех доступных параметрах, воспользуйтесь ключом `-h` или `--help`:

```
$ node --help
```

С этим параметром команда `node` выводит список всех параметров и синтаксис запуска приложения Node:

```
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]
```

Для получения версии Node используется следующая команда:

```
$ node -v или --version
```

Для проверки синтаксиса приложения Node используется параметр `-c`. С этим параметром команда `node` проверяет синтаксис без запуска приложения:

```
$ node -c or --check script.js
```

Чтобы получить информацию о параметрах V8, введите следующую команду:

```
$ node --v8-options
```

Команда выводит список разных параметров, в том числе параметр `--harmony`, используемый для включения всех завершенных функций Harmony. Сюда входит вся функциональность ES6, уже реализованная, но еще не вошедшая в версию с долгосрочной поддержкой (LTS, Long Term Support) или в текущую версию.

Мой любимый параметр Node — `-p` или `-print` — обрабатывает строку сценария Node и выводит результаты. Данная возможность особенно полезна при проверке свойств окружения процесса и более подробно рассматривается в главе 2. В следующем примере выводятся все значения свойства `process.env`:

```
$ node -p "process.env"
```

Среда хостинга Node

В процессе изучения Node вам стоит поближе познакомиться с работой Node в вашей локальной среде, будь то Windows, OS X или Linux. Когда вы будете готовы предоставить расширенный доступ к своим приложениям, вам придется найти либо среду для запуска приложения Node (например, виртуальную частную сеть — VPN (Virtual Private Network), которую использую я), либо хост, специально предоставляющий поддержку приложений Node. Первый вариант потребует существенного опыта управления сервером, доступным из Интернета, а второй нередко ограничивает то, что вы можете (или не можете) делать в своих приложениях Node.

Хостинг Node на вашем сервере, VPS или управляемом хосте

Скорее всего, хостинг Node на одном сервере с блогом на базе WordPress приведет в тупик из-за требований Node. Иметь привилегированный или административный доступ для запуска Node *не обязательно*, но желательно. Кроме того, компании, предоставляющие услуги хостинга, вряд ли обрадуются, если вы начнете предоставлять доступ к своему приложению на разных портах, которые могут породить хаос в их системах.

Хостинг Node на виртуальном частном сервере (VPS, Virtual Private Server) — как в моей конфигурации с VPN в Linode — простое решение. Вам не потребуется привилегированный доступ к VPS, и вы сможете делать практически все, что захотите, при условии, что это не создаст опасности для других пользователей, работающих на том же компьютере. Большинство компаний, предоставляющих услуги VPS, принимают меры к тому, чтобы все учетные записи были изолированы друг от друга и ни одна учетная запись не могла захватить все доступные ресурсы.

Однако с VPS возникает та же проблема, что и с собственным сервером: вам придется заниматься сопровождением сервера. В частности, вам придется настроить систему электронной почты и альтернативный веб-сервер (вероятнее всего, Apache или Nginx) для работы с брандмауэрами и другими средствами безопасности, электронной почтой и т. д. Все это не просто.

Тем не менее, если вы уверенно чувствуете себя в области управления всеми аспектами среды, доступной из Интернета, VPS может предоставить бюджетный вариант хостинга приложений Node — по крайней мере до того, как вы

будете готовы запустить приложение в эксплуатацию (в этом случае можно рассмотреть возможность хостинга приложения в *облаке*).

Облачный хостинг

В наши дни приложения размещаются на облачных серверах так же часто, как и на компьютерах отдельных лиц и групп. Приложения Node хорошо подходят для реализаций на базе облачных технологий.

Когда вы размещаете приложение Node в облаке, вы фактически строите приложение на вашем собственном сервере или PC, тестируете его, убеждаетесь в том, что оно работает так, как нужно, и пересылаете приложение на облачный сервер. Облачный сервер для Node позволяет создать нужное вам приложение Node с использованием ресурсов баз данных и любых других необходимых систем, но без прямого управления сервером. Вы можете сосредоточиться на функциональности приложения Node, не отвлекаясь на серверы FTP или электронной почты или общее управление сервером.

GIT И GITHUB: ПРЕДВАРИТЕЛЬНЫЕ УСЛОВИЯ ДЛЯ РАЗРАБОТКИ ПРИЛОЖЕНИЙ NODE

Если ранее вы еще не использовали систему управления исходным кодом Git, непременно установите ее в своей системе и научитесь пользоваться ею. Почти все операции с приложениями Node, включая отправку приложений на облачный сервер, выполняются через Git.

Система Git распространяется с исходным кодом, бесплатна и проста в установке. Программное обеспечение можно загрузить на сайте Git. Интерактивное руководство по использованию базовых команд Git доступно в GitHub.

Когда речь заходит о Git, эта система обычно ассоциируется с GitHub. Сопровождение исходного кода Node.js осуществляется на GitHub, как и код многих (если не всех) существующих модулей Node. Исходный код всех примеров этой книги также доступен на GitHub.

Вероятно, GitHub является самым большим репозиторием проектов с исходным кодом в мире, и, безусловно, это центр Вселенной Node. Это коммерческий проект, но для большинства пользователей он бесплатный. Организация GitHub предоставляет отличную документацию по использованию сайта; также существуют книги и другие учебники, которые помогут вам быстро освоить Git и GitHub. Среди них можно выделить бесплатную электронную книгу по Git (*https://git-scm.com/book/en/v2*), книгу Лолигера и Маккаллоу «Version Control with Git» (O'Reilly) и книгу Белла и Бира «Introducing GitHub» (O'Reilly).

Парадигма хостинга приложений Node в облаке на разных хостах имеет много общего. Сначала вы создаете приложение Node (локально либо на своем сервере). Когда все будет готово к тестированию среды развертывания, пора переходить к поиску облачного сервера. Для большинства известных мне серверов вы создаете учетную запись, создаете новый проект и указываете, что приложение работает на базе Node, если облачный сервер поддерживает хостинг разных сред. Возможно, вам также придется указать другие необходимые ресурсы (например, доступ к базе данных).

Когда все будет готово к развертыванию, приложение отправляется в облако. Вы либо отправляете приложение через Git, либо используете инструментарий, предоставленный поставщиком облачного сервиса. Например, облако Microsoft Azure использует Git для отправки приложений из локальной среды в облако, а Google Cloud Platform предоставляет собственный инструментарий для выполнения той же операции.

ПОИСК ХОСТА

Поиск хоста Node можно начать со страницы GitHub, посвященной этой теме (*https://github.com/nodejs/node-v0.x-archive/wiki/Node-Hosting*). Учтите, что на ней иногда встречается устаревшая информация.

LTS-версия и обновление Node

В 2014 году мир Node был потрясен (или по крайней мере некоторые из нас были сильно удивлены), когда группа людей, занимающихся сопровождением Node, отделилась от основного проекта и образовала собственную ветвь Node.js, которая называлась io.js. Раскол произошел из-за того, что люди из io.js посчитали, что Joyent (компания, занимающаяся сопровождением Node) недостаточно быстро двигается к открытому управлению Node. Также они решили, что Joyent отстает с поддержкой новейших обновлений ядра V8.

К счастью, две группы разрешили проблемы, приведшие к расколу, и снова объединили свои усилия в один продукт, который продолжает называться Node.js. Node сейчас управляется некоммерческой организацией Node Foundation под покровительством Linux Foundation. В результате кодовая база обеих групп была объединена, и вместо номера 1.0 первому официальному выпуску Node был присвоен номер версии Node 4.0: в нем отражен первоначальный медленный путь Node к версии 1.0 и более быстрый выход версии 3.0 в проекте io.js.

Новое семантическое управление версиями Node

Одним из результатов слияния стал жесткий график выхода новых версий Node на базе семантического управления версиями (Semver), знакомого пользователям Linux. Номер версии состоит из трех чисел, каждое из которых имеет определенный смысл. Например, на момент написания книги я использую на своем сервере Node.js версии 4.3.2. Этот номер расшифровывается следующим образом:

- Основная версия 4. Это число увеличивается только при внесении существенных изменений, несовместимых с предыдущими версиями Node.
- Дополнительная версия 3. Это число увеличивается при добавлении новой функциональности, но без потери обратной совместимости.
- Исправление версии 2. Это число изменяется в тех случаях, когда найденные уязвимости или другие дефекты требуют выпуска новой версии приложения. Обратная совместимость в этом случае также сохраняется.

Я использую стабильную версию 5.7.1 на машине с Wi, а код примеров был протестирован в текущей версии 6.0.0 на машине с Linux.

Node Foundation также поддерживает более логичный (хотя и создающий больше проблем) цикл выпуска по сравнению с традиционной системой. Он начался с выпуска первой версии с долгосрочной поддержкой (LTS, Long Term Support) Node.js v4, которая будет поддерживаться до апреля 2018 года. Затем Node Foundation выпустила в конце октября 2015 года первую стабильную версию Node.js v5. Версия Node 5.x.x поддерживалась до апреля 2015 года, когда ее заменила Node.js v6. По этой стратегии новая стабильная версия будет выпускаться каждые шесть месяцев, но только каждая вторая версия будет переходить в категорию LTS, как Node v4.

ВЕРСИЯ 6.0.0 КАК ТЕКУЩАЯ

В апреле 2016 года была выпущена версия 6.0.0, которая заменила 5.x и в октябре 2016 года стала новой LTS-версией. Также вместо термина «стабильная» (Stable) для обозначения версии, находящейся в активной разработке, будет использоваться термин «текущая» (Current).

После апреля 2018 года Node v4 переходит в режим сопровождения. В настоящее время ожидаются новые обновления с обратной совместимостью, а также исправления ошибок и устранения дефектов безопасности.

КАКАЯ ВЕРСИЯ РАССМАТРИВАЕТСЯ В КНИГЕ?

В книге рассматривается LTS-версия Node.js v4. Там, где это уместно, в тексте отмечаются основные различия между v4 и v5/v6.

Независимо от того, какую основную LTS-версию вы решите использовать, обязательно устанавливайте каждое новое исправление сразу же после его выпуска. С другой стороны, обновление дополнительных версий остается на ваше усмотрение и (или) на усмотрение вашей организации. При этом обновление должно сохранить обратную совместимость, затронув только усовершенствования внутреннего ядра. Несмотря на это постарайтесь включить каждую новую версию в план обновления и тестирования.

Какую версию следует использовать? Вероятно, в корпоративной среде лучше придерживаться LTS-версии, которой на момент написания книги была версия Node.js v4. Но если ваша среда способна быстро адаптироваться к критическим изменениям, вы можете получить доступ к новейшей функциональности v8 и другим полезным возможностям в последней текущей версии Node.

ТЕСТИРОВАНИЕ И ЗАПУСК В ЭКСПЛУАТАЦИЮ

Процедуры отладки и тестирования, а также другие процессы разработки и запуска приложения в эксплуатацию рассматриваются в главе 11.

Обновление Node

С ускорением выпуска новых версий задача поддержания Node в обновленном состоянии становится еще важнее. К счастью, процесс обновления проходит безболезненно и у вас есть альтернативы.

Версию Node можно проверить следующей командой:

```
node -v
```

Если вы используете программу установки пакетов, то запуск процедуры обновления пакетов приведет к обновлению Node и всего остального программного обеспечения на сервере (в системе Windows команда `sudo` не нужна):

```
sudo apt-get update
sudo apt-get upgrade --show-upgraded
```

При использовании программы установки пакетов выполните инструкции для этой программы, предоставленные на сайте Node. В противном случае возникает риск потери синхронизации с новыми выпусками.

Также для обновления Node можно воспользоваться менеджером пакетов npm. Последовательность команд обновления выглядит так:

```
sudo npm cache clean -f
sudo npm install -g
sudo n stable
```

Чтобы установить последнюю версию Node для Windows, OS X или Raspberry Pi, найдите программу установки в перечне загрузок Node.js и запустите ее. Программа устанавливает новую версию поверх старой.

NODE VERSION MANAGER

В среде Linux или OS X для обновления Node также можно воспользоваться программой nvm (Node Version Manager).

Программа npm (Node Package Manager) обновляется чаще, чем Node. Чтобы обновить только npm, выполните следующую команду:

```
sudo npm install npm -g n
```

Команда устанавливает новейшую версию нужного приложения. Версия проверяется следующей командой:

```
npm -v
```

Но учтите, что это может создать проблемы, особенно при работе в составе группы. Если участники вашей группы используют версию npm, установленную вместе с Node, а вы обновите npm вручную, то могут возникнуть проблемы с рассогласованием результатов сборки, которые довольно трудно обнаружить.

Программа npm более подробно рассматривается в главе 3, а пока просто запомните, что все модули Node можно обновить следующей командой:

```
sudo npm update -g
```

Node, V8 и ES6

В основе Node лежит ядро JavaScript. Для большинства реализаций используется ядро V8. Исходный код V8, изначально созданный компанией Google

для Chrome, был переведен в открытый доступ в 2008 году. Ядро JavaScript V8 было разработано для повышения скорости выполнения JavaScript за счет включения JIT-компилятора (Just In Time), который преобразует JavaScript в машинный код вместо того, чтобы его интерпретировать (что считалось нормой для JavaScript в течение многих лет). Ядро V8 написано на C++.

ОТВЕТВЛЕНИЕ NODE.JS ОТ MICROSOFT

Компания Microsoft создала собственное ответвление Node для версии, использующей ее ядро JavaScript (Chakra). Это ответвление должно было заложить основу видения «Интернета вещей» (IoT, Internet of Things) от Microsoft. Оно более подробно рассматривается в главе 12.

Когда была выпущена версия Node v4.0, в нее была включена поддержка V8 4.5 — той же версии ядра, которая используется Chrome. Группа сопровождения Node также обязалась поддерживать будущие версии V8 по мере их выпуска. Это означает, что Node теперь включает поддержку многих новых возможностей ECMA-262 (ECMAScript 2015 или ES6).

ПОДДЕРЖКА V8 В NODE V6

Node v6 поддерживает V8 версии 5.0. Соответственно, новые версии Node будут поддерживать новые версии V8.

В более ранних версиях Node для обращения к новым функциям ES6 при запуске приложения пришлось бы использовать флаг `--harmony`:

```
node --harmony app.js
```

Теперь поддержка функциональности ES6 базируется на следующих критериях (прямо из документации Node.js):

- Все *завершенные* возможности, которые команда V8 считает стабильными, активизируются по умолчанию для Node.js и не требуют никаких дополнительных флагов на стадии выполнения.
- *Частично реализованные* возможности, которые не считаются стабильными командой V8, активизируются специальным флагом времени выполнения: `--es_staging` (синоним `--harmony`).
- *Незавершенные* возможности активизируются по отдельности соответствующим флагом `harmony` (например, `--harmony_destructuring`), хотя делать это категорически не рекомендуется, кроме как при тестировании.

Поддержка ES6 в Node и эффективное использование различных возможностей рассматриваются в главе 9. А пока я перечислю *некоторые* возможности ES6, поддерживаемые в Node:

- Классы.
- Обещания.
- Символические имена.
- Стрелочные функции.
- Генераторы.
- Коллекции.
- `let`.
- Оператор расширения.

Дополнения C/C++

Итак, вы установили Node на своем компьютере и немного поэкспериментировали. Вероятно, вас интересует, что же именно вы установили?

Хотя язык, на котором пишутся приложения Node, базируется на JavaScript, большая часть Node на самом деле написана на C++. Обычно эта информация остается скрытой в большинстве приложений, с которыми вы работаете. Но если вы хорошо знаете C или C++, то сможете расширить функциональность Node — для этого следует написать на C/C++ *дополнение* (add-on).

Дополнения Node несколько отличаются от традиционных приложений C/C++. Во-первых, существуют специальные библиотеки, которые обычно используются в приложениях (например, библиотека V8). Во-вторых, дополнительный Node компилируется не теми инструментами, которые чаще всего используются программистами.

В документации Node приведен пример программы «Hello, World» в виде дополнения. Если вы занимались программированием на C/C++, то код примера покажется вам знакомым. А для компилирования написанного кода в файл .node используется программа node-gyp.

Работа начинается с создания конфигурационного файла с именем binding.gyp. Файл содержит информацию о дополнении в JSON-подобном формате:

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

Настройка конфигурации дополнения Node выполняется следующей командой:

```
node-gyp configure
```

Команда создает конфигурационный файл (Makefile для Unix, файл vcxproj для Windows) и помещает его в каталог build/. Чтобы построить дополнение Node, выполните следующую команду:

```
node-gyp build
```

Откомпилированное дополнение устанавливается в каталог build/release, теперь оно готово к использованию. Вы можете импортировать его в свое приложение точно так же, как вы будете импортировать другие дополнения, устанавливаемые с Node (см. главу 3).

СОПРОВОЖДЕНИЕ ПЛАТФОРМЕННЫХ МОДУЛЕЙ

Эта тема выходит за рамки книги, но если вы интересуетесь созданием платформенных модулей (дополнений), вы должны знать о различиях между платформами. Например, компания Microsoft предоставляет специальные инструкции для платформенных модулей в Azure (https://azure.microsoft.com/en-us/documentation/articles/nodejs-use-node-modules-azure-apps/), а специалист по сопровождению популярного модуля node-serialport описал проблемы, с которыми он столкнулся в процессе сопровождения модуля (http://www.voodootikigod.com/on-maintaining-a-native-node-module/).

Конечно, если вы не знаете C/C++, то, скорее всего, захотите создавать модули на JavaScript; эта тема также рассматривается в главе 3. Но если вы знаете эти языки, дополнение может стать эффективным средством расширения функциональности Node, особенно для конкретных систем.

И еще вы должны знать о довольно серьезных изменениях, происшедших с Node при переходе от v0.8 к новой версии v6.x. Чтобы справиться с возникающими проблемами, необходимо установить NAN (Native Abstractions for Node.js). Этот заголовочный файл помогает компенсировать различия между версиями Node.js.

2 Структурные элементы Node: глобальные объекты, события и асинхронная природа Node

Хотя и браузерные приложения, и приложения Node.js пишутся на JavaScript, условия их выполнения заметно различаются. Одно из принципиальных различий между Node и его браузерным «родственником» — *буфер* для двоичных данных. Правда, в Node появилась поддержка *типизованных массивов* и `ArrayBuffer` из ES6. Однако большая часть функциональности двоичных данных в Node реализуется при помощи класса `Buffer`.

Объект `buffer` принадлежит к числу глобальных объектов Node. Другой пример глобального объекта — сам объект `global`, хотя в Node он принципиально отличается от глобального объекта, к которому мы привыкли по браузеру. Разработчикам Node также доступен другой глобальный объект, `process`, — он связывает приложение Node со средой, в которой оно выполняется.

К счастью, один из аспектов Node — асинхронная природа, управляемая событиями, — должен быть хорошо знаком разработчикам интерфейсных приложений. Просто приложение Node ожидает открытия файла, а не нажатия кнопки пользователем.

Управляемость событиями также означает, что в Node доступны наши старые знакомые — функции-таймеры.

ДРУГИЕ ГЛОБАЛЬНЫЕ КОМПОНЕНТЫ

Позднее в книге будут рассмотрены и другие глобальные компоненты — require, exports, module и console. О require, exports и module будет рассказано в главе 3, а console рассматривается в главе 4.

Объекты global и process

Два важнейших объекта в Node — `global` и `process`. Объект `global` немного похож на глобальный объект в браузере, хотя между ними существуют очень серьезные различия. Объект `process`, напротив, существует только в Node.

Объект global

В браузере переменная, объявленная на верхнем уровне, объявляется глобально. В Node дело обстоит иначе. Переменная, объявленная в модуле или приложении Node, не обладает глобальной доступностью; она ограничивается модулем или приложением. Таким образом, можно объявить «глобальную» переменную с именем `str` в модуле и в приложении, использующем этот модуль, и никакого конфликта не будет.

Для демонстрации создадим простую функцию, которая прибавляет число к базовому значению и возвращает результат. Функция будет создана как библиотека JavaScript для использования в веб-странице, а также как модуль для использования в приложении Node.

Код библиотеки JavaScript из файла add2.js объявляет переменную `base`, присваивает ей значение 2, после чего прибавляет переданное число:

```
var base = 2;

function addtwo(input) {
   return parseInt(input) + base;
}
```

Затем создадим очень простой модуль, который делает то же самое, но с использованием синтаксиса модуля Node. Строение модулей более подробно рассматривается в главе 3, а пока просто скопируйте следующий код в файл с именем addtwo.js:

```
var base = 2;

exports.addtwo = function(input) {
  return parseInt(input) + base;
};
```

А теперь посмотрим, чем различается концепция глобальности в двух разных средах. Библиотека add2.js используется в веб-странице, которая также объявляет переменную `base`:

```
<!DOCTYPE html>
<html>
   <head>
      <script src="add2.js"></script>
      <script>

         var base = 10;
         console.log(addtwo(10));
      </script>
   </head>
<body>

</body>
</html>
```

При обращении к веб-странице в браузере на консоли браузера выводится значение 20 (вместо ожидаемого 12). Дело в том, что все переменные, объявленные за пределами функции в JavaScript, добавляются в один глобальный объект. Объявляя новую переменную с именем `base` в веб-странице, мы переопределяем значение во включенном файле сценария.

Теперь используем модуль `addtwo` в приложении Node:

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

var base = 10;

console.log(addtwo(base));
```

В приложении Node результат равен 12. Объявление новой переменной `base` в приложении Node никак не повлияло на значение `base` в модуле, потому что они существуют в разных глобальных пространствах имен.

Устранение общего пространства имен — безусловное усовершенствование, но оно не универсально. Объект `global` во всех средах предоставляет доступ к глобально доступным объектам и функциям `Node`, включая объект `process` (см. далее). Чтобы убедиться в этом, просто добавьте следующую команду в файл и запустите приложение. Команда выводит все глобально доступные объекты и функции:

```
console.log(global);
```

Объект process

Объект `process` принадлежит к числу важнейших компонентов среды Node, так как он предоставляет информацию о среде выполнения. Кроме того, через объект `process` выполняется стандартный ввод/вывод, вы можете корректно

завершить приложение Node и даже выдать сигнал при завершении итерации в *цикле событий* Node (см. раздел «Очередь событий (цикл)», с. 51).

Объект `process` встречается во многих приложениях в книге. А пока мы рассмотрим получение информации о среде выполнения объекта `process`, а также исключительно важные средства стандартного ввода/вывода.

Объект `process` предоставляет доступ к информации как о среде Node, так и о среде выполнения программы. Для получения информации мы воспользуемся параметром командной строки `-p`, который выполняет сценарий и возвращает полученный результат. Например, для проверки свойства `process.versions` введите следующую команду:

```
$ node -p "process.versions"
{ http_parser: '2.5.0',
  node: '4.2.1',
  v8: '4.5.103.35',
  uv: '1.7.5',
  zlib: '1.2.8',
  ares: '1.10.1-DEV',
  icu: '56.1',
  modules: '46',
  openssl: '1.0.2d' }
```

ОДИНОЧНЫЕ И ДВОЙНЫЕ КАВЫЧКИ В КОМАНДНОЙ СТРОКЕ

Обратите внимание на двойные кавычки, необходимые в окне командной строки Windows. Так как двойные кавычки работают во всех системах, используйте их для всех сценариев.

Команда выводит список различных компонентов Node и зависимостей, включая версии v8, OpenSSL (библиотека, используемая для безопасной передачи данных), собственно Node и т. д.

Свойство `process.env` предоставляет богатую информацию о том, что Node знает о вашей среде разработки:

```
$ node -p "process.env"
```

Особенно интересно проанализировать различия между архитектурами (например, Linux и Windows).

Чтобы просмотреть содержимое `process.release`, введите следующую команду:

```
$ node -p "process.release"
```

Полученный результат зависит от того, что установлено на вашем компьютере. И в LTS, и в среде текущей версии будет выведено имя приложения, а также URL исходного кода. Но в LTS также будет выведено дополнительное свойство:

```
$ node -p "process.release.lts"
'Argon'
```

Однако при обращении к тому же значению в текущей версии (например, v6) будет получен другой результат:

```
$ node -p "process.release.lts"
undefined
```

Информация о среде позволяет разработчику понять, что видит Node до и после разработки. Не включайте зависимости от этих данных прямо в приложение, потому что, как вы уже видели, они могут различаться между версиями Node. Тем не менее не жалейте времени на анализ этих данных.

Между версиями Node в любом случае должны остаться неизменными основные объекты и функции, важные для работы приложений. В их числе стандартный ввод/вывод и возможность корректного завершения приложения Node.

Стандартные потоки представляют собой заранее определенные каналы передачи данных между приложением и средой: стандартный ввод (`stdin`), стандартный вывод (`stdout`) и стандартный поток ошибок (`stderr`). В приложении Node эти каналы обеспечивают взаимодействие между приложением Node и терминалом, своего рода механизм прямого «общения» с приложением.

Node поддерживает каналы с тремя функциями `process`:

- `process.stdin`: поток с поддержкой чтения для `stdin`;
- `process.stdout`: поток с поддержкой записи для `stdout`;
- `process.stderr`: поток с поддержкой записи для `stderr`.

Вы не можете закрывать эти потоки в своем приложении. Поддерживается только чтение данных из канала `stdin` и запись в каналы `stdout` и `stderr`.

Функции ввода/вывода `process` наследуют от `EventEmitter` (см. раздел «EventEmitter», с. 58); это означает, что они могут генерировать события, а вы можете перехватывать эти события и обрабатывать любые данные. Чтобы обработать входные данные с использованием `process.stdin`, прежде всего необходимо

назначить кодировку потока. Если этого не сделать, вы получите результаты в виде буфера, а не в виде строки:

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

Затем начинается прослушивание события `readable`, которое сообщает о поступлении блока данных, готового к чтению. Затем функция `process.stdin.read()` используется для чтения данных, и если данные отличны от `null`, они копируются в `process.stdout` при помощи функции `process.stdout.write()`:

```
process.stdin.on('readable', function() {
   var input = process.stdin.read();
   if (input !== null) {
      // Эхо-вывод текста
      process.stdout.write(input);
   }
});
```

В принципе можно пропустить назначение кодировки и получить те же результаты; программа будет читать буфер и выводить буфер, но для пользователя приложения все выглядит так, словно вы работаете с текстом (строка). На самом деле это не так. Следующая функция `process`, которую мы рассмотрим, демонстрирует эти различия.

В главе 1 был создан очень простой веб-сервер, который прослушивал запрос и выводил сообщение. Чтобы завершить программу, вам придется либо уничтожить процесс с использованием сигнала, либо нажать клавиши Ctrl+C. Также возможен другой вариант: завершить приложение из кода самого приложения при помощи `process.exit()`. Вы даже можете передать информацию о том, успешно ли завершилось приложение или произошла ошибка.

Мы изменим тестовое приложение ввода/вывода так, чтобы оно «прослушивало» строку выхода и при ее обнаружении программа завершалась. Полный код приложения представлен в листинге 2.1.

Листинг 2.1. Стандартный ввод/вывод в Node и выход из приложения

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

process.stdin.on('readable', function() {
   var input = process.stdin.read();

   if (input !== null) {
      // Эхо-вывод текста
      process.stdout.write(input);
```

```
      var command = input.trim();
      if (command == 'exit')
         process.exit(0);

   }
});
```

Если запустить это приложение, любая введенная строка будет немедленно повторяться в выводе. А если ввести команду `exit`, то приложение завершается, не требуя нажатия Ctrl+C.

Если убрать вызов функции `process.stdin.setEncoding()` в начале кода, произойдет ошибка. Дело в том, что буферы не поддерживают функцию `trim()`. Можно преобразовать буфер в строку, а затем выполнить `trim`:

```
var command = input.toString().trim();
```

Но лучше просто добавить команду назначения кодировки и исключить все неожиданные побочные эффекты.

ИНТЕРФЕЙС STREAM

Объекты ввода/вывода являются реализациями интерфейса Stream, который рассматривается вместе с другими системными модулями в главе 6.

Запись в объект `process.stderr` выполняется при возникновении ошибки. Почему именно этот объект, а не `process.stdout`? По той же причине, по которой был создан канал `stderr`: чтобы отделить ожидаемый вывод от возникающих проблем. В некоторых системах вывод `stderr` даже может обрабатываться отдельно от `stdout` (например, сообщения `stdout` сохраняются в журнале, а вывод `stderr` выводится на консоль).

С объектом `process` также связаны другие объекты и полезные функции. Как упоминалось ранее, многие из них еще встретятся вам в книге.

Буферы, типизованные массивы и строки

В браузерном языке JavaScript на первых порах необходимость в обработке двоичных данных (*потока октетов*) вообще не возникала. Изначально язык JavaScript предназначался для работы со строковыми значениями, выводимыми в окнах уведомлений или на формах. Даже когда появление Ajax изменило

правила игры, данные между клиентом и сервером передавались в строковой форме (Юникод).

Однако ситуация изменилась с усложнением потребностей разработчиков. Сейчас наряду с Ajax приходится иметь дело и с WebSocket. Кроме того, возможности того, что можно делать в браузере, тоже расширились: кроме простой работы с формами появились такие новые технологии, как WebGL и Canvas.

В JavaScript и в браузере проблема решается при помощи объекта `ArrayBuffer`, для работы с которым используются *типизованные массивы*. В Node для этой цели используется объект `Buffer`.

Изначально два массива различались. Но когда io.js и Node.js объединились в Node v4.0.0, в Node также появилась поддержка типизованных массивов на базе V8 v4.5. Буферы Node в настоящее время базируются на `Uint8Array` — одном из типизованных массивов, представляющих собой массив 8-разрядных целых чисел без знака. Впрочем, это не означает, что один может использоваться вместо другого. В Node класс `Buffer` является основной структурой данных, используемой в большинстве операций ввода/вывода, и попытка заменить его типизованным массивом приведет к сбою приложения. Кроме того, преобразование буфера Node в типизованный массив возможно, но оно не обходится без проблем. Согласно документации Buffer API, при «преобразовании» буфера в типизованный массив:

- Память буфера копируется, а не открывается для общего доступа.
- Память буфера интерпретируется как массив, а не как массив байтов. Другими словами, конструкция `new Uint32Array(new Buffer([1,2,3,4]))` создает массив `Uint32Array` с четырьмя элементами [1,2,3,4], а не `Uint32Array` с одним элементом [0x1020304] или [0x4030201].

Таким образом, вы можете использовать в Node оба способа обработки потока октетов, но чаще всего будете использовать буфер. Итак, что же собой представляет буфер Node?

ЧТО ТАКОЕ «ПОТОК ОКТЕТОВ»?

Почему двоичный (низкоуровневый) файл данных здесь называется потоком октетов? Октет — единица измерения в информатике. Его длина равна 8 битам, отсюда и название «октет». В системах с 8-разрядными байтами октет и байт эквивалентны. Поток — всего лишь последовательность данных. Таким образом, двоичный файл представляет собой обычную последовательность октетов.

Буфер Node содержит низкоуровневые двоичные данные, которые были созданы за пределами кучи (heap) V8. Для управления буфером используется класс `Buffer`. После того как память будет выделена, изменить размер буфера не удастся.

Буфер по умолчанию используется для работы с файлами: если только при чтении и записи в файл не была назначена конкретная кодировка, данные читаются или записываются в буфер. В Node v4 для непосредственного создания буфера используется ключевое слово `new`:

```
let buf = new Buffer(24);
```

Учтите, что, в отличие от `ArrayBuffer`, при создании нового буфера Node его содержимое не инициализируется. Если вы хотите избежать неприятных сюрпризов при работе с буфером, который может содержать всевозможные аномальные (а возможно, конфиденциальные) данные, буфер лучше заполнить сразу же после его создания:

```
let buf = new Buffer(24);
buf.fill(0); // Буфер заполняется нулями
```

Также возможно частичное заполнение буфера с указанием начального и конечного значения.

НАЗНАЧЕНИЕ КОДИРОВКИ ДЛЯ ЗАПОЛНЕНИЯ БУФЕРА

В Node v5.7.0 появилась возможность назначения кодировки при вызове buf.fill() с использованием синтаксиса: buf.fill(строка[, начало[, конец]] [, кодировка]).

Также можно создать новый буфер напрямую, передав функции-конструктору массив октетов, другой буфер или строку. Буфер создается с копированием содержимого переданного объекта. Для строки, если она не хранится в кодировке UTF-8, необходимо указать кодировку; по умолчанию строки Node хранятся в кодировке UTF-8 (`utf8` или `utf-8`):

```
let str = 'New String';
let buf = new Buffer(str);
```

Я не стану описывать все методы класса `Buffer`, так как Node предоставляет подробную документацию по этой теме. Тем не менее некоторые составляющие функциональности заслуживают более пристального внимания.

РАЗЛИЧИЯ МЕЖДУ NODE V4 И NODE V5/V6

Типы кодировки raw и raws были исключены в Node v5 и более поздних версиях.

Однако в Node v6 на смену конструкторам пришли новые методы `Buffer` для создания буферов: `Buffer.from()`, `Buffer.alloc()` и `Buffer.allocUnsafe()`.

При передаче массива функция `Buffer.from()` возвращает буфер с копией содержимого. Однако если передать массив функции `ArrayBuffer` с необязательными параметрами смещения и длины, буфер использует ту же память, что и `ArrayBuffer`. При передаче буфера копируется содержимое буфера, а при передаче строки происходит копирование строки.

Функция `Buffer.alloc()` создает заполненный буфер определенного размера, а `Buffer.allocUnsafe()` создает буфер определенного размера, который может содержать старые или конфиденциальные данные и в дальнейшем должен быть заполнен вызовом `buf.fill()`.

Следующий код Node:

```
'use strict';

let a = [1,2,3];

let b = Buffer.from(a);

console.log(b);

let a2 = new Uint8Array([1,2,3]);

let b2 = Buffer.from(a2);

console.log(b2);
let b3 = Buffer.alloc(10);

console.log(b3);

let b4 = Buffer.allocUnsafe(10);

console.log(b4);
```

в моей системе выдает следующий результат:

```
<Buffer 01 02 03>
<Buffer 01 02 03>
<Buffer 00 00 00 00 00 00 00 00 00 00>
<Buffer a0 64 a3 03 00 00 00 00 01 00>
```

Обратите внимание на артефакты данных при использовании `Buffer.allocUnsafe()` вместо `Buffer.alloc()`.

Буферы, JSON, StringDecoder и строки UTF-8

Буфер можно преобразовать в формат JSON и в строковую форму. Сохраните следующий фрагмент в файле Node и выполните его в командной строке:

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

console.log(json);
```

Результат:

```
{"type":"Buffer",
"data":[84,104,105,115,32,105,115,32,109,121,32,112,114,101,116,
116,121,32,101,120,97,109,112,108,101]}
```

В формате JSON сначала указывается тип преобразуемого объекта `Buffer`, а затем следуют его данные. Разумеется, то, что вы видите, — данные после сохранения в буфере в виде последовательности октетов, непонятной для человека.

ES6

В большинстве примеров используется очень знакомый язык JavaScript, существующий уже несколько лет. Тем не менее время от времени я вставляю примеры из ES6. Node и ES6 (EcmaScript 2015) более подробно рассматриваются в главе 9.

Чтобы вернуться к исходному состоянию данных, можно восстановить содержимое буфера по объекту JSON, а затем воспользоваться методом `Buffer.toString()` для преобразования в строку (листинг 2.2).

Листинг 2.2. Строка преобразуется в буфер и JSON, затем обратно в буфер и в строку

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

let buf2 = new Buffer(JSON.parse(json).data);

console.log(buf2.toString()); // this is my pretty example
```

Функция `console.log()` выводит исходную строку после ее преобразования из данных буфера. Функция `toString()` по умолчанию преобразует строку в UTF-8, но если бы нам понадобилась строка другого типа, достаточно было бы указать нужную кодировку:

```
console.log(buf2.toString('ascii')); // this is my pretty example
```

Также можно задать начальную и конечную позицию преобразования:

```
console.log(buf2.toString('utf8', 11,17)); // pretty
```

Функция `Buffer.toString()` не единственный способ преобразования буфера в строку; также можно воспользоваться вспомогательным классом `StringDecoder`. Единственная цель этого объекта — декодирование содержимого буферов в строки UTF-8, но делает он это с улучшенной гибкостью и возможностями восстановления данных. Если метод `buffer.toString()` получает неполную последовательность символов UTF-8, он вернет бессмыслицу. С другой стороны, `StringDecoder` сохраняет неполную последовательность до тех пор, пока она не будет завершена, после чего возвращает результат. Если вы получаете результат в кодировке UTF-8 по фрагментам в потоке, используйте `StringDecoder`.

Различия между функциями преобразования продемонстрированы в следующем приложении Node. Знак евро (€) кодируется тремя октетами, но первый буфер содержит только два октета. Второй буфер содержит третий октет.

```
"use strict";

let StringDecoder = require('string_decoder').StringDecoder;
let decoder = new StringDecoder('utf8');

let euro = new Buffer([0xE2, 0x82]);
let euro2 = new Buffer([0xAC]);

console.log(decoder.write(euro));
console.log(decoder.write(euro2));

console.log(euro.toString());
console.log(euro2.toString());
```

В результате при использовании `StringDecoder` на консоль выводится пустая строка и вторая строка со знаком евро (€), а при использовании `buffer.toString()` — две строки абракадабры.

Строку также можно преобразовать в существующий буфер вызовом `buffer.write()`. Однако при этом важно, чтобы буфер имел правильный размер для

хранения количества октетов, необходимых для представления символов. И снова для представления знака евро необходимы три октета(0xE2, 0x82, 0xAC):

```
let buf = new Buffer(3);
buf.write('€','utf-8');
```

Этот пример также наглядно показывает, что количество символов UTF-8 не эквивалентно количеству октетов в буфере. Если у вас возникнут сомнения, проверьте размер буфера при помощи `buffer.length`:

```
console.log(buf.length); // 3
```

Операции с буфером

Для чтения и записи содержимого буфера с заданным смещением применяются различные типизованные функции. Примеры использования этих функций представлены в следующем фрагменте, который записывает в буфер четыре 8-разрядных целых числа без знака, а затем снова читает и выводит их:

```
var buf = new Buffer(4);

// Запись значений в буфер
buf.writeUInt8(0x63,0);
buf.writeUInt8(0x61,1);
buf.writeUInt8(0x74,2);
buf.writeUInt8(0x73,3);

// Вывод буфера в строковом виде
console.log(buf.toString());
```

Попробуйте выполнить этот пример: скопируйте код в файл и выполните его. Вы также можете прочитать каждое отдельное 8-разрядное число методом `buffer.readUInt8()`.

Node поддерживает чтение и запись 8-, 16- и 32-разрядных целых чисел со знаком и без, а также вещественных чисел одинарной и двойной точности. Для всех типов, кроме 8-разрядных целых чисел, также можно выбрать формат с прямым (little-endian) или обратным (big-endian) порядком байтов. Несколько примеров поддерживаемых функций:

- `buffer.readUIntLE()`: чтение значения с заданным смещением и прямым порядком байтов.
- `buffer.writeUInt16BE()`: запись 16-разрядного целого без знака с заданным смещением и обратным порядком байтов.

- `buffer.readFloatLE()`: чтение вещественного числа одинарной точности с заданным смещением и прямым порядком байтов.
- `buffer.writeDoubleBE()`: запись 64-разрядного вещественного числа двойной точности с заданным смещением и обратным порядком байтов.

ПОРЯДОК БАЙТОВ

Для тех, кто не знаком с концепцией порядка байтов: этот формат определяет способ хранения данных в памяти, то есть должен ли по меньшему адресу памяти храниться старший (более значимый) байт (обратный порядок) или же младший (менее значимый) байт.

Диаграмма на рис. 2.1, взятая из описания порядка байтов в Википедии, наглядно демонстрирует различия между ними.

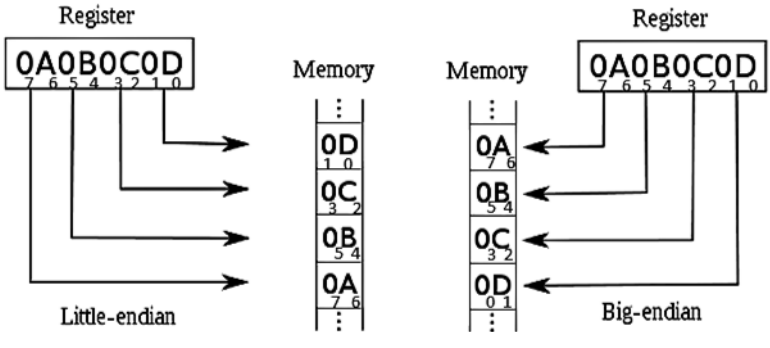


Рис. 2.1. Прямой и обратный порядок байтов (с разрешения Википедии)

8-разрядные целые числа также можно записывать напрямую в формате, сходном с форматом массива:

```
var buf = new Buffer(4);

buf[0] = 0x63;
buf[1] = 0x61;
buf[2] = 0x74;
buf[3] = 0x73;
```

Кроме чтения и записи по заданному смещению вы также можете создать *срез* — новый буфер, содержащий часть старых данных, вызовом `buffer.slice()`. В этой возможности особенно интересно то, что изменение содержимого нового буфера также приводит к изменению содержимого старого буфера.

В листинге 2.3 для демонстрации этой особенности на базе строки создается буфер, срез существующего буфера используется для создания нового, после чего содержимое нового буфера модифицируется. Далее оба буфера выводятся на консоль, чтобы вы могли убедиться в изменении данных «на месте».

Листинг 2.3. Демонстрация автоматического изменения старого буфера при изменении нового буфера

```
var buf1 = new Buffer('this is the way we build our buffer');
var lnth = buf1.length;

// Создание нового буфера как среза старого
var buf2 = buf1.slice(19,lnth);
console.log(buf2.toString()); // build our buffer

//Изменение второго буфера
buf2.fill('*',0,5);
console.log(buf2.toString()); // ***** our buffer

// Проверка содержимого первого буфера
console.log(buf1.toString()); // this is the way we ***** our buffer
```

Если вам потребуется проверить буферы на совпадение содержимого, используйте функцию `buffer.equals()`:

```
if (buf1.equals(buf2)) console.log('buffers are equal');
```

Вы также можете скопировать байты из одного буфера в другой вызовом `buffer.copy()`. Дополнительные параметры позволяют скопировать все байты или их часть. Учтите, что, если размер второго буфера недостаточен для размещения всего содержимого, будет скопирована только помещающаяся часть байтов:

```
var buf1 = new Buffer('this is a new buffer with a string');

// Копирование буфера
var buf2 = new Buffer(10);
buf1.copy(buf2);

console.log(buf2.toString()); // this is a
```

Для сравнения байтов используется метод `buffer.compare()`, который возвращает признак относительного расположения сравниваемых буферов в лексикографическом порядке. Если сравниваемый буфер предшествует второму, то возвращается значение –1; если нет — значение 1. Если два буфера содержат одинаковые байты, то возвращается 0:

```
var buf1 = new Buffer('1 is number one');
var buf2 = new Buffer('2 is number two');

var buf3 = new Buffer(buf1.length);
buf1.copy(buf3);

console.log(buf1.compare(buf2)); // -1
console.log(buf2.compare(buf1)); // 1
console.log(buf1.compare(buf3)); // 0
```

Существует еще один класс буфера `SlowBuffer`, который может использоваться, если вам когда-нибудь потребуется сохранить содержимое небольшого буфера в течение продолжительного времени. Обычно Node выделяет память для небольших буферов (менее 4 Кбайт) в заранее выделенном блоке памяти. Это делается для того, чтобы механизму уборки мусора не приходилось отслеживать и освобождать большое количество мелких блоков памяти.

Класс `SlowBuffer` позволяет создавать мелкие буферы за пределами заранее выделенного блока памяти, чтобы они могли существовать в течение более долгого периода времени. Впрочем, как нетрудно представить, этот класс может существенно повлиять на быстродействие программы. Используйте его только в том случае, если *никакое* другое решение не подходит.

Обратные вызовы и асинхронная обработка событий в Node

Язык JavaScript является однопоточным, то есть синхронным по своей природе. Это означает, что код JavaScript выполняется строка за строкой до тех пор, пока приложение не будет завершено. Так как среда Node базируется на JavaScript, она наследует это однопоточное синхронное поведение.

Но если в приложении используется функциональность, требующая ожидания некоторого условия (например, открытия файла), получения веб-ответа или другой аналогичной операции, блокирование приложения до завершения операции создаст серьезную критическую точку в серверном приложении.

Для предотвращения блокировки используется *цикл событий*.

Очередь событий (цикл)

Для достижения асинхронной функциональности приложение может пойти по одному из двух путей. Первый вариант — выделить каждый продолжительный

процесс в отдельный программный поток (thread). В это время остальной код выполняется параллельно. Недостаток многопоточного решения заключается в том, что потоки обходятся дорого. Они требуют значительных затрат ресурсов и сильно повышают сложность приложения.

Второй вариант — использование событийной архитектуры. В этом случае при запуске процесса, занимающего много времени, приложение не ожидает его завершения. Вместо этого процесс сигнализирует о своем завершении выдачей события. Событие помещается в специальную очередь (цикл событий). Зависимая функциональность регистрирует свой интерес к данному событию в приложении, и когда событие извлекается из очереди и обрабатывается, зависимая функциональность активизируется и получает данные, сопровождающие событие.

JavaScript в браузере и Node выбирают второй подход. В браузере при назначении элементу обработчика щелчка вы регистрируете событие (подписываетесь на него) и определяете функцию обратного вызова, которая будет вызываться при возникновении события; таким образом, выполнение остального кода приложения может продолжаться:

```
<div id="someid"> </div>
<script>
   document.getElementById("someid").addEventListener("click",
                                        function(event) {
       event.target.innerHTML = "I been clicked!";
   }, false);
</script>
```

Node использует собственный цикл событий, но вместо ожидания событий пользовательского интерфейса (например, щелчков на элементах) этот цикл обеспечивает поддержку серверной функциональности — прежде всего ввода/вывода. События могут сообщать об открытии файла; о том, что его содержимое прочитано в буфер; о получении веб-запроса от пользователя и т. д. Все эти процессы могут не только занимать много времени, но и создавать значительную конкуренцию за ресурсы, а каждое обращение к ресурсу обычно блокирует этот ресурс от других обращений до завершения исходного процесса. Кроме того, приложения на базе веб-технологий зависят от действий пользователя, а иногда и от действий других приложений.

Node обрабатывает все события в очереди по порядку. Добравшись до события, которое вас интересует, Node вызывает указанную вами функцию обратного вызова (callback) и передает ей всю информацию, связанную с событием.

В простейшем веб-сервере, созданном в первом примере из главы 1, был представлен цикл событий в действии. Я повторю код, чтобы вам было проще следить за происходящим:

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

В листинге 2.4 я изменила код, чтобы выделить отдельные действия и прослушивать дополнительные события, возникающие при создании сервера, подключении клиента и прослушивании.

Листинг 2.4. Базовый веб-сервер с дополнительными событиями

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

var server = http.createServer();

server.on('request', function (request, response) {

   console.log('request event');

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

server.on('connection', function() {
   console.log('connection event');
});

server.listen(8124, function() {

   console.log('listening event');
});

console.log('Server running on port 8124');
```

Обратите внимание: `requestListener()`, функция обратного вызова серверного запроса, уже не вызывается в функции `http.createServer()`. Вместо этого приложение присваивает только что созданный сервер HTTP переменной, которая затем используется для перехвата двух событий:

- Событие `request`, генерируемое каждый раз при выдаче веб-запроса клиентом.
- Событие `connection`, генерируемое каждый раз, когда новый клиент подключается к веб-приложению.

В обоих случаях для подписки на события используется функция `on()`, наследуемая классом сервера HTTP от класса `EventEmitter`. Объект, от которого наследуется эта функциональность, будет рассмотрен в следующем разделе, а пока займемся самими событиями. В этом примере приложение подписывается еще на одно событие — событие прослушивания, для которого функция обратного вызова используется с функцией HTTP `server.listen()`.

Итак, имеется один объект — сервер HTTP и три события — `request`, `connection` и `listening`. Что же происходит при создании приложения и выдаче веб-запросов?

При запуске приложения немедленно выводится сообщение о том, что сервер выполняется на порте 8124. Это объясняется тем, что приложение не блокируется при создании сервера, подключении клиента или когда начинается прослушивание запросов. Таким образом, сообщение `console.log()` впервые отрабатывает после прохождения всех неблокирующих асинхронных функций.

Следующее сообщение — «listening event». Сразу же после создания сервера необходимо начать прослушивание новых подключений и запросов. Для этого вызывается функция `server.listen()`. Дожидаться событий «сервер создан» не нужно, так как функция `http.createServer()` немедленно возвращает управление. Вы можете убедиться в этом, вставив сообщение `console.log()` прямо после вызова функции `http.createServer()`. Если вы добавите эту строку, она первой будет выведена на консоль при запуске приложения.

В предыдущей версии приложения вызов `server.listen()` был объединен в цепочку с вызовом `http.createServer()`, но это не обязательно. Такая конструкция — вопрос удобства и элегантности кода, с генерированием событий она не связана. С другой стороны, `server.listen()` — асинхронная функция с функцией обратного вызова, активизируемой при выдаче события прослушивания. Таким образом, сообщение будет выведено *после* сообщения о выполнении сервера на порте 8124.

Никакие другие сообщения не выводятся до тех пор, пока к веб-приложению не подключится клиент. Тогда выводится сообщение о событии подключения (`connection`), потому что это событие первым генерируется при появлении нового клиента. За ним выводится одно или два сообщения о событии запроса

(`request`); различия связаны с особенностями выдачи запросов к новым сайтам в браузере. Chrome запрашивает ресурс, но также запрашивает и значок сайта favicon.ico, поэтому приложение получает два запроса. Firefox и IE этого не делают, поэтому для этих браузеров приложение получает только одно сообщение о запросе.

Если обновить запрос страницы в том же браузере, вы получите только сообщение(-я) о событии запроса. Подключение уже установлено, и оно сохраняется до тех пор, пока пользователь не закроет браузер или не произойдет какой-нибудь тайм-аут. При обращении к одному ресурсу из разных браузеров для каждого из них генерируется отдельное событие подключения.

При обращении к веб-приложению из Chrome на консоль выводится следующая последовательность сообщений:

- *Server running on port 8124.*
- *Listening event.*
- *Connection event.*
- *Request event.*
- *Request event.*

Если у вас в модуле или прямо в приложении определяется функция, которая должна быть асинхронной, она определяется по специальным правилам, описанным в следующем разделе.

Создание асинхронной функции обратного вызова

Чтобы продемонстрировать фундаментальную структуру функциональности обратных вызовов, в листинге 2.5 представлено полноценное приложение Node, в котором создается объект с единственной функцией `doSomething()`. Функция получает три аргумента: первый возвращается как данные при отсутствии ошибки, второй должен содержать строку, а в третьем передается функция обратного вызова. Если второй аргумент `doSomething()` отсутствует или не содержит строку, то объект создает новый объект `Error`, который возвращается функцией обратного вызова. Если ошибки не происходит, вызывается функция обратного вызова, коду ошибки присваивается `null`, а функция возвращает данные (в этом случае первый аргумент.)

Ключевые элементы функциональности обратного вызова выделены жирным шрифтом в листинге 2.5.

Листинг 2.5. Фундаментальная структура функциональности обратного вызова

```
var fib = function (n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
};

var Obj = function() { };

Obj.prototype.doSomething = function(arg1_) {
   var callback_ = arguments[arguments.length - 1];
   callback = (typeof(callback_) == 'function' ? callback_ : null);
   var arg1 = typeof arg1_ === 'number' ? arg1_ : null;

   if (!arg1)
      return callback(new Error('first arg missing or not a number'));

      process.nextTick(function() {

         // Блокирующая операция
         var data = fib(arg1);
         callback(null, data);
    });
}

var test = new Obj();
var number = 10;

test.doSomething(number, function(err,value) {
      if (err)
         console.error(err);
      else
         console.log('fibonaci value for %d is %d', number, value);
});

console.log('called doSomething');
```

Первое, что нужно сделать, — проследить за тем, что в последнем аргументе передается функция обратного вызова, а в первом аргументе функции обратного вызова — код ошибки. Мы не можем определить намерения пользователя, но можем убедиться в том, что последний аргумент содержит функцию, и этого достаточно.

Второй ключевой аспект — создание объекта Node `Error` при возникновении ошибки и возвращение его как результата функции обратного вызова. В асинхронной архитектуре положиться на `throw...catch` нельзя, так что

обработка ошибок должна выполняться в объекте `Error` в функции обратного вызова.

Последний критичный аспект функциональности — вызов функции обратного вызова с передачей данных функции при отсутствии ошибки. Но чтобы обратный вызов был асинхронным, мы вызываем его в функции `process.nextTick()`. Причина в том, что `process.nextTick()` гарантирует, что цикл событий будет очищен перед вызовом функции. Это означает, что вся синхронная функциональность будет обработана до активизации блокирующей функциональности (если она есть). В примере блокировка происходит не из-за ввода/вывода, а из-за того, что операция нагружает процессор. Возможно, вызов функции, генерирующей последовательность Фибоначчи для значения 10, займет немного времени, но со значением 50 и более (в зависимости от ресурсов вашей системы) задержка станет более заметной. Функция Фибоначчи вызывается в `process.nextTick()`; это гарантирует, что функциональность, нагружающая процессор, будет выполняться асинхронно.

Короче говоря, все остальное может изменяться, но следующие четыре ключевых аспекта должны присутствовать всегда:

- Убедитесь в том, что последний аргумент содержит функцию обратного вызова.
- Создайте объект Node `Error` и верните его в первом аргументе функции обратного вызова, если произошла ошибка.
- Если ошибки нет, вызовите функцию обратного вызова, присвойте аргументу ошибки `null` и передайте все нужные данные.
- Функция обратного вызова должна вызываться из `process.nextTick()`, чтобы процесс не блокировался.

Если заменить значение `number` на 10, приложение выдаст на консоль следующий результат:

```
called doSomething
[Error: first argument missing or not a number]
```

Если просмотреть код в каталоге `lib` установки Node, вы увидите, что паттерн с функцией обратного вызова в последней позиции постоянно встречается в нем. Функциональность может изменяться, но паттерн остается постоянным.

Этот метод достаточно прост, и он обеспечивает непротиворечивость результатов асинхронных методов.

ВЛОЖЕНИЕ ОБРАТНЫХ ВЫЗОВОВ

Использовать функции обратного вызова несложно, но они создают свои проблемы, включая вложение обратных вызовов. Эта проблема и способы ее решения рассматриваются в главе 3, в разделе «Управление обратными вызовами с использованием Async», с. 101.

Ранее я упоминала о том, что объект `http.Server` наследует от другого объекта, и именно таким образом он наделяется способностью генерирования событий. Этот объект с именем `EventEmitter` будет рассмотрен в следующем разделе.

NODE — ОДНОПОТОЧНАЯ СРЕДА... В ОСНОВНОМ

Цикл событий Node является однопоточным. Однако это не означает, что во время его выполнения в фоновом режиме не работают другие программные потоки.

Node обращается к функциональности (например, функциональности файловой системы fs), реализованной на C++, а не на JavaScript. Функциональность fs использует рабочие потоки для достижения своих целей. Кроме того, Node использует библиотеку libuv, которая использует пул рабочих потоков для реализации своей функциональности. Конкретное количество потоков зависит от операционной системы.

Если вы будете ограничиваться JavaScript и создавать модули JavaScript, вам никогда не придется беспокоиться о рабочих потоках и libuv. Факт: меня похлопали по плечу и предложили не забивать голову мыслями о рабочих потоках. В прошлом мне доводилось работать с многопоточными средами, так что меня такой подход устраивает.

Но если вы интересуетесь разработкой дополнений для Node, вам придется очень близко познакомиться с libuv. Начать можно с введения в libuv (*https://nikhilm.github.io/uvbook/basics.html*).

За дополнительной информацией об интересном, тайном мире многопоточности в Node я рекомендую обратиться к ответам на вопрос об использовании пула потоков на сайте Stack Overflow (*http://stackoverflow.com/questions/22644328/when-is-the-thread-pool-used*).

EventEmitter

Стоит немного поскрести многие базовые объекты Node, и под оболочкой обнаружится `EventEmitter`. Каждый раз, когда вы видите объект, генерирующий событие, которое обрабатывается функцией `on`, знайте: перед вами `EventEmitter`. Понимание того, как работает класс `EventEmitter`, и умение его использовать — две важнейшие составляющие программирования для Node.

Класс `EventEmitter` обеспечивает асинхронную обработку событий в Node. Чтобы продемонстрировать его базовую функциональность, напишем короткое тестовое приложение.

Начнем с включения модуля `Events`:

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

Затем создадим экземпляр `EventEmitter`:

```
var em = new events.EventEmitter();
```

Только что созданный объект `EventEmitter` используется для выполнения двух важнейших задач: присоединения обработчика к событию и генерирования самого события. Обработчик события `EventEmitter.on()` активизируется при генерировании конкретного события. Первый параметр метода содержит имя события; во втором передается функция обратного вызова для выполнения некоторой функциональности:

```
em.on('someevent', function(data) { ... });
```

Событие генерируется для объекта методом `EventEmitter.emit()` при выполнении некоторого условия:

```
if (somecriteria) {
   en.emit('data');
}
```

В листинге 2.6 создается экземпляр `EventEmitter`, который генерирует хронометражное событие через каждые три секунды. В функции обработчика этого события на консоль выводится сообщение со счетчиком. Обратите внимание на связь между аргументом `counter` в функции `EventEmitter.emit()` и соответствующими данными в функции `EventEmitter.on()`, обрабатывающей событие.

Листинг 2.6. Простейшая проверка функциональности EventEmitter

```
var eventEmitter = require('events').EventEmitter;
var counter = 0;

var em = new eventEmitter();

setInterval(function() { em.emit('timed', counter++); }, 3000);

em.on('timed', function(data) {
  console.log('timed ' + data);
});
```

При запуске приложения хронометражные сообщения о событиях выводятся на консоль до момента завершения приложения. Главный вывод, который следует сделать из этого простого примера: событие генерируется функцией `EventEmitter.emit()`, а функция `EventEmitter.on()` может использоваться для перехвата этого события и его обработки.

Этот пример интересен, но не особенно полезен. Нас в данном случае интересует возможность добавления функциональности `EventEmitter` к существующим объектам, а не использование экземпляров `EventEmitter` в приложениях. Именно эту задачу решают `http.Server` и многие другие классы с поддержкой событий в Node.

Функциональность `EventEmitter` наследуется, и чтобы наследование стало возможным, приходится использовать другой объект Node, `Util`. Модуль `Util` импортируется в приложение следующей командой:

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

Модуль `Util` чрезвычайно полезен. Основная его функциональность будет рассмотрена в главе 11, когда мы займемся отладкой приложений Node. Впрочем, одна из функций — `util.inherits()` — потребуется прямо сейчас.

Функция `util.inherits()` позволяет одному конструктору унаследовать методы прототипа другого — *суперконструктора*. Чтобы функция `util.inherits()` была еще более интересной, вы также можете обращаться к суперконструктору прямо в функциях конструктора.

Функция `util.inherits()` позволяет унаследовать функциональность очереди сообщений Node от `EventEmitter`в любом классе:

```
util.inherits(Someobj, EventEmitter);
```

В результате вызова `util.inherits()` с объектом вы можете вызвать метод `emit` с методами объекта и программировать обработчики для экземпляров объекта:

```
Someobj.prototype.someMethod = function() { this.emit('event'); };
...
Someobjinstance.on('event', function() { });
```

Вместо того чтобы разбираться, как `EventEmitter` работает в абстрактном смысле, перейдем к листингу 2.7, в котором продемонстрировано использование класса, наследующего функциональность `EventEmitter`. В приложении создается новый класс `inputChecker`. Конструктор получает два значения, имя

человека (`name`) и имя файла (`file`). Он присваивает первое значение свойству, а также создает ссылку на поток с поддержкой записи при помощи метода `createWriteStream` модуля `File System`.

Объект также содержит метод `check`, который проверяет входные данные на конкретные команды. Одна команда (`wr:`) генерирует событие записи, а другая (`en:`) — событие завершения. Если команда не задана, генерируется событие эхо-вывода. Экземпляр объекта предоставляет обработчики для всех трех событий. Для события записи выполняется запись в выходной файл, для ввода без команды выполняется эхо-вывод входных данных, а для события завершения приложение завершается методом `process.exit`.

Все входные данные поступают из стандартного входного потока (`process.stdin`). Для записи выходных данных используется поток с возможностью записи; таким образом, на заднем плане создается новый приемник вывода, в очередь которого будут поступать будущие операции записи. Такой способ вывода в файл более эффективен, особенно если ожидается частое выполнение операций, как в данном приложении. Эхо-вывод реализуется простым выводом в `process.stdout`.

Листинг 2.7. Создание объекта, наследующего от EventEmitter

```
"use strict";

var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

function InputChecker (name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0o666});
};

util.inherits(InputChecker,eventEmitter);

InputChecker.prototype.check = function check(input) {

  // Удаление лишних пропусков
  let command = input.trim().substr(0,3);

  // Обработка команды
  // Команда wr: входные данные записываются в файл
```

```
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));

  // Команда en: процесс завершается
  } else if (command == 'en:') {
     this.emit('end');

  // Эхо-вывод в стандартный выходной поток
  } else {
     this.emit('echo',input);
  }
};

// Тестирование нового объекта и обработки событий
let ic = new InputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.on('echo', function( data) {
   process.stdout.write(ic.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

// Получение ввода после назначения кодировки
process.stdin.setEncoding('utf8');
process.stdin.on('readable', function() {
   let input = process.stdin.read();
   if (input !== null)
      ic.check(input);
});
```

Обратите внимание: функциональность также включает метод обработчика события `process.stdin.on`, так как `process.stdin` — один из многих объектов Node, наследующих от `EventEmitter`.

ЗАПРЕТ НА ВОСЬМЕРИЧНЫЕ ЛИТЕРАЛЫ В РЕЖИМЕ STRICT

В листинге 2.7 выбран режим strict, потому что я использую команду ES6 let. Однако из-за действия режима strict я не могу использовать восьмеричные литералы (например, 0666) в флагах дескрипторов выходного файла. Вместо этого используется запись 0o666 — литерал в стиле ES6.

Функция `on()` в действительности является сокращенной формой записи для функции `EventEmitter.addListener`, получающей те же параметры. Таким образом, следующий фрагмент:

```
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

в точности эквивалентен:

```
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

Для прослушивания следующего события можно использовать метод `EventEmitter.once()`:

```
ic.once(event, function);
```

При превышении десяти слушателей для события по умолчанию выводится предупреждение. Чтобы изменить количество максимальных слушателей, используйте команду `setMaxListeners` с числом. Значение 0 соответствует неограниченному количеству слушателей.

Слушателей можно удалять вызовом `EventEmitter.removeListener()`:

```
ic.on('echo', callback);
ic.removeListener('echo',callback);
```

Этот вызов удаляет из массива слушателей одного слушателя с сохранением порядка остальных.

Однако если вы по какой-то причине скопировали массив слушателей с использованием `EventEmitter.listeners()`, вам придется создать его заново после удаления слушателей.

Цикл событий Node и таймеры

В браузере для работы с таймерами существуют `setTimeout()` и `setInterval()`; эти функции также доступны и в Node. Они не совсем равноценны, потому что браузер использует цикл событий, реализуемый на уровне ядра браузера, а цикл событий Node реализуется библиотекой C++, libuv, но в основном различия между ними несущественны.

Функция Node `setTimeout()` получает в первом параметре функцию обратного вызова, во втором — время задержки (в миллисекундах), а также необязательный список аргументов:

```
setTimeout(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

console.log("waiting on timer...");
```

Имя в списке аргументов передается в аргументе функции обратного вызова в `setTimeout()`. Продолжительность задержки устанавливается равной 3000 миллисекунд. Сообщение `console.log()` «waiting on timer...» выводится почти немедленно, так как функция `setTimeout()` выполняется асинхронно.

Таймер можно отменить, если ваше приложение присвоит его переменной при создании. Я изменю предыдущую версию приложения Node и включу в него ускоренную отмену и вывод сообщения:

```
var timer1 = setTimeout(function(name) {
             console.log('Hello ' + name);
           }, 30000, 'Shelley');

console.log("waiting on timer...");

setTimeout(function(timer) {
            clearTimeout(timer);
            console.log('cleared timer');
           }, 3000, timer1);
```

Таймер устанавливается на очень долгий период времени. У нового таймера будет достаточно времени для вызова функции обратного вызова, которая непосредственно отменяет таймер.

Функция `setInterval()` работает аналогично `setTimeout()`, если не считать того, что таймер продолжает повторно срабатывать, пока приложение не будет завершено или таймер не будет сброшен вызовом `clearInterval()`. В следующем примере вместо `setTimeout()` демонстрируется использование `setInterval()`: сообщение повторяется девять раз перед отменой таймера.

```
var interval = setInterval(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

setTimeout(function(interval) {
            clearInterval(interval);
            console.log('cleared timer');
```

```
          }, 30000, interval);

console.log('waiting on first interval...');
```

Как указано в документации Node, нет гарантий того, что функция обратного вызова будет вызвана ровно через *n* миллисекунд (для любого *n*). Этим она не отличается от использования `setTimeout()` в браузере — у нас нет полного контроля над окружением и разные факторы могут вызвать небольшую задержку таймера. В большинстве случаев расхождения в периодичности срабатывания функций таймера не ощущается. Впрочем, если вы создаете анимации, задержка может стать заметной.

Существует две функции, специфичные для Node, которые могут использоваться с таймерами/интервалами, возвращаемыми при вызове `setTimeout()` или `setInterval()`: `ref()` и `unref()`. Если вызвать `unref()` для таймера и это единственное событие в очереди событий, то таймер отменяется, а программа сможет завершиться. Если вызвать `ref()` для того же объекта таймера, программа будет выполняться до тех пор, пока таймер не будет обработан.

Вернемся к первому примеру, создадим таймер с большей продолжительностью, вызовем для него `unref()` и посмотрим, что произойдет:

```
var timer = setTimeout(function(name) {
            console.log('Hello ' + name);
          }, 30000, 'Shelley');

timer.unref();

console.log("waiting on timer...");
```

Запущенное приложение выводит сообщение на консоль, а затем завершается. Причина в том, что таймер, установленный вызовом `setTimeout()`, является единственным событием в очереди событий приложения. А если добавить еще одно событие? Изменим программу, добавим интервал и тайм-аут и вызовем `unref()` для тайм-аута:

```
var interval = setInterval(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

var timer = setTimeout(function(interval) {
            clearInterval(interval);
            console.log('cleared timer');
           }, 30000, interval);

timer.unref();

console.log('waiting on first interval...');
```

Таймер получает возможность продолжать; это означает, что он завершает интервал. Однако именно интервальные события позволили таймеру просуществовать достаточно долго для того, чтобы таймер мог сбросить интервал.

Последняя пара таймерных функций уникальна для Node: `setImmediate()` и `clearImmediate()`. Функция `setImmediate()` создает событие, но это событие имеет более высокий приоритет, чем события, созданные `setTimeout()` и `setInterval()`. Однако при этом оно не превосходит по приоритету события ввода/вывода и с ним не связывается собственный таймер. Событие `setImmediate()` генерируется после всех событий ввода/вывода, до событий таймера, и в текущей очереди событий. Если вызвать его из функции обратного вызова, то оно помещается в следующий цикл событий после завершения того цикла, в котором оно было вызвано. Фактически эти функции позволяют добавить событие в текущий или в следующий цикл событий без лишних таймеров. Этот способ эффективнее `setTimeout(callback, 0)`, потому что он превосходит по приоритету другие события таймеров.

Он похож на другую функцию — `process.nextTick()`, не считая того, что функция обратного вызова `process.nextTick()` активизируется после завершения текущего цикла событий, но до добавления каких-либо новых событий ввода/вывода. Как было показано ранее в разделе «Создание асинхронной функции обратного вызова» на с. 55, эта функция используется исключительно для реализации асинхронной функциональности Node.

Вложенные обратные вызовы и обработка исключений

В клиентских приложениях JavaScript нередко встречаются конструкции следующего вида:

```
val1 = callFunctionA();
val2 = callFunctionB(val1);
val3 = callFunctionC(val2);
```

Функции вызываются поочередно, результат предыдущего вызова передается следующей функции. Так как все функции синхронны, нам не нужно беспокоиться о возможном нарушении последовательности выполнения — неожиданные результаты исключены.

В листинге 2.8 представлен относительно частый пример последовательного программирования такого рода. Приложение использует синхронные версии

методов модуля File System Node для открытия файла и получения его данных, модификации данных с заменой всех вхождений «apple» на «orange» и выводом полученной строки в новый файл.

Листинг 2.8. Последовательное синхронное приложение

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

try {
   var data = fs.readFileSync('./apples.txt','utf8');
   console.log(data);
   var adjData = data.replace(/[A|a]pple/g,'orange');

   fs.writeFileSync('./oranges.txt', adjData);
} catch(err) {
  console.error(err);
}
```

Так как при выполнении могут возникнуть проблемы и мы не можем быть уверены, что ошибки будут обработаны во внутренней реализации функций модуля, все вызовы функций заключаются в блок try для корректной (или хотя бы более содержательной) обработки ошибок. Ниже показано, как выглядит ошибка, если приложение не может найти файл для чтения:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

Может, это и не самый удобный для пользователя вариант, но он намного лучше альтернативы:

```
$ node nested2
fs.js:549
  return binding.open(pathModule._makeLong(path), stringToFlags(flags),
mode);
                 ^
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
    at Object.fs.openSync (fs.js:549:18)
    at Object.fs.readFileSync (fs.js:397:15)
    at Object.<anonymous>
               (/home/examples/public_html/learnnode2/nested2.js:3:18)
    at Module._compile (module.js:435:26)
    at Object.Module._extensions..js (module.js:442:10)
    at Module.load (module.js:356:32)
```

```
    at Function.Module._load (module.js:311:12)
    at Function.Module.runMain (module.js:467:10)
    at startup (node.js:136:18)
```

Преобразование синхронной схемы последовательного приложения в асинхронную реализацию потребует пары изменений. Во-первых, все функции необходимо заменить их асинхронными аналогами. Также необходимо учесть, что каждая функция не блокируется при вызове, а это означает, что мы не можем гарантировать правильную последовательность выполнения, если функции будут вызываться независимо друг от друга. Гарантировать, что функции будут вызываться в правильной последовательности, можно только одним способом — использованием вложения обратных вызовов.

В листинге 2.9 приведена асинхронная версия приложения из листинга 2.8. Все вызовы функций файловой системы заменены их асинхронными версиями, а функции вызываются в правильном порядке посредством вложения. Кроме того, блок `try...catch` исключен из программы.

Мы не можем использовать `try...catch`, потому что из-за использования асинхронных функций блок `try...catch` будет реально обработан до вызова асинхронной функции. Таким образом, попытка инициирования ошибки в функции обратного вызова равносильна инициированию ошибки за пределами процесса, эту ошибку перехватывающего. Вместо этого ошибка обрабатывается напрямую: если она существует, то функция обрабатывает ее и возвращает управление; если же ошибки нет, процесс функции обратного вызова продолжает выполнение.

Листинг 2.9. Приложение из листинга 2.8, преобразованное с использованием асинхронных вложенных обратных вызовов

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
      console.error(err);
   } else {

     var adjData = data.replace(/apple/g,'orange');

     fs.writeFile('./oranges.txt', adjData, function(err) {

        if (err) console.error(err);
     });
   }
});
```

В листинге 2.9 выполняется открытие и чтение входного файла, и только при завершении обеих операций вызывается функция обратного вызова. Эта функция проверяет переменную ошибки; если переменной присвоено значение, то объект ошибки выводится на консоль. При отсутствии ошибки происходит обработка данных с вызовом асинхронного метода `writeFile()`. Его функция обратного вызова имеет только один аргумент: объект ошибки. Если он отличен от `null`, то он также выводится на консоль.

Если в ходе выполнения происходит ошибка, она выглядит так:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

Если вы захотите просмотреть трассировку стека ошибки, выведите свойство `stack` объекта ошибки Node:

```
if (err) {
   console.error(err.stack);
}
```

Результат выглядит примерно так:

```
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
```

Включение последовательного асинхронного вызова функции добавляет новый уровень вложения обратных вызовов и может усложнить обработку ошибок. В листинге 2.10 приложение получает список файлов в каталоге. В каждом файле обобщенное доменное имя заменяется конкретным доменным именем с использованием строкового метода `replace`, а результат записывается *обратно* в исходный файл. Информация обо всех измененных файлах записывается в открытый выходной поток.

Листинг 2.10. Получение списка изменяемых файлов

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

writeStream.on('open', function() {
   // Получение списка файлов
   fs.readdir('./data/', function(err, files) {
```

```
      // Для каждого файла
      if (err) {
         console.log(err.message);
      } else {
         files.forEach(function(name) {

            // Изменение содержимого
            fs.readFile('./data/' + name,'utf8', function(err,data) {

               if (err){
                  console.error(err.message);
               } else {
                  var adjData = data.replace(/somecompany\.com/g,
                              'burningbird.net');

                  // Запись в файл
                  fs.writeFile('./data/' + name, adjData, function(err)
                    {
                     if (err) {
                        console.error(err.message);
                     } else {

                        // Вывод информации об изменении файла
                        writeStream.write('changed ' + name + '\n',
                        'utf8', function(err) {
                           if(err) console.error(err.message);
                        });
                     }
                  });
               }
            });
         });
      }
   });
});

writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

Прежде всего мы видим нечто новое: применение механизма обработки событий для обработки ошибок при вызове функции `fs.createWriteStream`. Причина, по которой мы используем обработку событий, заключается в том, что `createWriteStream` выполняется асинхронно, поэтому мы не сможем использовать традиционный механизм `try...catch`. В то же время `createWriteStream` не предоставляет функцию обратного вызова, в которой можно было бы

перехватывать ошибки. Вместо этого мы перехватываем событие ошибки и обрабатываем его, выводя сообщение об ошибке. После этого проверяется событие открытия (успешная операция) и выполняются операции с файлом.

Приложение выводит сообщение об ошибке напрямую.

Хотя все выглядит так, словно приложение обрабатывает каждый файл по отдельности, прежде чем переходить к следующему, запомните, что все методы, использованные в приложении, являются асинхронными. Если вы запустите приложение несколько раз и просмотрите файл log.txt, вы увидите, что файлы обрабатываются в разном порядке, который кажется случайным. В моей системе подкаталог data содержит пять файлов. При троекратном запуске приложения в файл log.txt были выведены следующие данные (пустые строки вставлены для наглядности):

```
changed data1.txt
changed data2.txt
changed data3.txt
changed data4.txt
changed data5.txt

changed data2.txt
changed data4.txt
changed data3.txt
changed data1.txt
changed data5.txt

changed data1.txt
changed data2.txt
changed data5.txt
changed data3.txt
changed data4.txt
```

Другая проблема возникает, если вы хотите узнать, когда все файлы были изменены, чтобы что-то сделать. Метод `forEach` вызывает функции обратного вызова итератора асинхронно, так что выполнение не блокируется. Если вы добавите команду после `forEach`, например такую:

```
console.log('all done');
```

вывод сообщения вовсе не означает, что приложение завершено, а лишь то, что `forEach` не блокируется. Если вы добавите команду `console.log` для вывода информации об измененном файле:

```
// Вывод информации об изменении файла
writeStream.write('changed ' + name + '\n',
'utf8', function(err) {
    if(err) {
      console.log(err.message);
    } else {
      console.log('finished ' + name);
    }
 });
```

и включите следующую команду после вызова `forEach`:

```
console.log('all finished');
```

то вывод на консоль может выглядеть так:

```
all finished
finished data3.txt
finished data1.txt
finished data5.txt
finished data2.txt
finished data4.txt
```

Что же делать? Добавьте счетчик, увеличивающийся с каждым сообщением, и проверяйте его значение по длине массива файлов для вывода сообщения «all done»:

```
// Перед обращением к каталогу
var counter = 0;
...
  // Вывод информации об изменении файла
  writeStream.write('changed ' + name + '\n',
  'utf8', function(err) {

     if(err) {
        console.log(err.message);
     } else {
        console.log ('finished ' + name);
        counter++;
        if (counter >= files.length) {
           console.log('all done');
        }
     }
  });
```

На этот раз вы получаете ожидаемый результат: сообщение «all done» выводится после того, как все файлы были обновлены.

Приложение работает хорошо — если не считать того, что каталог, к которому мы обращаемся, содержит не только файлы, но и другие подкаталоги. Если приложение обнаруживает подкаталог, оно выдает следующую ошибку, хотя и продолжает обработку прочего содержимого:

```
EISDIR: illegal operation on a directory, read
```

Код в листинге 2.11 предотвращает такие ошибки, используя метод `fs.stats` для получения объекта, представляющего данные команды Unix `stat`. Объект содержит информацию о заданном элементе каталога, в том числе признак того, является он файлом или нет. Конечно, метод `fs.stats` также является асинхронным и требует дополнительного вложения обратных вызовов.

Листинг 2.11. Добавление проверки каждого элемента каталога на то, что он является файлом

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {flags : 'a',
       encoding : 'utf8',
       mode : 0666});

writeStream.on('open', function() {

   var counter = 0;

   // Получение списка файлов
   fs.readdir('./data/', function(err, files) {
      // Для каждого файла
      if (err) {
         console.error(err.message);
      } else {
         files.forEach(function(name) {

            fs.stat('./data/' + name, function (err, stats) {

               if (err) return err;
               if (!stats.isFile()) {
                  counter++;
                  return;
               }

               // Изменение содержимого
               fs.readFile('./data/' + name,'utf8', function(err,data) {

                  if (err){
```

```
                    console.error(err.message);
                } else {
                    var adjData = data.replace(/somecompany\.com/g,
                            'burningbird.net');

                    // Запись в файл
                    fs.writeFile('./data/' + name, adjData,
                                                function(err) {

                        if (err) {
                            console.error(err.message);
                        } else {

                            // Вывод информации об изменении файла
                            writeStream.write('changed ' + name + '\n',
                             function(err) {

                                if(err) {
                                    console.error(err.message);
                                } else {
                                    console.log('finished ' + name);
                                    counter++;
                                    if (counter >= files.length) {
                                        console.log('all done');
                                    }
                                 }
                            });
                        }
                    });
                }
            });
        });
    }
  });
});

writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

Приложение выполняет свою задачу, и выполняет ее хорошо, но как же трудно читать и сопровождать этот код! Я использую `return` при обработке ошибок, чтобы избавиться от одного условного вложения, но нормальное сопровождение такого кода становится почти невозможным. Я слышала, как подобные конструкции с вложенными обратными вызовами называют «спагетти

обратных вызовов», и даже более красочно — «пирамида погибели»... Оба термина уместны.

Вложенные обратные вызовы неуклонно сдвигаются к правой части документа, отчего разработчику становится еще труднее убедиться в том, что он работает с правильным кодом, относящимся к нужному обратному вызову. Тем не менее разбить вложение невозможно, потому что очень важно, чтобы методы вызывались по порядку:

1. Начать просмотр каталога.
2. Отфильтровать подкаталоги.
3. Прочитать содержимое каждого файла.
4. Изменить содержимое.
5. Записать данные обратно в исходный файл.

Хотелось бы найти способ реализации подобных серий вызовов методов, не зависящий от вложенных обратных вызовов. Для этого следует воспользоваться модулями независимых разработчиков и другими решениями. В главе 3 я использую модуль `Async` для разрушения «пирамиды погибели». А в главе 9 мы разберемся, поможет ли в решении этой задачи ES6.

Другой подход заключается в определении именованной функции как функции обратного вызова для каждого метода. Такое решение «сглаживает» пирамиду и может упростить отладку. Тем не менее оно не решает других проблем, например определения момента завершения всех процессов. Для этого вам все равно потребуется модуль управления асинхронными вызовами.

3 Модули Node и менеджер пакетов Node (npm)

В этой книге вам предоставится возможность поработать с несколькими модулями Node — в основном *базовыми* модулями. Такие модули включаются в установку Node и встраиваются в приложение глобальной командой `require`.

В этой главе мы поближе познакомимся с концепцией модуля Node и подробно рассмотрим команду `require`. Кроме того, я представлю npm (Node Package Manager) — менеджер пакетов Node и несколько модулей, которые не являются базовыми, но, по мнению многих людей, абсолютно необходимы для разработки любых приложений Node.

Система модулей Node

Базовая реализация Node остается настолько простой, насколько это возможно. Вместо того чтобы встраивать все возможные компоненты прямо в Node, разработчики предоставляют дополнительную функциональность в виде модулей.

Система модулей Node построена по образцу *системы модулей CommonJS*, механизма создания взаимодействующих модулей. Центральное место в системе занимает контракт, который должен выполняться разработчиками, чтобы их модули нормально взаимодействовали с другими модулями.

Некоторые требования системы модулей CommonJS, реализованные в Node:

- Поддержка функции `require`, которая получает идентификатор модуля и возвращает экспортируемый API.

- Имя модуля представляет собой строку символов и может включать в себя символы слеша (для идентификации путей).
- Модуль должен явно экспортировать всю функциональность, которая должна быть доступна за его пределами.
- Переменные являются приватными для модуля.

Часть функциональности Node глобальна; это означает, что вам не придется ничего делать для ее включения. Тем не менее бóльшая часть функциональности Node подключается в приложения с использованием системы модулей.

Как Node находит и загружает модуль

Если вы хотите предоставить приложению доступ к модулю Node (базовому модулю или модулю, установленному за пределами приложения), используйте команду `require`:

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

Вы также можете обратиться к одному конкретному свойству экспортируемого объекта. Например, при использовании модуля `URL` разработчики часто используют только функцию `parse()`:

```
var name = require('url').parse(req.url, true).query.name;
```

Или же вы создаете объект модуля для дальнейшего использования в приложении:

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

Когда ваше приложение включает модуль, сначала Node проверяет, находится ли этот модуль в кэше. Вместо того чтобы перезагружать модуль при каждом включении, Node кэширует его при первом обращении. Кэширование устраняет задержку, связанную с необходимостью поиска файла системой.

ФАЙЛЫ И МОДУЛИ: ОДНОЗНАЧНОЕ СООТВЕТСТВИЕ

Node позволяет создавать только один модуль в каждом файле.

Если модуль не кэширован, то Node проверяет, является ли он платформенным (native) модулем. Платформенные модули представляют собой

предварительно откомпилированные двоичные файлы (как, например, дополнения C++, о которых упоминалось в главе 1). Если модуль является платформенным, используется функция этого модуля, возвращающая экспортированную функциональность.

Если модуль не кэширован и не является платформенным, для него создается новый объект `Module` и возвращается свойство `exports` модуля. Экспортирование функциональности модулями более подробно рассматривается в разделе «Создание и публикация собственных модулей Node» на с. 92, а пока при этом возвращается общедоступная функциональность приложения.

Модуль также кэшируется. Если вы по какой-то причине захотите удалить модуль из кэша, это можно сделать:

```
delete require('./circle.js');
```

Модуль заново загружается при следующем включении его приложением.

В процессе загрузки модуля среда Node должна определить его местонахождение. Поиск файла модуля представляет собой иерархию проверок.

Прежде всего базовые модули обладают приоритетом. Вы можете присвоить своему модулю имя `http`, если захотите, но при загрузке Node возьмет базовую версию. Использовать `http` в качестве имени модуля можно только в одном случае: если вы также укажете путь, который будет отличать его от базового модуля:

```
var http = require ('/home/mylogin/public/modules/http.js');
```

Как показывает этот пример, при включении абсолютного или относительного пути в имя файла Node использует этот путь. Следующая команда ищет модуль в подкаталоге local:

```
var someModule = require('./somemodule.js');
```

Я указываю расширение модуля, но это необязательно. Если указать только имя модуля без расширения, Node сначала ищет в текущем подкаталоге модуль с расширением .js. Если модуль будет найден, то он загружается. Если модуль не найден, то Node ищет файл с расширением .json. Если файл с подходящим именем и расширением .json будет найден, то содержимое модуля загружается как данные в формате JSON. Наконец, Node ищет модуль с расширением .node. Среда предполагает, что этот модуль представляет собой предварительно откомпилированное дополнение Node, и обрабатывает его соответствующим образом.

Файлы JSON не требуют явного включения команды exports. От них требуется лишь одно: правильный формат данных JSON.

Также возможно использование более сложного относительного пути:

```
var someModule = require('./somedir/someotherdir/somemodule.js');
```

А если вы уверены в том, что приложение никогда не переместится в другое место, используйте абсолютный путь. Он задается в формате файловой системы, а не в формате URL:

```
var someModule = require('/home/myname/public/modules/somemodule.js');
```

Если модуль установлен с использованием npm, указывать путь не нужно; достаточно указать имя модуля:

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

Node ищет модуль в подкаталоге node_modules, используя при этом иерархию поиска, включающую поиск в следующих подкаталогах:

1. Подкаталог node_modules, локальный для приложения (/home/myname/projects/node_modules).
2. Подкаталог node_modules в родительском подкаталоге текущего приложения (/home/myname/node_modules).
3. Далее вверх по родительским подкаталогам с проверкой node_modules, пока не будет достигнут каталог верхнего уровня, то есть корневой (/node_modules).
4. Поиск модуля завершается среди глобально установленных (см. далее).

Node использует эту иерархию, чтобы локализованные версии модуля проверялись до глобальных версий. Таким образом, если вы тестируете новую версию модуля и установили ее локально относительно своего приложения:

```
npm install somemodule
```

то она будет загружена первой вместо глобально установленного модуля:

```
npm install -g somemodule
```

Чтобы узнать, какой модуль был загружен, воспользуйтесь функцией `require.resolve()`:

```
console.log(require.resolve('async'));
```

Функция возвращает итоговое местонахождение модуля.

Если в качестве модуля указано имя папки, Node ищет файл package.json со свойством `main`, определяющим загружаемый файл модуля:

```
{ "name" : "somemodule",
  "main" : "./lib/somemodule.js" }
```

Если Node не может найти файл package.json, то для загрузки ищется файл index.js или index.node. Если все варианты завершаются неудачей, вы получаете ошибку.

КЭШИРОВАНИЕ ПО ИМЕНИ

Учтите, что при кэшировании используется имя и путь загрузки файла. Если вы кэшировали глобальную версию модуля, а затем загрузили локальную версию, то локальная версия также будет помещена в кэш.

Так как объект `Module` реализован на базе JavaScript, мы можем заглянуть в исходный код Node и более внимательно присмотреться к тому, что происходит «за кулисами».

Каждый модуль, представленный объектом `Module`, содержит функцию `require`, и глобальная версия `require`, которую мы используем, активизирует функцию конкретного модуля. В свою очередь функция `Module.require()` вызывает другую внутреннюю функцию `Module._load()`, которая выполняет всю только что описанную функциональность. Единственным исключением является запрос REPL, о котором речь пойдет в следующей главе; он использует собственную уникальную обработку.

Если модуль является *главным* модулем, то есть модулем, непосредственно вызванным из командной строки (то, что я называю приложением), он будет присвоен свойству `require.main` глобального объекта `require`. Введите следующую команду в файл с именем test.js и выполните его в Node:

```
console.log(require);
```

Вы увидите объект `main` (объект `Module`, заключающий код приложения) и сможете убедиться в том, что имя файла соответствует имени файла и пути

приложения. Также в выходных данных видны пути, используемые Node для поиска модулей, и кэш приложения, который в данном случае содержит только приложение.

ГДЕ НАЙТИ ИСХОДНЫЙ КОД NODE

Вы можете ознакомиться со всей функциональностью Node, загрузив исходный код. Вам не обязательно использовать его для построения Node в вашей системе; ознакомьтесь с ним в учебных целях. Функциональность JavaScript находится в подкаталоге /lib, а код Node на языке C++ находится в подкаталоге /src.

Все это заставляет меня снова вернуться к концепции глобального объекта Node. В главе 2 я рассмотрела объект Node `global` и описала, чем он отличается от глобального объекта браузера. В частности, в отличие от браузера, переменные верхнего уровня в нем ограничиваются своим непосредственным контекстом; это означает, что переменные, объявленные в модуле, не будут конфликтовать с переменными в приложении или любом другом модуле, включенном в приложение. Это происходит из-за того, что Node заключает все сценарии в следующую конструкцию:

```
function (module, exports, __filename, ...) {}
```

Иначе говоря, Node заключает модули (главные или любые другие) в анонимные функции, предоставляя доступ только к тому, что хочет открыть разработчик модуля. А поскольку при использовании модулей их свойствам предшествует имя модуля, эти свойства не конфликтуют с локально объявленными переменными.

И раз уж речь зашла о контексте, эта тема будет более подробно рассмотрена в следующем разделе.

Изоляция и модуль VM

Одно из первых правил, которые узнает разработчик JavaScript: что `eval()` следует избегать любой ценой. Дело в том, что `eval()` выполняет код JavaScript в одном контексте с остальным кодом приложения. Вы берете произвольный блок JavaScript и выполняете его на том же уровне доверия, что и тщательно написанный вами код. Это то же самое, что взять текст, введенный в текстовое поле, и включить его в запрос, не убедившись, что он не содержит вредоносных вставок.

Если по какой-то причине вам потребуется выполнить произвольный фрагмент JavaScript в вашем приложении Node, воспользуйтесь модулем `VM` для изоляции сценария. Впрочем, как указывают разработчики Node, этот метод не является стопроцентно надежным. Единственный надежный способ выполнения произвольного фрагмента JavaScript — выполнение в отдельном процессе. Впрочем, если вы хорошо понимаете исходный код JavaScript, но предпочитаете избегать непреднамеренных и случайных последствий, вы можете изолировать этот сценарий от своего локального окружения с помощью VM.

Сценарии могут предварительно компилироваться при помощи объекта `vm.Script` или же передаваться в составе функции, вызываемой непосредственно для `vm`. Также существует три типа функций. Функции первого типа — `script.runInNewContext()` или `vm.runInNewContext()` — выполняют сценарий в новом контексте, при этом локальные переменные или глобальный объект недоступны для сценария. Вместо этого функции передается новая изолированная среда («песочница», sandbox), обладающая собственным контекстом. Следующий код демонстрирует эту концепцию. Изолированная среда содержит два глобальных значения с такими же именами, как у двух глобальных объектов Node, но с переопределенным содержимым:

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

var sandbox = {
   process: 'this baby',
   require: 'that'
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

Происходит ошибка, потому что объект `console` не является частью контекста исполнительной среды сценария. Можно поступить так:

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

var sandbox = {
   process: 'this baby',
   require: 'that',
   console: console
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

Но такое решение противоречит цели создания совершенно нового контекста для сценария. Если вы хотите, чтобы сценарий получил доступ к глобальному объекту `console` (или другому объекту), используйте функцию

runInThisContext(). В листинге 3.1 я использую объект Script, чтобы продемонстрировать, что контекст включает в себя глобальный объект, но не локальные объекты.

Листинг 3.1. Выполнение сценария с доступом к глобальному объекту консоли

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

global.count1 = 100;
var count2 = 100;

var txt = 'if (count1 === undefined) var count1 = 0; count1++;' +
          'if (count2 === undefined) var count2 = 0; count2++;' +
          'console.log(count1); console.log(count2);';

var script = new vm.Script(txt);
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

Результат выполнения приложения выглядит так:

```
101
1
101
100
```

Переменная count1 объявлена для глобального объекта, и она доступна в контексте, в котором выполняется сценарий. Переменная count2 является локальной переменной, и она должна определяться в контексте. Изменения локальной переменной в изолированном сценарии не отражаются на одноименной локальной переменной в содержащем ее приложении.

Если я не объявлю count2 в сценарии, выполняемом в отдельном контексте, произойдет ошибка. Информация об ошибке выводится из-за того, что один из параметров контекстных функций изолированной среды — displayErrors — по умолчанию равен true. Другие параметры runInThisContext() — filename (показан в этом примере) и timeout (количество миллисекунд, по истечении которых сценарий завершается с выдачей ошибки).

Параметр filename задает имя файла, которое отображается в трассировке стека при выполнении сценария. Но если вы захотите задать имя файла для своего объекта Script, его следует передать при создании объекта Script, а не при вызове контекстной функции:

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

global.count1 = 100;
var count2 = 100;

var txt = 'count1++;' +
          'count2++;' +
          'console.log(count1); console.log(count2);';

var script = new vm.Script(txt, {filename: 'count.vm'});
-+


try {
  script.runInThisContext();
} catch(err) {
   console.log(err.stack);
}
```

Кроме различия `filename`, Script поддерживает еще два глобальных параметра в вызовах контекстных функций: `displayErrors` и `timeout`.

Выполнение кода приводит к выводу ошибки, потому что изолированный сценарий не может получить доступ к локальной переменной (`count2`) в приложении, хотя для него доступна глобальная переменная `count1`. Кроме того, при выводе ошибки выводится *трассировка стека*, в которой присутствует имя файла, переданное в параметре.

Вместо того чтобы записывать код прямо в приложении, его также можно загрузить из файла. Допустим, вы хотите выполнить следующий сценарий:

```
if (count1 === undefined) var count1 = 0; count1++;
if (count2 === undefined) var count2 = 0; count2++;
console.log(count1); console.log(count2);
```

Его можно заранее откомпилировать и выполнить в изолированной среде:

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

global.count1 = 100;
var count2 = 100;

var script = new vm.Script(fs.readFileSync('script.js','utf8'));
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

Что мешает использовать модуль файловой системы `File System` прямо в сценарии? Он присваивается локальной переменной, а следовательно, недоступен. Почему его нельзя импортировать в сценарий? Из-за недоступности `require`. Любые глобальные объекты или функции (в том числе и `require`) в сценарии доступны.

Последняя функция изолированной среды `runInContext()` тоже получает изолированную среду, но эта изолированная среда должна обладать контекстом (то есть контекст должен быть явно создан до вызова функции). В следующем коде эта функция вызывается непосредственно для объекта VM. Обратите внимание: в коде в изолированную среду с контекстом добавляется новая переменная, которая затем появляется в приложении:

```
var vm = require('vm');
var util = require('util');

var sandbox = {
     count1 : 1
    };

vm.createContext(sandbox);
if (vm.isContext(sandbox)) console.log('contextualized');

vm.runInContext('count1++; counter=true;',sandbox,
                {filename: 'context.vm'});

console.log(util.inspect(sandbox));
```

Результат выполнения приложения выглядит так:

```
contextualized
{ count1: 2, counter: true }
```

Функция `runInContext()` поддерживает три параметра, поддерживаемых `runInThisContext()` и `runInNewContext()`. И снова различие между выполнением функций в `Script` и непосредственно в VM заключается в том, что объект `Script` выполняет предварительное компилирование кода, а имя файла передается при создании объекта (а не в одном из параметров при вызове функции).

ВЫПОЛНЕНИЕ СЦЕНАРИЯ В ОТДЕЛЬНОМ ПРОЦЕССЕ

Если вы захотите еще лучше изолировать сценарий, выполняя его в отдельном процессе, ознакомьтесь с модулями независимых разработчиков, обеспечивающими дополнительную защиту. О том, где найти эти модули, рассказано в следующем разделе.

Знакомство с NPM

Значительная часть обширной функциональности, связанной с Node, реализована в модулях независимых разработчиков. Это модули маршрутизации, модули для работы с реляционными и документными СУБД, модули шаблонов, модули тестирования и даже модули платежных сервисов.

GITHUB

Хотя это и не является обязательным требованием, разработчикам рекомендуется отправлять свои модули в репозиторий GitHub.

Чтобы использовать модуль, вы можете загрузить его исходный код, а затем установить его вручную в своей среде. Многие модули содержат базовые инструкции по установке, или, как минимум, требования к установке можно вычислить просмотром файлов и каталогов, включенных в модуль. Однако существует куда более простой способ установки модулей Node — менеджер npm.

Сайт npm находится по адресу *http://npmjs.org/*. Документация доступна на сайте документации npm (*https://docs.npmjs.com/*).

Node поставляется с установленной копией npm, но эта версия npm не всегда является самой новой. Если вы захотите использовать другую версию npm, введите следующую команду для обновления имеющейся версии (используйте `sudo`, если этого требует ваша система):

```
npm install npm -g
```

Впрочем, будьте осторожны с установкой npm: если ваша версия будет отличаться от версии, установленной у ваших коллег, это может привести к самым неожиданным последствиям.

Чтобы получить подробную сводку команд npm, воспользуйтесь следующей командой:

```
$ npm help npm
```

Модули могут устанавливаться глобально или локально. Локальная установка лучше всего подходит для работы над изолированным проектом, когда всем остальным пользователям той же системы доступ к модулю не нужен. При

локальной установке (а этот вариант используется по умолчанию) модуль устанавливается в текущей позиции каталога node_modules.

```
$ npm install имя_модуля
```

Например, для установки Request используется следующая команда:

```
$ npm install request
```

Программа npm не только устанавливает модуль `Request`, но и находит модули, от которых он зависит, и устанавливает их. Чем сложнее модуль, тем больше зависимостей устанавливается. На рис. 3.1 показан неполный отчет об установке Request на моем компьютере с Windows (для Node 5.0.0).

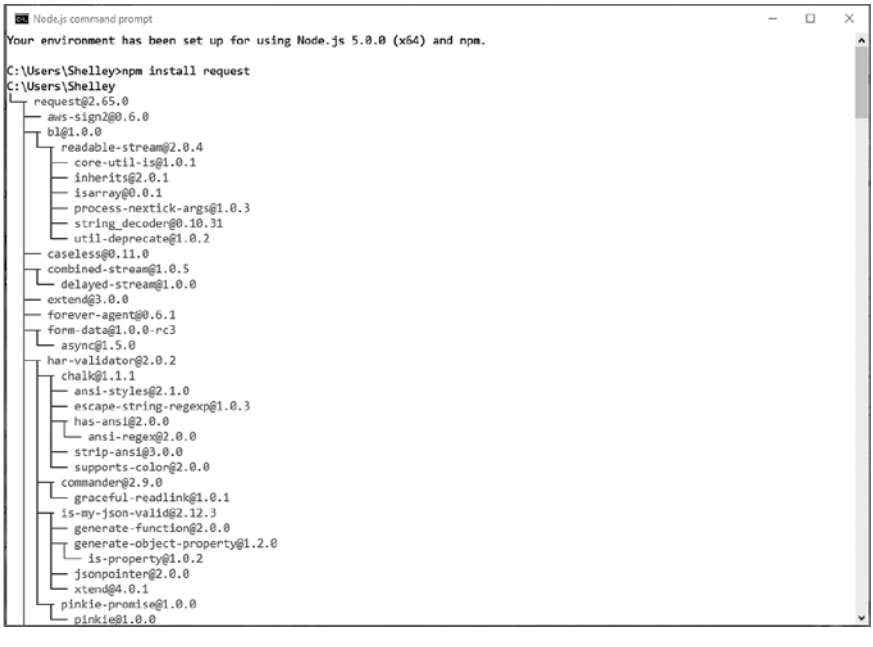


Рис. 3.1. Часть списка загруженных зависимостей для модуля Request

После того как модуль будет установлен, он будет находиться в каталоге node_modules вашего локального каталога. Все зависимости также будут установлены в каталог node_modules этого модуля. С первого взгляда видно, что при установке такого модуля, как `Request`, выполняется немалая работа.

Чтобы установить пакет глобально, используйте параметр `-g` или `--global`:

```
$ npm install request -g
```

Если вы работаете в системе Linux, не забудьте использовать `sudo` для глобальной установки модуля:

```
$ sudo npm install request -g
```

Некоторые модули, а также приложения командной строки могут потребовать глобальной установки. В следующих примерах устанавливаются пакеты, зарегистрированные на сайте npm. Вы также можете установить модуль, находящийся в папке файловой системы, или же *tar*-архив — локальный или загружаемый по URL:

```
npm install http://somecompany.com/somemodule.tgz
```

Если пакет существует в нескольких версиях, есть возможность установить конкретную версию:

```
npm install modulename@0.1
```

Вы даже можете установить хорошо знакомый вам модуль `jQuery`:

```
npm install jquery
```

Если модуль больше не используется, его можно удалить:

```
npm uninstall имя_модуля
```

Следующая команда приказывает npm проверить наличие новых модулей и обновить найденные:

```
npm update
```

Также можно обновить один модуль:

```
npm update имя_модуля
```

Чтобы обновить саму программу npm, выполните следующую команду:

```
npm install npm -g
```

А если вы хотите узнать, есть ли среди пакетов устаревшие, воспользуйтесь командой

```
npm outdated
```

Эта команда также может быть выполнена для отдельного модуля.

Для вывода списка всех установленных пакетов и зависимостей передайте параметр `list`, `ls`, `la` или `ll`:

```
npm ls
```

С параметрами `la` и `ll` команды выводят расширенные описания. Например, в число зависимостей Request входит tunnel-agent@0.4.1 (версия 0.4.1 пакета tunnel-agent). Что это вообще такое — tunnel-agent? Ввод запроса `npm la` в командной строке выводит список всех зависимостей, включая tunnel-agent, с дополнительной информацией:

```
tunnel-agent@0.4.1
HTTP proxy tunneling agent. Formerly part of mikeal/request,
now a standalone module
git+https://github.com/mikeal/tunnel-agent.git
https://github.com/mikeal/tunnel-agent#readme
```

Иногда в выводе встречаются предупреждения — допустим, неразрешенная зависимость или необходимость установки более старой версии модуля. Чтобы исправить ситуацию, установите модуль или необходимую версию модуля:

```
npm install jsdom@0.2.0
```

Флаг `-d` устанавливает все зависимости напрямую. Например, в каталоге модуля введите команду

```
npm install -d
```

Если вы хотите установить версию модуля, которая еще не была загружена в реестр npm, установите ее непосредственно из репозитория Git:

```
npm install https://github.com/visionmedia/express/tarball/master
```

Однако будьте осторожны: как выяснилось, если установить еще не выпущенную версию модуля и провести обновление npm, версия из реестра npm может заменить используемую версию.

ИСПОЛЬЗОВАНИЕ NPM В LINUX ЧЕРЕЗ PUTTY

Если вы работаете с Node/npm в Linux, используя приложение PuTTY в Windows, вы заметите, что при использовании команд npm вместо четко определенных строк с зависимостями выводятся некие странные символы.

Чтобы решить эту проблему, прикажите PuTTY транслировать символы с использованием кодировки UTF-8. В PuTTY выберите команду Window>Translation и затем UTF-8 в раскрывающемся списке.

Чтобы узнать, какие модули установлены глобально, используйте следующую команду:

```
npm ls -g
```

Для получения дополнительной информации об установке npm используйте команду `config`. Следующая команда выводит список параметров конфигурации npm:

```
npm config list
```

Более подробное описание всех параметров конфигурации выводится командой

```
npm config ls -l
```

Изменение или удаление параметров конфигурации из командной строки осуществляется либо командой

```
npm config delete ключ
npm config set ключ значение
```

либо прямым редактированием конфигурационного файла:

```
$ npm config edit
```

Если у вас нет полной уверенности в том, к каким последствиям приведет изменение, параметры конфигурации лучше не трогать.

Также можно провести поиск модуля по условиям, которые, как вы думаете, могут вернуть лучший результат:

```
npm search html5 parser
```

При первом проведении поиска npm строит индекс, на что может потребоваться несколько минут. Впрочем, после завершения индексирования вы получаете список возможных модулей, соответствующих заданному условию (или условиям).

ОШИБКА РАЗБОРА JSON

Если при использовании npm вы получаете ошибку «registry error parsing json», попробуйте воспользоваться одним из зеркальных серверов npm для завершения задачи. Например, для европейского зеркала используется следующий адрес:

npm --registry http://registry.npmjs.eu/ search html5 parser

Сайт npm предоставляет реестр модулей с возможностью просмотра и регулярно обновляемый список модулей, чаще всего используемых другими модулями или приложениями Node. В следующем разделе я приведу подборку таких модулей.

И последнее замечание по поводу npm перед тем, как двигаться дальше. В самом начале ваших экспериментов с npm в конце вывода могут появляться предупреждения. В первой строке сообщается о том, что файл package.json не найден, а в остальных содержатся всевозможные предупреждения, связанные с отсутствующим файлом package.json.

Документация npm рекомендует создать файл package.json для сопровождения локальных зависимостей. В принципе, это не обязательно, но предупреждения слегка раздражают.

Чтобы создать файл package.json по умолчанию в каталоге проекта, выполните следующую команду:

```
npm init --yes
```

Команда создает в каталоге файл package.json по умолчанию; при этом вам будет предложено ответить на ряд вопросов по проекту: ваше имя, название проекта и т. д., причем у каждого вопроса имеется значение по умолчанию. В дальнейшем при установке модуля вы уже не будете получать часть раздражающих сообщений. Чтобы избавиться от остальных, необходимо обновить данные JSON в этом файле и включить в них описание и информацию о репозитории. Кроме того, если вы хотите, чтобы файл обновлялся при установке обновленного модуля, используйте следующий синтаксис:

```
npm install request --save-dev
```

Имя и версия модуля сохраняются в поле `devDependencies` файла package.json. Также можно сохранить модуль в зависимостях модулей, но я расскажу об этом подробнее в файле package.json (см. «Создание и публикация собственных модулей Node», с. 92), когда вы займетесь созданием собственных модулей Node.

Для автоматического сохранения зависимостей следует добавить и (или) отредактировать файл npmrc. Этот файл может быть добавлен на уровне пользователя (~/.npmrc), на уровне проекта (/path/project/.npmrc) или глобально ($PREFIX/etc/npmrc), а также с использованием встроенного конфигурационного файла (/path/to/npm/npmrc). Чтобы включить автоматическое сохранение зависимостей в личных настройках, выполните следующие команды:

```
npm config set save=true
npm config set save-exact=true
```

Эти настройки автоматически добавляют флаг `--save` (для сохранения пакета в зависимостях) и `--save-exact` (сохранение с точной версией, а не с диапазоном по умолчанию семантического управления версиями npm) при установке новых пакетов.

Также есть много других параметров конфигурации, которые вы можете настраивать для своих целей. Знакомство с ними хорошо начать с документации npm (*https://docs.npmjs.com/misc/config*).

Создание и публикация собственных модулей Node

Как и в случае с кодом JavaScript на стороне клиента, многократно используемый код JavaScript стоит выделить в собственные библиотеки. Единственное отличие заключается в том, что вам придется выполнить пару лишних действий для преобразования библиотеки JavaScript в модуль, рассчитанный на взаимодействие с Node.

Создание модуля

Допустим, у вас имеется библиотечная функция JavaScript `concatArray`, которая получает строку и массив строк, выполняет конкатенацию первой строки с каждой строкой в массиве и возвращает новый массив:

```
function concatArray(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
}
```

Вы хотите использовать эту функцию наряду с другими в своих приложениях Node. Чтобы преобразовать библиотеку JavaScript для использования в Node, необходимо экспортировать все функции с помощью объекта `exports`, как показано в следующем примере:

```
exports.concatArray = function(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
};
```

Чтобы использовать concatArray в приложении Node, импортируйте библиотеку. После этого вы сможете использовать экспортированную функцию в своем приложении:

```
var newArray = require ('./arrayfunctions.js');
console.log(newArray.concatArray('hello', ['test1','test2']));
```

Вы также можете создать модуль, состоящий из конструктора или функции, и экспортировать его с использованием module.exports.

Например, модуль Mime, от которого могут зависеть многие другие модули, создает функцию Mime():

```
function Mime() { ... }
```

добавляет функциональность с использованием свойства prototype:

```
Mime.prototype.define = function(map) {...}
```

создает экземпляр по умолчанию:

```
var mime = new Mime();
```

присваивает функцию Mime одноименному свойству:

```
mime.Mime = Mime;
```

и после этого экспортирует экземпляр:

```
module.exports=mime;
```

После этого вы сможете использовать различные функции Mime в своем приложении:

```
var mime = require('mime');
console.log(mime.lookup('phoenix5a.png')); // image/png
```

Упаковка целого каталога

Вы можете разбить свой модуль на несколько файлов JavaScript, находящихся в каталоге. Node может загружать содержимое каталогов при условии, что его содержимое организовано одним из двух способов.

Первый способ основан на создании файла package.json с информацией о каталоге. Структура содержит разнообразную информацию, но к упаковке модуля относятся два свойства — name и main:

```
{ "name" : "mylibrary",
  "main" : "./mymodule/mylibrary.js"
}
```

Первое свойство, `name`, содержит имя модуля, а второе свойство, `main`, обозначает точку входа модуля.

Во втором способе загрузки содержимого в каталог добавляется файл index.js или index.node, служащий главной точкой входа модуля.

Зачем использовать каталог вместо одного модуля? Чаще всего это делается из-за того, что вы используете существующие библиотеки JavaScript и предоставляете файл-«обертку», который «упаковывает» экспортируемые функции командой `exports`. А может быть, ваша библиотека настолько велика, что вы хотите разбить ее для удобства внесения изменений.

В любом случае следует помнить, что все экспортируемые объекты должны находиться в одном главном файле, загружаемом Node.

Подготовка модуля к публикации

Если вы захотите сделать свой пакет доступным для других, вы можете рекламировать его на своем сайте, но при этом вы упустите значительную аудиторию. Готовые модули лучше публиковать в реестре npm.

Ранее я упоминала о файле package.json. Документацию по JSON для npm можно найти в Интернете (*https://docs.npmjs.com/files/package.json*). Она базируется на рекомендациях системы модулей CommonJS.

В пакет package.json рекомендуется включать следующие поля:

- `name` — имя пакета (обязательно).
- `description` — описание пакета.
- `version` — текущая версия, соответствующая требованиям семантической версии (обязательно).
- `keywords` — массив условий поиска.
- `maintainers` — массив ответственных за сопровождение пакета (с именами, адресами электронной почты и сайтами).
- `contributors` — массив соавторов пакета (с именами, адресами электронной почты и сайтами).

- `bugs` — URL-адрес для отправки ошибок.
- `licenses` — массив лицензий.
- `repository` — репозиторий пакетов.
- `dependencies` — необходимые пакеты и их номера версий.

Обязательны только поля `name` и `version`, хотя включить рекомендуется все поля. К счастью, npm упрощает создание этого файла. Если ввести в командной строке следующую команду:

```
npm init
```

программа переберет все обязательные/рекомендованные поля, предложив вам ввести значение каждого. Когда все будет сделано, она сгенерирует файл package.json.

В главе 2 (листинг 2.7) я создала объект с именем `InputChecker`, который ищет команды во входных данных и обрабатывает эти команды. Пример показывал, как включать функциональность `EventEmitter` в программу. Теперь мы изменим этот простой объект, чтобы его можно было использовать в других приложениях и модулях.

Сначала мы создадим в node_modules подкаталог с именем inputcheck и перенесем в него существующий код `InputChecker`. Файлу необходимо присвоить имя index.js. Затем в код вносятся изменения и из него извлекается часть, реализующая новый объект. Мы сохраним ее для будущего тестового файла. В последнем изменении добавляется объект `exports`; итоговый код приведен в листинге 3.2.

Листинг 3.2. Приложение из листинга 2.7, преобразованное в модуль

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
};
```

```
util.inherits(InputChecker,eventEmitter);
InputChecker.prototype.check = function (input) {
  var command = input.toString().trim().substr(0,3);
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));
  } else if (command == 'en:') {
     this.emit('end');
  } else {
     this.emit('echo',input);
  }
};
```

Экспортировать функцию объекта напрямую не удастся, потому что `util.inherits` ожидает, что в файле с именем `InputChecker` существует объект. Прототип объекта `InputChecker` изменяется позднее в файле, также можно было изменить ссылки в коде и использовать `exports.InputChecker`, но это неуклюжее решение. С таким же успехом объект можно присвоить в отдельной команде.

Чтобы создать файл package.json, я выполнила команду `npm init` и ответила на все вопросы. Полученный файл показан в листинге 3.3.

Листинг 3.3. Сгенерированный файл package.json для модуля inputChecker

```
{
  "name": "inputcheck",
  "version": "1.0.0",
  "description": "Looks for and implements commands from input",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "command",
    "check"
  ],
  "author": "Shelley Powers",
  "license": "ISC"
}
```

Команда `npm init` не запрашивает информацию о зависимостях, поэтому их придется добавить прямо в файл. Впрочем, есть другое, более правильное решение — использовать команду `npm install --save` при установке модулей, чтобы зависимости были добавлены автоматически. Однако модуль `InputChecker` не зависит от внешних модулей, поэтому эти поля можно оставить пустыми.

На этой стадии мы можем протестировать новый модуль и убедиться в том, что он действительно работает как модуль. В листинге 3.4 приведен фрагмент

предыдущей версии приложения InputChecker, которая тестирует новый объект, теперь выделенный в отдельное тестовое приложение.

Листинг 3.4. Тестовое приложение InputChecker

```
var inputChecker = require('inputcheck').InputChecker;

// Тестирование нового объекта и обработка событий
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

Теперь тестовое приложение можно переместить в новый подкаталог test в каталоге модуля, чтобы оно было упаковано в модуле как пример. По общепринятым правилам следует предоставить каталог test с одним или несколькими тестовыми приложениями, а также каталог doc с документацией. Для такого маленького модуля файла README должно быть достаточно.

Так как у нас теперь имеется тестовое приложение, необходимо внести изменения в файл package.json и добавить в него ссылку:

```
 "scripts": {
    "test": "node test/test.js"
  },
```

Это довольно примитивный тест, который не использует мощные тестовые возможности Node, но и он показывает, как может работать тестирование. Чтобы запустить тестовое приложение, введите в командной строке следующую команду:

```
npm test
```

Остается создать сжатый tar-архив модуля. Впрочем, если вы публикуете модуль так, как описано в разделе «Публикация модуля» (см. ниже), этот шаг можно пропустить.

ЧТО ТАКОЕ TAR-АРХИВ?

Термин «tar-архив» уже встречался в этой и предыдущих главах. Если вы не работали в Unix, этот термин может быть вам незнаком. Tar-архив представляет собой один или несколько файлов, объединенных командой tar.

Когда все будет сделано, модуль можно опубликовать.

НЕ ТОЛЬКО МОДУЛЬ

Как минимум можно обойтись упрощенным модулем (таким, как inputcheck), но для публикации модуля в Интернете необходимо предоставить другие сведения. Вы должны предоставить репозиторий для своего модуля, URL-адрес для отправки сообщений об ошибках, домашнюю страницу модулей и т. д. Начните с малого и двигайтесь вверх.

Публикация модуля

Создатели `npm` также предоставили нам руководство с подробным описанием того, что необходимо сделать для публикации модуля Node: Developer Guide (*https://docs.npmjs.com/misc/developers*).

В документации приведены дополнительные требования к файлу `package.json`. В дополнение к уже созданным полям также необходимо добавить поле `directories` с хешем папок, включая уже упоминавшиеся `test` и `doc`:

```
"directories" : {
    "doc" : ".",
    "test" : "test",
    "example" : "examples"
}
```

Перед публикацией в руководстве рекомендуется протестировать модуль на корректность установки. Чтобы выполнить тестирование, введите следующую команду в корневом каталоге модуля:

```
npm install . -g
```

К этому моменту мы протестировали модуль `inputChecker`, изменили файл package.json, добавили в него каталоги и убедились в том, что пакет успешно устанавливается.

Затем необходимо добавить себя как пользователей npm, если это не было сделано ранее. Введите следующую команду:

```
npm adduser
```

после чего по отображаемым подсказкам введите имя пользователя, пароль и адрес электронной почты.

Осталось сделать последний шаг:

```
npm publish
```

Можно указать путь к tar-архиву или каталогу. Как предупреждает Guide, доступ предоставляется ко всему содержимому каталога, если только вы не используете в файле package.json директиву `.npmignore` со списком файлов для игнорирования материала. Тем не менее перед публикацией модуля лучше просто удалить все, без чего можно обойтись.

После публикации — и после того, как исходный код также будет загружен в GitHub (если вы используете этот репозиторий), — модуль официально готов для использования другими разработчиками. Рекламируйте свой модуль в Твиттере, Google+, Facebook, на своем сайте и вообще повсюду, где люди смогут о нем узнать. Такая реклама не хвастовство, а распространение информации.

Изучение модулей Node и три важнейших модуля

Найти модули Node несложно. Большинство модулей вы найдете по рекомендациям друзей и коллег. Другие могут обнаружиться, когда вы ищете нужную функциональность в поисковой системе где-то в первых позициях списка результатов.

Когда-то модулей было не так много и лучшие представители разных типов помещались на одной странице. Эти времена давно прошли. В наши дни, если вам потребуется найти модуль, скорее всего, вы будете делать это на сайте npm (*https://www.npmjs.com/*). Здесь вы не только сможете найти документацию по npm, но и получите доступ к каталогу модулей Node с поддержкой поиска.

Скажем, если вы ищете модули для поддержки протокола OAuth, введите это название в строке поиска в начале страницы. Первый результат — `OAuth` — является самым популярным модулем поддержки OAuth среди используемых в наши дни. Это подтверждает статистика использования модуля.

Обязательно проверяйте статистику модуля. Самая большая проблема при поиске модулей Node на сайте npm заключается в том, что многие модули не поддерживаются или постепенно забываются. Определить, какие модули активно и широко поддерживаются, можно только одним способом — по статистике в правой части страницы. На рис. 3.2 показан пример статистики для модуля `OAuth`.

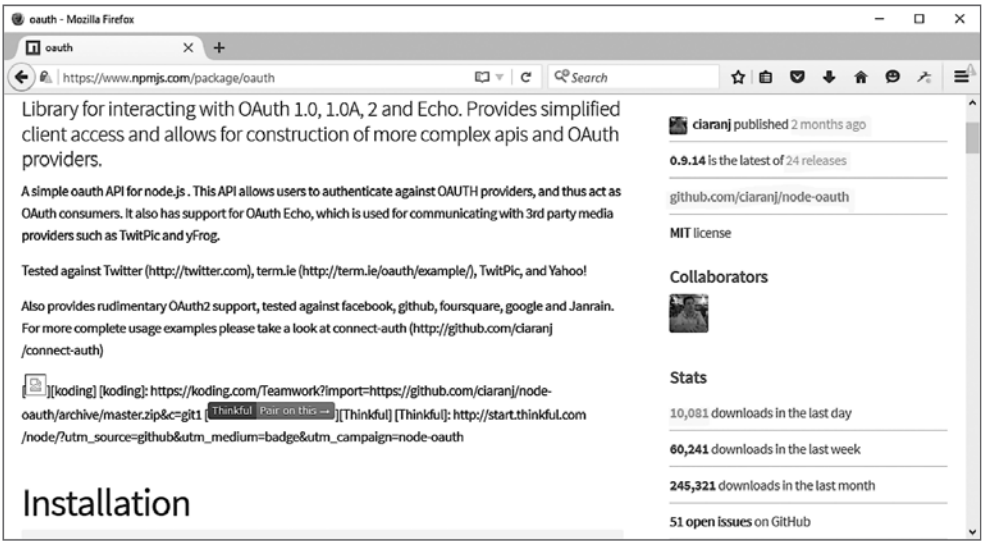


Рис. 3.2. Статистика NPM для модуля OAuth

Обратите внимание на выделенный текст. Текущая версия модуля была опубликована пару месяцев назад; это показывает, что модуль обновляется. Так как это последняя версия из 24 выпусков, это показывает, что модуль находится в активной разработке. Впрочем, самым важным показателем можно считать количество загрузок: более 10 000 всего за один день. Модуль очень активно используется, а это означает, что он жив и, скорее всего, сопровождается своими разработчиками.

Кроме того, вы можете обратиться к странице GitHub и проверить список текущих проблем, чтобы понять, насколько активно разработчики реагируют

на сообщения об ошибках и нет ли каких-либо проблем, из-за которых с этим модулем лучше не связываться (например, проблем безопасности).

Бесспорно, модуль `OAuth` является и наиболее активно поддерживаемым, и лучшим в своей категории модулем. Это не значит, что не существует других хороших вариантов, но, пока вы не освоитесь с Node, лучше придерживаться наиболее часто используемых модулей Node.

Раз уж речь зашла о часто используемых модулях, на сайте npm (на самой первой странице) также перечислены самые популярные модули Node на сегодняшний день. Кроме того, имеется список модулей с самыми высокими оценками (*https://www.npmjs.com/browse/star*) и модулей, чаще всего используемых в зависимостях (*https://www.npmjs.com/browse/depended*). Последнюю группу составляют модули, которые устанавливаются как зависимости других модулей. В следующих пяти разделах мы рассмотрим пять модулей, входящих в оба списка: Async, Commander и Underscore. Также есть несколько других модулей, часто используемых в книге.

Управление обратными вызовами с использованием Async

Приложение в листинге 2.11 (глава 2) использует асинхронный паттерн, при котором каждая функция вызывается по очереди и передает свои результаты следующей функции, причем вся цепочка останавливается только при возникновении ошибок. Есть несколько подобных паттернов, хотя одни из них являются разновидностями других и не во всех случаях применяется в точности одинаковая терминология.

Пример также демонстрировал главную проблему при использовании функций обратного вызова: эффект «пирамиды погибели» с последовательным смещением вложенных обратных вызовов вправо.

Один из самых популярных модулей для обхода этого эффекта, Async, заменяет типичный паттерн обратного вызова более линейными и лучше управляемыми паттернами. Некоторые асинхронные паттерны, поддерживаемые Async:

- `waterfall` — функции вызываются поочередно, результаты всех функций передаются в виде массива последней функции обратного вызова.
- `series` — функции вызываются поочередно с необязательной передачей результатов всех функций в виде массива последней функции обратного вызова.

- `parallel` — функции выполняются параллельно, при их завершении результаты передаются последней функции обратного вызова (хотя массив результатов не является частью паттерна в некоторых интерпретациях этого паттерна).
- `whilst` — многократный вызов одной функции с активизацией последнего обратного вызова только в том случае, если предварительное условие равно `false` или произошла ошибка.
- `queue` — функции выполняются параллельно до заданного уровня лимита. Новые функции ставятся в очередь и ожидают завершения одной из выполняемых функций.
- `until` — многократный вызов одной функции с активизацией последнего обратного вызова только в том случае, если условие после окончания выполнения равно `false` или произошла ошибка.
- `auto` — функции вызываются по мере необходимости, каждая функция получает результаты предыдущих обратных вызовов.
- `iterator` — каждая функция вызывает следующую, с возможностью индивидуального обращения к следующему итератору.
- `apply` — функция продолжения с применением аргументов; часто используется в сочетании с другими функциями управления последовательностью выполнения.
- `nextTick` — вызывает функцию обратного вызова в следующей итерации цикла событий (использует `process.nextTick` в Node).

Модуль `Async` также предоставляет функциональность для управления коллекциями: собственные разновидности `forEach`, `map` и `filter`, а также другие вспомогательные функции, включая функции мемоизации. Тем не менее сейчас нас интересуют средства управления потоком выполнения.

У Async имеется свой репозиторий на сайте GitHub.

Установите `Async` при помощи npm. Чтобы установить Async глобально, используйте флаг `-g`. Если вы хотите обновить зависимости, используйте параметр `--save` или `--save-dev`:

```
npm install async
```

Как упоминалось ранее, Async предоставляет средства управления потоком выполнения для различных асинхронных паттернов, включая `serial`, `parallel` и `waterfall`. Например, структура примера в главе 2 аналогична паттерну Async `waterfall`, поэтому мы воспользуемся методом `async.waterfall`. В листинге 3.5 метод `async.waterfall` открывает и читает файл данных методом `fs.readFile`, выполняет синхронную замену строки, а затем записывает строку обратно в файл методом `fs.writeFile`. Обратите особое внимание на функцию обратного вызова, используемую на каждом шаге приложения.

Листинг 3.5. Использование async.waterfall для асинхронного чтения, модификации и записи содержимого файла

```
var fs = require('fs'),
    async = require('async');

async.waterfall([
   function readData(callback) {
      fs.readFile('./data/data1.txt', 'utf8', function(err, data){
           callback(err,data);
       });
   },
   function modify(text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, adjdata);
   },
   function writeData(text, callback) {
       fs.writeFile('./data/data1.txt', text, function(err) {
          callback(err,text);
       });
   }
], function (err, result) {
      if (err) {
        console.error(err.message);
      } else {
        console.log(result);
      }
});
```

Метод `async.waterfall` получает два параметра: массив задач и необязательную завершающую функцию обратного вызова. Каждая функция асинхронной задачи является элементом массива `async.waterfall`, и у каждой функции список параметров завершается функцией обратного вызова. Именно эта функция обратного вызова позволяет объединять результаты асинхронных обратных вызовов в цепочку без физического вложения функций. Но как видно из кода, функция обратного вызова каждой функции обрабатывается

так же, как она обрабатывалась бы обычно в схеме с вложением, — не считая того, что нам не нужно проверять ошибки в каждой функции. Async ожидает найти объект ошибки в первом параметре каждой функции обратного вызова. Если при обратном вызове передается объект ошибки, то процесс в этой точке завершается и вызывается завершающая функция обратного вызова. В этой завершающей функции обрабатывается ошибка или окончательный результат.

В листинге 3.5 используются именованные функции, тогда как в документации Async приведены анонимные функции. Именованные функции упрощают отладку и обработку ошибок, но возможны оба варианта.

Обработка очень похожа на то, что было в главе 2, но без вложения (и необходимости проверять ошибки в каждой функции). Такое решение может показаться более сложным, и я не буду рекомендовать его для простого вложения, но посмотрите, что оно может сделать с более сложным вложением обратных вызовов. Листинг 3.6 повторяет функциональность из главы 2, но без вложения обратных вызовов и избыточных отступов.

Листинг 3.6. Чтение объектов из каталога, тестирование поиска файлов, чтения файла и записи результатов обратно в файл

```
var fs = require('fs'),
    async = require('async'),
    _dir = './data/';

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

async.waterfall([
   function readDir(callback) {
      fs.readdir(_dir, function(err, files) {
         callback(err,files);
      });
   },
   function loopFiles(files, callback) {
      files.forEach(function (name) {
         callback (null, name);
      });
   },

   function checkFile(file, callback) {
      fs.stat(_dir + file, function(err, stats) {
```

```
         callback(err, stats, file);
      });
   },
   function readData(stats, file, callback) {
      if (stats.isFile())
         fs.readFile(_dir + file, 'utf8', function(err, data){
           callback(err,file,data);
       });
   },
   function modify(file, text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, file, adjdata);
   },
   function writeData(file, text, callback) {
       fs.writeFile(_dir + file, text, function(err) {
          callback(err,file);
       });
   },
   function logChange(file, callback) {
       writeStream.write('changed ' + file + '\n', 'utf8',
                         function(err) {
          callback(err);
      });
   }
], function (err) {
         if (err) {
            console.error(err.message);
         } else {
            console.log('modified files');
         }
});
```

Здесь присутствуют все аспекты функциональности из главы 2. Метод `fs.readdir` используется для получения массива объектов каталога. Метод Node `forEach` (не `forEach` из Async!) используется для обращения к каждому отдельному объекту. Метод `fs.stats` используется для получения объекта `stats` для каждого объекта; объект `stats` используется для обнаружения файлов среди объектов каталога. Обнаруженный файл открывается, и программа обращается к его данным. Затем данные модифицируются и записываются обратно в файл вызовом `fs.writeFile`. Операция регистрируется в журнальном файле, а ее успешное завершение дублируется на консоли.

Обратите внимание на передачу дополнительных данных при некоторых обратных вызовах. Большинству функций должно передаваться имя файла и текст; они передаются в нескольких последних методах. Методам может передаваться любое количество данных при условии, что в первом параметре

передается объект ошибки (или `null`, если объекта ошибки нет), а в последнем — функция обратного вызова. Нам не нужно проверять ошибку в каждой функции асинхронной задачи, потому что Async проверяет объект ошибки в каждом обратном вызове и при обнаружении ошибки останавливает обработку и вызывает завершающую функцию обратного вызова.

Другие методы управления потоком выполнения Async, такие как `async.parallel` и `async.serial`, работают аналогично: в первом параметре метода передается массив задач, а во втором — необязательная завершающая функция обратного вызова. Однако способы обработки асинхронных задач различаются, как и следовало ожидать.

Метод `async.parallel` вызывает все асинхронные функции одновременно, а когда они все завершатся, вызывает необязательную завершающую функцию обратного вызова. Листинг 3.7 использует `async.parallel` для параллельного чтения содержимого трех файлов. Однако вместо массива функций в этом примере используется альтернативный механизм, поддерживаемый Async: передача объекта, в котором каждое свойство представляет асинхронную задачу. Затем результаты выводятся на консоль после завершения всех трех задач.

Листинг 3.7. Параллельное открытие трех файлов и чтение их содержимого

```
var fs = require('fs'),
    async = require('async');

async.parallel({
   data1 : function (callback) {
      fs.readFile('./data/fruit1.txt', 'utf8', function(err, data){
           callback(err,data);
       });
   },
   data2 : function (callback) {
      fs.readFile('./data/fruit2.txt', 'utf8', function(err, data){
           callback(err,data);
       });
   },
   data3 : function readData3(callback) {
      fs.readFile('./data/fruit3.txt', 'utf8', function(err, data){
           callback(err,data);
       });
   },

}, function (err, result) {
      if (err) {
         console.log(err.message);
      } else {
```

```
        console.log(result);
    }
});
```

Результаты возвращаются в виде массива объектов, в котором каждый результат представлен одним из свойств. Если три файла данных в примере содержат следующие данные:

- fruit1.txt: apples
- fruit2.txt: oranges
- fruit3.txt: peaches

результат выполнения листинга 3.7 выглядит так:

```
{ data1: 'apples\n', data2: 'oranges\n', data3: 'peaches\n' }
```

Тестирование других методов управления потоком выполнения Async остается читателю для самостоятельной работы. Помните, что при работе с методами Async от вас требуется совсем немного: передать функцию обратного вызова каждой асинхронной задаче и вызвать эту функцию после завершения с передачей объекта ошибки (или `null`) и других необходимых данных.

Commander и волшебство командной строки

Модуль `Commander` предоставляет поддержку параметров командной строки — флагов, передаваемых при выполнении приложения (например, вездесущего флага `-h` или `--help` для получения информации об использовании утилиты или приложения).

Установите модуль при помощи npm:

```
npm install commander
```

Включите его в программу директивой `require`:

```
var program = require('commander');
```

Чтобы использовать модуль, создайте цепочку из вызовов `option` для всех параметров, поддерживаемых приложением. В следующем коде приложение поддерживает два параметра по умолчанию: `-V` или `--version` для версии и `-h` или `--help` для получения, а также два специализированных параметра: `-s` или `--source` для определения сайта-источника и `-f` или `--file` для имени файла:

```
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [web site]', 'Source web site')
   .option ('-f, --file [file name]', 'File name')
   .parse(process.argv);

console.log(program.source);
console.log(program.file);
```

Мы предоставляем нестандартные параметры, а `Commander` предоставляет параметры по умолчанию для вывода версии и справки. Если запустить приложение командой

```
node options -h
```

`Commander` выведет сводку поддерживаемых параметров:

```
Usage: options [options]
Options:
  -h, --help               output usage information
  -V, --version            output the version number
  -s, --source [web site]  Source web site
  -f, --file [file name]   File name
```

Короткие параметры можно объединять (например, `-sf`); `Commander` правильно обработает их. Также поддерживаются конструкции вида `--file-name`, а в результатах используется «верблюжий регистр» (camel-casing): `program.fileName`.

`Commander` также поддерживает преобразование типов (type casting):

```
.option('-i, --integer <n>', 'An integer argument', parseInt)
```

регулярные выражения:

```
.option('-d --drink [drink]', 'Drink', /^(coke|pepsi|izze)$/i)
```

и возможность передачи произвольного количества аргументов в последнем параметре, например, если ваше приложение может поддерживать несколько ключевых слов, количество которых заранее неизвестно. Параметр `command` в таких случаях создается следующим образом:

```
var program = require('commander');

program
```

```
  .version('0.0.1')
  .command('keyword <keywork> [otherKeywords...]')
  .action(function (keyword, otherKeywords) {
    console.log('keyword %s', keyword);
    if (otherKeywords) {
      otherKeywords.forEach(function (oKey) {
        console.log('keyword %s', oKey);
      });
    }
  });

program.parse(process.argv);
```

Командная строка

```
node options2 keyword one two three
```

приводит к следующему результату:

```
keyword one
keyword two
keyword three
```

ГДЕ ЗАГРУЗИТЬ COMMANDER

Если вы захотите загрузить `Commander` напрямую, это можно сделать в репозитории GitHub (https://github.com/tj/commander.js). Модуль `Commander` особенно полезен для приложений командной строки, как показано в главе 6.

Модуль Underscore

Установите модуль `Underscore` при помощи npm:

```
npm install underscore
```

По словам разработчиков, `Underscore` — своего рода «пояс с инструментами» для Node. Этот модуль предоставляет расширенную функциональность JavaScript, для которой обычно использовались сторонние библиотеки, такие как jQuery или Prototype.js.

Модуль `Underscore` («подчеркивание») назван так из-за того, что его функциональность обычно вызывается с символом подчеркивания (_), по аналогии с символом `$` в jQuery. Пример:

```
var _ = require('underscore');
_.each(['apple','cherry'], function (fruit) { console.log(fruit); });
```

Конечно, проблема с символом подчеркивания заключается в том, что этот символ имеет конкретный смысл в REPL, как вы увидите в следующей главе. Впрочем, не беспокойтесь — вместо него можно использовать другую переменную — `us`:

```
var us = require('underscore');
us.each(['apple','cherry'], function(fruit) { console.log(fruit); });
```

Модуль `Underscore` предоставляет расширенную функциональность для работы с массивами, коллекциями, функциями, объектами, а также ряд общих функций. К счастью, по всей функциональности на сайте имеется превосходная документация (*http://underscorejs.org/*), поэтому я не стану приводить подробное описание. Тем не менее я хочу упомянуть одну интересную возможность: механизм расширения `Underscore` собственными служебными функциями при помощи функции `mixin`. Вы можете быстро опробовать этот метод (и другие методы) в сеансе REPL:

```
> var us = require('underscore');
> us.mixin({
... betterWithNode: function(str) {
..... return str + ' is better with Node';
..... }
... });
> console.log(us.betterWithNode('chocolate'));
chocolate is better with Node
```

Термин «примесь» (mixin) встречается в нескольких модулях Node. Он происходит от паттерна, в котором свойства одного объекта добавляются («примешиваются») к другому объекту.

`Underscore` не единственный вспомогательный модуль с высоким рейтингом. Другой пример — `lodash`. Если вы еще не работали ни с одним из этих модулей, я порекомендовала бы рассмотреть оба. На сайте `lodash` (*https://lodash.com/docs*) имеется полная документация по этому полезному модулю.

4 Интерактивная работа с REPL и подробнее о Console

Пока вы изучаете Node и пытаетесь разобраться в коде своего модуля или приложения, вам не придется сохранять код JavaScript в файле и запускать его в Node, чтобы протестировать свой код. В поставку Node также включен интерактивный компонент *REPL* (сокращение от «Read-Eval-Print Loop»).

REPL (произносится «репл») поддерживает упрощенный строковый редактор и небольшой набор базовых команд. Код, который вы вводите в REPL, почти всегда выполняется точно так же, как если бы вы сохранили его в файле и запустили файл с использованием Node. Собственно, вы можете использовать REPL для написания кода всего приложения — буквально протестировать приложение «на ходу».

В этой главе я научу вас использовать REPL, а также опишу некоторые интересные особенности REPL и расскажу, как работать с ними. В частности, мы рассмотрим замену базового механизма сохранения команд, а также использование режима редактирования командной строки. А если встроенное ядро REPL не делает того, что вы хотите от интерактивной среды, также существует API для построения вашей собственной версии REPL.

Консоль, как и REPL, является одним из важнейших инструментов разработки Node. Мы используем консоль в большинстве приложений книги, однако возможности этого полезного объекта не ограничиваются простым выводом сообщений.

REPL: первые впечатления и неопределенные выражения

Чтобы запустить REPL, просто введите команду `node` без указания файла приложения:

```
$ node
```

REPL выдает приглашение командной строки — по умолчанию это угловая скобка (>).

Весь текст, который вы будете вводить после этого, будет обрабатываться ядром JavaScript V8.

Интерпретатор REPL очень прост в использовании. Вводите код JavaScript точно так же, как если бы собирались сохранить его в файле:

```
> a = 2;
2
```

Программа выводит результат полученного выражения. В этом фрагменте сеанса выражение равно 2. В следующем примере результат выражения представляет собой массив из трех элементов:

```
> b = ['a','b','c'];
[ 'a', 'b', 'c' ]
```

Для обращения к результату последнего выражения используется специальная переменная _ (символ подчеркивания). В следующем примере переменной `a` присваивается 2, полученное выражение увеличивается на 1, а потом снова на 1:

```
> a = 2;
2
> ++_;
3
> ++_;
4
```

Выражения с символом подчеркивания даже могут использоваться для обращения к свойствам или для вызова методов:

```
> ['apple','orange','lime']
[ 'apple', 'orange', 'lime' ]
> _.length
3
```

```
> 3 + 4
7
> _.toString();
'7'
```

Ключевое слово `var` может использоваться в REPL для будущих обращений к выражению или значению, но результаты порой оказываются неожиданными. Например, возьмем следующую строку в REPL:

```
var a = 2;
```

Она возвращает не значение 2, а неопределенное значение `undefined`. Дело в том, что присваивание переменной не возвращает результат при вычислении.

Следующий фрагмент дает некоторое представление о том, что происходит во внутренней работе REPL:

```
console.log(eval('a = 2'));
console.log(eval('var a = 2'));
```

Если сохранить этот фрагмент в файле и выполнить этот файл в Node, вы получите следующий результат:

```
2
undefined
```

Второй вызов `eval` не возвращает никакого значения; следовательно, возвращаемое значение не определено (`undefined`). В сокращении «Read-Eval-Print Loop» часть `eval` играет особенно важную роль.

Переменные используются в REPL так же, как и в приложениях Node:

```
> var a = 2;
undefined
> a++;
2
> a++;
3
```

Последние две команды выдают результаты, которые выводятся REPL.

Я покажу, как создать вашу собственную версию REPL — такую, в которой undefined не выводится, — в разделе «Специализированная версия REPL», с. 122.

Чтобы завершить сеанс REPL, нажмите либо Ctrl+C дважды, либо Ctrl+D один раз. Другие способы завершения сеанса будут рассмотрены позднее, в разделе «Команды REPL» на с. 119.

Преимущества REPL: понимание внутренних принципов работы JavaScript

Перед вами типичный пример использования REPL:

```
> 3 > 2 > 1;
false
```

Этот фрагмент наглядно демонстрирует, чем вам может пригодиться REPL. На первый взгляд введенное выражение должно дать результат `true`, так как 3 больше 2, а 2 больше 1. Однако в JavaScript операторы отношений обрабатываются слева направо, и результат каждого выражения возвращается для дальнейшей обработки.

В том, что происходит в этом фрагменте, проще разобраться в сеансе REPL:

```
> 3 > 2 > 1;
false
> 3 > 2;
true
> true > 1;
false
```

Теперь результат становится более понятным. Происходит следующее: сначала вычисляется выражение 3 > 2, для которого возвращается результат `true`. Но затем значение `true` сравнивается с числом 1. JavaScript предоставляет автоматическое преобразование типа данных, после которого `true` и 1 становятся эквивалентными значениями. В этом случае `true` не больше 1 и результат равен `false`.

Интерпретатор REPL особенно полезен именно для обнаружения подобных интересных странностей в коде JavaScript и наших приложений. После тестирования кода в REPL в ваших приложениях не должно быть неожиданных побочных эффектов (например, когда вы ожидаете получить результат `true`, но вместо него получаете `false`).

Многострочный и более сложный код JavaScript

В REPL можно вводить такой же код JavaScript, какой вы сохраняете в файлах, включая команды `require` для импортирования модулей. Ниже приведен протокол сеанса с использованием модуля `Query String` (`qs`):

```
$ node
> qs = require('querystring');
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
> val = qs.parse('file=main&file=secondary&test=one').file;
[ 'main', 'secondary' ]
```

Так как ключевое слово `var` не используется, выводится результат выражения — в данном случае это интерфейс объекта `querystring`. Неплохо? Вы не только получаете доступ к объекту, но и узнаете больше об интерфейсе объекта прямо во время работы с ним. А если вы предпочитаете обойтись без объемистого вывода, просто используйте ключевое слово `var`:

```
> var qs = require('querystring');
```

В любом случае вы сможете работать с объектом `querystring` через переменную `qs`.

Кроме возможности подключения внешних модулей, REPL корректно обрабатывает многострочные выражения, отображая текстовый индикатор вложения кода после открывающей фигурной скобки (`{`):

```
> var test = function (x, y) {
... var val = x * y;
... return val;
... };
undefined
> test(3,4);
12
```

Многоточия показывают, что дальнейший текст следует за открытой фигурной скобкой, а следовательно, команда еще не закончена. То же самое происходит и с открывающими круглыми скобками:

```
> test(4,
... 5);
20
```

При повышении уровня вложенности количество точек увеличивается; в интерактивной среде это необходимо, чтобы пользователь в любой момент понимал, к какому уровню относится вводимый код:

```
> var test = function (x, y) {
... var test2 = function (x, y) {
..... return x * y;
..... }
... return test2(x,y);
... }
undefined
> test(3,4);
12
>
```

Вы можете ввести (или скопировать) код целого приложения Node и запустить его из REPL. Я сократила фактические отображаемые значения для объекта `Server` (выделены жирным шрифтом), потому что они занимали слишком много места и с большой вероятностью изменятся бы к тому моменту, когда вы будете читать эту книгу:

```
> var http = require('http');
undefined
> http.createServer(function (req, res) {
...
...   // Заголовок контента
...   res.writeHead(200, {'Content-Type': 'text/plain'});
...
...   res.end("Hello person\n");
... }).listen(8124);
{ domain:
   { domain: null,
     _events: { error: [Function] },
     _maxListeners: undefined,
     members: [] },
  _
...
  httpAllowHalfOpen: false,
  timeout: 120000,
```

```
  _pendingResponseData: 0,
  _connectionKey: '6:null:8124' }
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
Undefined
```

Вы можете обратиться к приложению из браузера так же, как если бы вы ввели код в файл и запустили его из Node.

У того, что выражение не присваивается переменной, есть и «оборотная сторона»: где-то в середине приложения может появиться длинное описание объекта. Тем не менее одним из самых полезных применений REPL я считаю быстрое знакомство с объектами. Например, базовый объект Node `global` недостаточно подробно описан на сайте Node.js. Чтобы поближе познакомиться с ним, я открываю сеанс REPL и передаю объект методу `console.log`:

```
> console.log(global)
```

Я также могла действовать иначе — добавить переменную `gl` с тем же результатом (вывод сокращен для экономии места):

```
> gl = global;
...
_: [Circular],
  gl: [Circular] }
```

Простой вывод `global` предоставляет ту же информацию:

```
> global
```

Я не повторяю вывод REPL; вы можете сделать это самостоятельно в своей системе, поскольку интерфейс `global` весьма обширен. Из этого упражнения важно вынести то, что вы можете в любой момент быстро и просто получить информацию об интерфейсе объекта. Это поможет вам запомнить, как называется тот или иной метод или какие свойства поддерживаются объектом.

Объект global более подробно рассматривается в главе 2.

Клавиши ↑ и ↓ могут использоваться для перебора команд, вводившихся в REPL. Это удобный способ просмотра уже выполненных операций, а также редактирования вводившихся команд, хотя возможности такого редактирования ограниченны.

Возьмем следующий сеанс REPL:

```
> var myFruit = function(fruitArray,pickOne) {
... return fruitArray[pickOne - 1];
... }
undefined
> fruit = ['apples','oranges','limes','cherries'];
[ 'apples',  'oranges', 'limes',  'cherries' ]
> myFruit(fruit,2);
'oranges'
> myFruit(fruit,0);
undefined
> var myFruit = function(fruitArray,pickOne) {
... if (pickOne <= 0) return 'invalid number';
... return fruitArray[pickOne - 1];
... };
undefined
> myFruit(fruit,0);
'invalid number'
> myFruit(fruit,1);
'apples'
```

И хотя из листинга это не видно, когда я изменила функцию для проверки входного значения, на самом деле я перешла клавишей ↑ к началу объявления функции, а затем нажала Enter для ее перезапуска. Я добавила новую строку, а затем снова воспользовалась клавишами со стрелками для повторения ранее введенных команд, пока функция не была закончена. Клавиша ↑ также была использована для повторения вызова функции, приведшего к результату `undefined`.

Казалось бы, набрать код заново проще, чем возиться с клавишами, но представьте, что вы работаете с регулярными выражениями, например с такими:

```
> var ssRe = /^\d{3}-\d{2}-\d{4}$/;
undefined
> ssRe.test('555-55-5555');
true
> var decRe = /^\s*(\+|-)?((\d+(\.\d+)?)|(\.\d+))\s*$/;
undefined
> decRe.test(56.5);
true
```

У меня с регулярными выражениями вечно что-то не ладится, и мне приходится по нескольку раз править их, пока они не начнут делать то, что требуется. Использовать REPL для тестирования регулярных выражений очень удобно, однако набирать заново длинное регулярное выражение будет слишком долго и хлопотно.

К счастью, все, что от вас потребуется в REPL, — нажимать клавишу ↑, пока вы не найдете строку, в которой было создано регулярное выражение, внести изменения, нажать Enter и продолжить тестирование.

Кроме клавиш со стрелками вы также можете использовать клавишу Tab для автоматического завершения команды (в том случае, если она завершается однозначно). Например, введите `va` в командной строке и нажмите Tab; REPL предложит `var` и `valueOf`: два возможных варианта завершения введенного текста. С другой стороны, если вы введете `querys` и нажмете Tab, возможный вариант будет всего один: `querystring`. Клавиша Tab также может использоваться для автоматического завершения любой глобальной или локальной переменной. В табл. 4.1 приведена краткая сводка клавиатурных команд REPL.

Таблица 4.1. Клавиатурные команды в REPL

Клавиша или комбинация клавиш	Что делает
Ctrl+C	Завершает текущую команду. Повторное нажатие Ctrl+C прерывает сеанс
Ctrl+D	Выходит из REPL
Tab	Автоматически дополняет имя глобальной или локальной переменной
↑	Переходит на одну позицию назад в истории команд
↓	Переходит на одну позицию вперед в истории команд
Подчеркивание (_)	Представляет результат последнего выражения

Если вас беспокоит, что вы проведете много времени за программированием REPL, а потом вся проделанная работа пропадет, не волнуйтесь: результаты текущего контекста сохраняются командой `.save`. Эта и другие команды REPL рассматриваются в следующем разделе.

Команды REPL

REPL использует простой интерфейс с небольшим набором полезных команд. В предыдущем разделе упоминалась команда `.save`. Эта команда сохраняет ввод из текущего контекстного объекта в файле. Если только вы не создали новый контекст явно или не воспользовались командой `.clear`, контекст должен содержать весь ввод из текущего сеанса REPL:

```
> .save ./dir/session/save.js
```

Сохраняется только ваш ввод — так, словно вы набрали его прямо в текстовом редакторе и сохранили в файле.

Ниже приведен полный список команд REPL с краткими описаниями.

- `.break` — если вы запутались во время многострочного ввода, команда `.break` начнет все заново. Впрочем, введенные многострочные данные будут потеряны.
- `.clear` — сброс контекстного объекта с очисткой всех многострочных выражений. Фактически эта команда начинает работу заново.
- `.exit` — выход из REPL.
- `.help` — вывод полного списка команд REPL.
- `.save` — сохранение текущего сеанса REPL в файле.
- `.load` — загружает файл в текущем сеансе (`.load /path/to/file.js`).

Если вы работаете над приложением, используя REPL в качестве редактора, дам один совет: почаще сохраняйте свою работу командой `.save`. Хотя текущие команды сохраняются в истории, попытка воссоздания кода по истории может быть довольно неприятным делом.

Раз уж речь зашла о сохранении данных и истории, давайте посмотрим, как настроить оба этих аспекта REPL.

REPL и rlwrap

В документации REPL на сайте Node.js говорится о настройке переменной окружения, чтобы REPL можно было использовать с `rlwrap`. Что такое `rlwrap` и почему это нужно использовать с REPL?

Программа `rlwrap` — обертка, добавляющая функциональность библиотеки GNU `readline` в командную строку для повышения гибкости ввода с клавиатуры. Она перехватывает нажатия клавиш и предоставляет дополнительную функциональность, например расширенное редактирование строк и долгосрочную историю команд.

Вам придется установить `rlwrap` и `readline` для использования этой функциональности с REPL. Впрочем, в большинстве разновидностей Unix существуют простые системы установки пакетов. Например, в моей системе Ubuntu процедура установки `rlwrap` была совсем простой:

```
apt-get install rlwrap
```

Пользователи Mac должны использовать для таких приложений подходящую программу установки. Пользователям Windows придется использовать эмулятор среды Unix, например Cygwin.

Быстрая и наглядная демонстрация использования REPL с `rlwrap`, которая окрашивает приглашение REPL в фиолетовый цвет:

```
NODE_NO_READLINE=1 rlwrap -ppurple node
```

Чтобы приглашение REPL всегда выводилось фиолетовым цветом, можно добавить псевдоним в файл bashrc:

```
alias node="NODE_NO_READLINE=1 rlwrap -ppurple node"
```

Чтобы изменить как внешний вид приглашения, так и его цвет, я использую следующую команду:

```
NODE_NO_READLINE=1 rlwrap -ppurple -S "::> " node
```

Теперь выводится приглашение:

```
::>
```

с фиолетовым цветом символов.

В высшей степени полезный компонент `rlwrap` — возможность сохранения истории между сеансами REPL. По умолчанию история команд доступна только в сеансе REPL. Если вы используете `rlwrap`, при следующем обращении к REPL вы получите доступ не только к истории команд текущего сеанса, но и к истории команд прошлых сеансов (и другим данным, введенным в командной строке). В следующем примере команды не вводились, а были извлечены из истории клавишей ↑ *после* выхода и повторного входа в REPL:

```
::> e = ['a','b'];
[ 'a', 'b' ]
::> 3 > 2 > 1;
false
```

При всей полезности `rlwrap` мы все равно получаем `undefined` при вводе каждого выражения, которое не возвращает значение. Однако и эту проблему можно решить (вместе с другой функциональностью); для этого достаточно создать собственную версию REPL. Эта тема будет рассмотрена в следующем разделе.

Специализированная версия REPL

Node предоставляет программный интерфейс (API), который вы можете использовать для создания собственной версии REPL. Для этого сначала необходимо включить модуль `REPL`:

```
var repl = require("repl");
```

Чтобы создать новую версию REPL, вызовите метод `start` для объекта `repl`. Синтаксис метода выглядит так:

```
repl.start(options);
```

Объект `options` получает несколько значений; сейчас для нас представляют интерес следующие:

- `prompt` — значение по умолчанию >.
- `input` — поток для чтения; по умолчанию `process.stdin`.
- `output` — поток для записи; по умолчанию `process.stdout`.
- `eval` — по умолчанию `async`-обертка для `eval`.
- `useGlobal` — по умолчанию `false` (для создания нового контекста вместо использования глобального объекта).
- `useColors` — признак использования цветов функцией `writer`. По умолчанию используется значение REPL `terminal`.
- `ignoreUndefined` — по умолчанию `false` (неопределенные ответы не игнорируются).
- `terminal` — `true`, если поток должен рассматриваться как терминал, включая поддержку служебных кодов ANSI/VT100.
- `writer` — функция для обработки каждой команды и возвращения форматирования. По умолчанию `util.inspect`.
- `replMode` — режим выполнения REPL: строгий (strict), по умолчанию (default) или гибридный (hybrid).

Начиная с Node 5.8.0 repl.start() не требует передачи объекта options.

Мне неопределенные результаты выражений в REPL кажутся невыносимыми, поэтому я создала собственную специализацию REPL. Я также переопределила подсказку и установила строгий режим, чтобы каждая строка выполнялась в этом режиме:

```
var repl = require('repl');
repl.start( {
  prompt: 'my repl> ',
  replMode: repl.REPL_MODE_STRICT,
  ignoreUndefined: true,
});
```

Я запускаю файл repl.js с использованием Node:

```
node repl
```

Специализированная версия REPL используется точно так же, как и встроенная, если не считать измененного приглашения и отсутствия раздражающего `undefined` после первого присваивания. При этом я получаю другие ответы, отличные от `undefined`:

```
my repl> let ct = 0;
my repl> ct++;
0
my repl> console.log(ct);
1
my repl> ++ct;
2
my repl> console.log(ct);
2
```

В своем коде я хотела использовать значения по умолчанию для всех свойств, кроме перечисленных. Другим свойствам, не включенным в объект `options`, присвоены значения по умолчанию.

Функцию `eval` можно заменить вашей специализированной версией REPL. Единственное требование — конкретный формат функции:

```
function eval(cmd, callback) {
   callback(null, result);
}
```

Параметры `input` и `output` представляют интерес. Вы можете запустить несколько версий REPL, получая ввод как от стандартного ввода (по умолчанию), так и из сокетов. В документации REPL на сайте Node.js приведен пример прослушивания сокета TCP:

```
var net = require("net"),
    repl = require("repl");

connections = 0;

repl.start({
  prompt: "node via stdin> ",
  input: process.stdin,
  output: process.stdout
});

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via Unix socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  })
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via TCP socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  });
}).listen(5001);
```

Запустив это приложение, вы получите стандартное приглашение для выполнения приложений Node. Тем не менее к REPL также можно обратиться через TCP. Я использовала Telnet-клиент PuTTY для обращения к версии REPL с поддержкой TCP. Такой способ работает... до определенной степени. Мне пришлось сначала выдать команду `.clear`, отключить форматирование, а при попытке использовать символ подчеркивания для ссылки на последнее выражение у Node возникли проблемы. Я также пыталась использовать Telnet-клиента для Windows, — результат был еще хуже. Однако Telnet-клиент для Linux работал идеально.

Проблема, как нетрудно догадаться, кроется в настройках Telnet-клиента. Впрочем, дальше я разбираться не стала, потому что запуск REPL из сокета Telnet — не то, что я собиралась реализовать... или хотя бы рекомендовать — по

крайней мере, без усиленной безопасности. Это все равно что использовать `eval()` в коде на стороне клиента без проверки текста, который пользователи отправляют для выполнения, только еще хуже.

Для взаимодействия с работающим интерпретатором REPL через сокет Unix можно воспользоваться чем-то вроде утилиты GNU Netcat:

```
nc -U /tmp/node-repl-sock
```

Вы можете вводить команды практически так же, как при использовании `stdin`. Однако учтите, что при использовании сокетов TCP или Unix все команды `console.log` выводятся на консоль сервера, а не клиента:

```
console.log(someVariable); // Выводится на сервере
```

Я считаю более полезной другую возможность: создание приложения REPL с предварительной загрузкой модулей. В приложении из листинга 4.1 после запуска REPL сторонние модули `Request` (мощный HTTP-клиент), `Underscore` (библиотека вспомогательных инструментов) и `Q` (управление «обещаниями» — promises) загружаются и присваиваются контекстным свойствам.

Листинг 4.1. Создание специализированной версии REPL с предварительной загрузкой модулей

```
var repl = require('repl');
var context = repl.start({prompt: '>> ',
                          ignoreUndefined: true,
                          replMode: repl.REPL_MODE_STRICT}).context;

// Предварительная загрузка модулей
context.request = require('request');
context.underscore = require('underscore');
context.q = require('q');
```

При запуске приложения в Node появляется приглашение REPL, в котором можно обращаться к модулям:

```
>> request('http://blipdebit.com/phoenix5a.png')
.pipe(fs.createWriteStream('bird.png'))
```

Явно включать базовые модули Node не нужно; просто обращайтесь к ним напрямую по имени модуля.

Если вы хотите запускать приложение REPL как исполняемый файл в Linux, добавьте следующую строку в начало своего приложения:

```
#!/usr/local/bin/node
```

Разрешите исполнение для своего файла и запустите его:

```
$ chmod u+x replcontext.js
$ ./replcontext.js
>>
```

Бывает всякое — сохраняйтесь чаще

REPL — удобный интерактивный инструмент, который облегчает разработчику жизнь. REPL позволяет не только опробовать код JavaScript перед включением его в файлы, но и создавать приложения в интерактивном режиме, с сохранением результатов после завершения работы.

Другая полезная особенность REPL — возможность создания специализированной версии REPL, способной исключать нежелательные ответы `undefined`, осуществлять предварительную загрузку модулей, изменять приглашение или используемую функцию `eval` и делать многое другое.

Я также рекомендую вам присмотреться к использованию REPL с `rlwrap` для сохранения команд между сеансами. Эта возможность может заметно сэкономить ваше время. Кроме того, кто из нас не любит дополнительные возможности редактирования?

Однако в ходе изучения REPL следует руководствоваться одним очень важным правилом: бывает всякое. Сохраняйтесь чаще.

Если вы будете проводить много времени за разработкой в REPL, даже с использованием `rlwrap` для сохранения истории, постарайтесь часто сохранять свою работу. Работа в REPL не отличается от работы в других средах редактирования, поэтому я повторю еще раз: *бывает всякое — сохраняйтесь чаще*.

О необходимости консоли

Практически в каждом примере этой книги используется консоль (объект `console`). Консоль применяется для вывода значений, проверки операций, проверки правильности работы асинхронных механизмов приложения и для получения обратной связи.

Чаще всего я использую `console.log()` и ограничиваюсь выводом сообщений. Однако консоль — нечто большее, чем серверный аналог окна уведомления в браузере.

Типы консольных сообщений, класс Console и блокировка

В большинстве примеров книги используется метод `console.log()`, потому что в процессе экспериментов с Node нас в основном интересует обратная связь. Эта функция выводит сообщение в поток `stdout` (как правило, на терминал). Но когда вы начнете строить приложения Node, рассчитанные на реальную эксплуатацию, следует использовать другие функции консольного вывода.

Функция `console.info()` является эквивалентом `console.log()`. Обе функции записывают данные в `stdout`; обе выводят символ новой строки в составе сообщения. Функция `console.error()` отличается тем, что она выводит сообщение (также с символом новой строки) в `stderr`, а не в `stdout`:

```
console.error("An error occurred...");
```

Функция `console.warn()` делает то же самое.

Сообщения обоих типов выводятся на терминал; в чем разница, спросите вы? На самом деле различий нет. Чтобы понять это, необходимо повнимательнее познакомиться с объектом `console`.

ИСПОЛЬЗОВАНИЕ МОДУЛЕЙ ЖУРНАЛЬНОГО ВЫВОДА

Средства сохранения информации в журнале не ограничиваются встроенным объектом console. Существуют и более совершенные инструменты, например модули Bunyan (*https://github.com/trentm/node-bunyan*) и Winston (*https://github.com/winstonjs/winston*).

Прежде всего следует сказать, что `console` — глобальный объект, созданный на основе класса `Console`. При желании вы можете создать собственную версию `console` с использованием того же класса. И сделать это можно двумя разными способами.

Чтобы создать новый экземпляр `Console`, необходимо либо импортировать класс `Console`, либо обратиться к нему через глобальный объект `console`. В обоих случаях вы получаете новый объект, похожий на `console`:

```
// Использование require
var Console = require('console').Console;

var cons = new Console(process.stdout, process.stderr);
```

```
cons.log('testing');

// Использование существующего объекта console
var cons2 = new console.Console(process.stdout, process.stderr);

cons2.error('test');
```

Обратите внимание на передачу свойств `process.stdout` и `process.stderr` в качестве экземпляров потоков с поддержкой записи для журнальных сообщений и сообщений об ошибках соответственно. Глобальный объект `console` создается именно так.

Свойства `process.stdout` и `process.stderror` рассматривались в главе 2. О них нам известно, что они отображаются на дескрипторы `stdout` и `stderr` в среде выполнения и отличаются от многих потоков Node тем, что операции с ними обычно выполняются с блокировкой, то есть синхронно. Единственная ситуация, при которой они работают не синхронно, — это направление потоков в канал (pipe). Таким образом, в основном объект `console` блокируется как для `console.log()`, так и `console.error()`. Впрочем, это может создать проблемы лишь при передаче очень большого объема данных через поток.

Зачем использовать `console.error()` при возникновении ошибок? Чтобы обеспечить правильное поведение в тех средах, где эти два потока различны. Если в вашей среде вывод сообщений в журнал выполняется без блокировки, а вывод в поток ошибок — с блокировкой, следует позаботиться о том, чтобы вывод ошибок Node блокировался. Кроме того, при выполнении приложения Node выводы `console.log()` и `console.error()` можно направить в разные файлы средствами перенаправления командной строки. Следующий пример направляет сообщения `console.log()` в журнальный файл, а ошибки — в файл ошибок:

```
node app.js 1> app.log 2> error.log
```

Следующее приложение Node:

```
// Журнальные сообщения
console.log('this is informative');
console.info('this is more information');

// Сообщения об ошибках
console.error('this is an error');
console.warn('but this is only a warning');
```

направляет первые две строки в файл app.log, а следующие две — в error.log.

Вернемся к классу `Console`: вы можете продублировать функциональность глобального объекта `console`, использовав класс `Console` и передав `process.stdout` и `process.stderr`. Вы также можете создать новый объект, направляющий вывод в разные потоки (например, файлы журнала и ошибок). В документации `Console`, предоставленной Node Foundation, приводится как раз такой пример:

```
var output = fs.createWriteStream('./stdout.log');
var errorOutput = fs.createWriteStream('./stderr.log');
// Простой объект вывода в журнал
var logger = new Console(output, errorOutput);
// Используется как console
var count = 5;
logger.log('count: %d', count);
// В stdout.log: count 5
```

Преимущество подобных объектов заключается в том, что вы можете использовать глобальный объект `console` для получения общей обратной связи, а вновь созданный объект резервируется для вывода более формальных отчетов.

ОБЪЕКТ PROCESS И ПОТОКИ

Как упоминалось ранее, объект process рассматривается в главе 2, а потоки — в главе 6.

Форматирование сообщения с использованием util.format() и util.inspect()

Все четыре функции `console` — `log()`, `warn()`, `error()` и `info()` — могут получать данные любого типа, включая объекты. Не-объектные значения, которые не являются строками, преобразуются в строку. Если данные относятся к объектному типу, учтите, что Node отображает только два уровня вложенности. Если вам этого недостаточно, используйте метод `JSON.stringify()` с объектом, который строит более удобочитаемое дерево с отступами:

```
var test = {
   a : {
       b : {
          c : {
            d : 'test'
          }
       }
```

```
   }
}

// Выводятся только два уровня отступов
console.log(test);

// Выводятся три уровня отступов
var str = JSON.stringify(test, null, 3);
console.log(str);
```

Вывод приложения выглядит так:

```
{ a: { b: { c: [Object] } } }
{
   "a": {
      "b": {
         "c": {
            "d": "test"
         }
      }
   }
}
```

Если вы используете строку, используйте форматирование в стиле `printf` для всех четырех функций:

```
var val = 10.5;
var str = 'a string';

console.log('The value is %d and the string is %s', val, str);
```

Такое решение эффективно при работе с данными, переданными в аргументах функций или полученными из веб-запросов. Допустимый тип форматирования основан на форматировании, поддерживаемом вспомогательным методом `util.format()`; вы также можете воспользоваться этим модулем напрямую для создания строки:

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

var val = 10.5,
    str = 'a string';

var msg = util.format('The value is %d and the string is %s',val,str);
console.log(msg);
```

Впрочем, если вы используете только одну функцию, проще воспользоваться форматированием в `console.log()`. Допустимые форматные обозначения:

- `%s` — строка.
- `%d` — число (целое или вещественное).

- `%j` — JSON. Если аргумент содержит циклические ссылки, заменяется на `['circular']`.
- `%%` — литеральный знак процента (%).

Дополнительные аргументы преобразуются в строки и присоединяются к выводу. При недостаточном количестве аргументов выводится сам спецификатор:

```
var val = 3;

// Результат: 'val is 3 and str is %s'
console.log('val is %d and str is %s', val);
```

Для получения обратной связи используются не только эти четыре функции. Также имеется функция `console.dir()`.

Функция `console.dir()` отличается от других функций обратной связи тем, что переданный ей объект передается `util.inspect()`. Эта функция модуля `Utilities` позволяет более точно управлять способом отображения объекта при помощи дополнительного объекта `options`. Как и `util.format()`, она также может использоваться напрямую. Пример:

```
var test = {
   a : {
       b : {
          c : {
            d : 'test'
          }
       }
   }
}

var str = require('util').inspect(test, {showHidden: true, depth: 4 });
console.log(str);
```

Объект анализируется, и результат возвращается в виде строки в зависимости от того, какие параметры были переданы в объекте `options`. Возможные варианты:

- `showHidden`: для вывода неперечисляемых (non-enumerable) или символических свойств (по умолчанию `false`).
- `depth`: глубина анализа объекта (по умолчанию 2).
- `colors`: если значение равно `true`, то вывод оформляется с использованием цветовых кодов ANSI (по умолчанию `false`).

- `customInspect`: если параметр равен `false`, то специализированные функции анализа, определенные для анализируемых объектов, вызываться не будут (по умолчанию `true`).

Цветовое отображение может определяться глобально с использованием объекта `util.inspect.styles`. Также возможно изменение глобальных цветов. Используйте `console.log()` для вывода свойств объекта:

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

console.log(util.inspect.styles);
console.log(util.inspect.colors);
```

Приложение в листинге 4.2 изменяет выводимый объект, добавляет дату, число и логическое значение. Кроме того, цветовой код логического значения переключается с желтого на синий, чтобы оно отличалось от числа (по умолчанию оба выводятся синим цветом). Объект выводится разными способами: после обработки `util.inspect()`, с использованием `console.dir()` с теми же параметрами, с использованием базовой функции `console.log()` и с вызовом функции `JSON.stringify()` для объекта.

Листинг 4.2. Разные способы форматирования при выводе объекта

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

var today = new Date();

var test = {
   a : {
       b : {
          c : {
            d : 'test'
          },
          c2 : 3.50
       },
       b2 : true
   },
   a2: today
}

util.inspect.styles.boolean = 'blue';

// Использование util.inspect с прямым форматированием
var str = util.inspect(test, {depth: 4, colors: true });
console.log(str);
// Использование console.dir и options
```

```
console.dir(test, {depth: 4, colors: true});
// Использование базовой функции console.log
console.log(test);
// Использование JSON stringify
console.log(JSON.stringify(test, null, 4));
```

Результат показан на рис. 4.1. В своем окне терминала я использую белый фон с черным текстом.

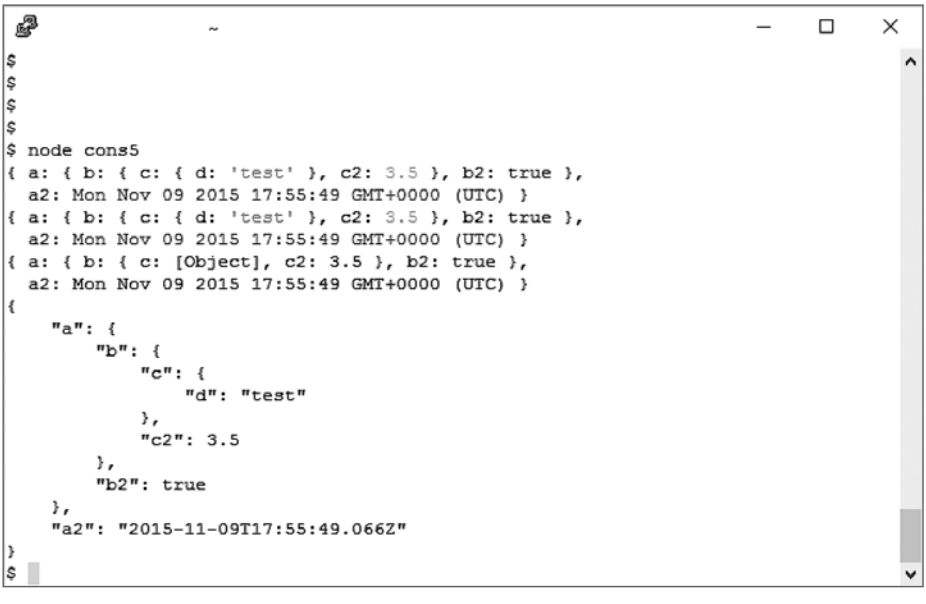


Рис. 4.1. Окно терминала с разными форматами строкового вывода

Функция console.dir() поддерживает три из четырех параметров util.inspect(): showHidden, depth и colors. Параметр customInspect не поддерживается. Если этот параметр содержит true, это означает, что объект поддерживает собственную функцию анализа.

Расширенная обратная связь с объектом console и таймером

Вернемся к объекту `console`: есть еще один способ получения более глубокого представления о том, что происходит в приложении. Для этого в приложение добавляется таймер, используемый для вывода начального и конечного времени.

Для реализации этой функциональности мы используем две консольные функции, `console.time()` и `console.timeEnd()`; каждой из этих функций передается имя таймера.

В следующем фрагменте используется более продолжительный цикл. Увеличенный промежуток времени нужен для того, чтобы функции таймера зарегистрировали разность:

```
console.time('the-loop');

for (var i = 0; i < 10000; i++) {
   ;
}

console.timeEnd('the-loop');
```

Даже с увеличенным циклом разность во времени едва заметна. Конкретная продолжительность зависит от загруженности машины, а также от процесса. Однако функциональность таймера не ограничивается синхронными событиями. Возможность присваивания таймерам имен означает, что эта функциональность может использоваться с асинхронными событиями.

Ниже я изменю приложение Hello, World из главы 1: в начале работы приложение будет запускать таймер, останавливать его для каждого веб-запроса, а потом перезапускать. Приложение измеряет интервалы между запросами. Конечно, можно было воспользоваться функцией `Date()` для получения более представительного таймера, но разве это интересно?

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

console.time('hello-timer');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
  console.timeEnd('hello-timer');
  console.time('hello-timer');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

А если говорить серьезно, этот код демонстрирует возможность включения таймеров в асинхронные операции благодаря возможности присваивания уникальных имен таймерам.

5 Node и Веб

Технология Node еще не достигла такого уровня, чтобы заменить повсеместно распространенную комбинацию Apache/PHP, но она быстро набирает популярность, прежде всего из-за простоты создания кросс-платформенных приложений и поддержки на уровне технологических гигантов.

В этой главе мы исследуем веб-корни Node, более глубоко рассмотрим модуль `HTTP`, а также попробуем создать очень простой статический веб-сервер. Также будут рассмотрены базовые модули Node, упрощающие веб-разработку.

Модуль HTTP: сервер и клиент

Не рассчитывайте, что модуль `HTTP` позволит создать веб-сервер с возможностями Apache или Nginx. Как указано в документации Node, этот сервер достаточно низкоуровневый, специализирующийся на обработке потоков и разборе сообщений. Тем не менее он предоставляет фундаментальную поддержку более сложной функциональности (например, Express — см. главу 10).

Модуль `HTTP` поддерживает несколько объектов, включая объект `http.Server`, который возвращается при использовании функции `http.createServer()` (см. главу 1). В этой главе мы встроили функцию обратного вызова для обработки веб-запроса, но также можно использовать отдельные события, так как `http.Server` наследует от `EventEmitter`.

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

var server = http.createServer().listen(8124);
```

```
server.on('request', function(request,response) {

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

console.log('server listening on 8214');
```

Также возможно прослушивание других событий, например создания подключения (`connect`) или запроса обновления клиентом (`upgrade`). Последнее событие происходит тогда, когда клиент запрашивает обновление версии HTTP или другого протокола.

ВНУТРЕННЕЕ УСТРОЙСТВО МОДУЛЯ HTTP

Для тех, кто захочет побольше узнать о внутреннем устройстве модуля HTTP: класс HTTP.Server в действительности является реализацией класса Net.Server на базе TCP, рассмотренного в главе 7. Протокол TCP предоставляет транспортный уровень, а HTTP — прикладной уровень коммуникаций.

Функция обратного вызова, используемая для ответов на веб-запросы, получает два параметра: запрос и ответ. Второй параметр (`response`) представляет собой объект типа `http.ServerReponse`. Это поток для записи, поддерживающий несколько функций, включая функцию `response.writeHead()` для создания заголовка ответа, `response.write()` для записи данных ответа и `response.end()` для завершения ответа.

ПОТОКИ ДЛЯ ЧТЕНИЯ И ЗАПИСИ

В главе 6 описаны обе разновидности потоков (для чтения и для записи).

Первый параметр, `request`, представляет собой результат класса `IncomingMessage`, который является *потоком для чтения*. Некоторые виды информации, которую можно получить из запроса:

- `request.headers` — объекты заголовков запроса/ответа.
- `request.httpVersion` — версия запроса HTTP.
- `request.method` — только для запроса `http.Server`; возвращает метод `HTTP` (`GET` или `POST`).

- `request.rawHeaders` — необработанные заголовки.
- `request.rawTrailers` — необработанные завершители.

Чтобы понять различия между `request.headers` и `request.rawHeaders`, выведите их с использованием `console.log()` в запросе. Обратите внимание: значения задаются свойствами для `request.headers` и элементами массива для `request.rawHeaders`; в первом элементе массива хранится свойство, а во втором — значение (на случай, если вы захотите обратиться к отдельным значениям):

```
console.log(request.headers);
console.log(request.rawHeaders);

// Получение хоста
console.log(request.headers.host);
console.log(request.rawHeaders[0] +  ' is ' + request.rawHeaders[1]);
```

В документации Node обратите внимание на то, что некоторые свойства `IncomingMessage` (`statusCode` и `statusMessage`) доступны только для *ответа* (не запроса) из объекта `HTTP.ClientRequest`. Помимо создания сервера, прослушивающего запросы, вы также можете создать клиент, который эти запросы выдает. Для этого используется класс `ClientRequest`, экземпляр которого создается функцией `http.request()`.

Чтобы продемонстрировать оба типа функциональности, серверную и клиентскую, я возьму пример кода создания клиента из документации Node и изменю его так, чтобы он обращался к серверу, локальному для клиента. В коде клиента я создаю метод `POST` для отправки данных, а это означает, что мне придется изменить свой сервер для чтения этих данных. Здесь в игру вступает часть `IncomingMessage`, связанная с потоком для чтения. Вместо того чтобы прослушивать событие `request`, приложение прослушивает одно или несколько событий `data`, используемых для получения *фрагментов* (chunks) данных в запросе. Приложение продолжает получать фрагменты до тех пор, пока не получит событие `end` для объекта запроса. Затем другой полезный модуль Node, `Query String` (рассматривается более подробно в разделе «Разбор запроса с использованием Query String», с. 152), используется для разбора данных и вывода их на консоль. Только после этого происходит отправка ответа.

Код измененного сервера приведен в листинге 5.1. Обратите внимание: он очень похож на то, что вы пытались делать ранее, если не считать новой обработки событий для данных, отправленных методом `POST`.

Листинг 5.1. Сервер прослушивает отправку POST и обрабатывает отправленные данные

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

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
        var body = '';

        // Фрагмент данных присоединяется к body
        request.on('data', function (data) {
            body += data;

        });

        // Переданные данные
        request.on('end', function () {
            var post = querystring.parse(body);
            console.log(post);
            response.writeHead(200, {'Content-Type': 'text/plain'});
            response.end('Hello World\n');
        });
    }
});
console.log('server listening on 8214');
```

Код нового клиента показан в листинге 5.2. И снова в нем используется `http.ClientRequest` — реализация потока для записи, на что очевидно указывает использование метода `req.write()` в примере.

Код почти полностью повторяет пример из документации Node, если не считать сервера, к которому мы обращаемся. Так как и сервер и клиент находятся на одном компьютере, в качестве хоста указывается `localhost`. Кроме того, свойство `path` в `options` не указывается, так как используется значение по умолчанию `/`. Для заголовков запроса назначается тип контента `application/x-www-form-urlencoded`, используемый при отправке методом `POST`. Также обратите внимание на то, что клиент получает данные от сервера через объект `response`, который является единственным аргументом функции обратного вызова, передаваемой функции `http.request()`. Данные, отправленные методом `POST`, извлекаются из ответа по фрагментам, но на этот раз каждый фрагмент выводится на консоль сразу же после получения. Возвращаемое сообщение очень короткое, поэтому инициируется всего одно событие `data`.

Отправка запроса методом POST не обрабатывается асинхронно, потому что мы инициируем действие без блокировки в ожидании его выполнения.

Листинг 5.2. Клиент HTTP отправляет данные серверу

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

var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');

  // Получение данных по фрагментам
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });

  // Завершение ответа
  res.on('end', function() {
    console.log('No more data in response.')
  })
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// Запись данных в тело запроса
req.write(postData);
req.end();
```

Откройте второй терминал, запустите сервер в первом терминале, а затем запустите клиент во втором. Клиент обращается с приветствием к серверу, а тот выдает ответное приветствие.

И все это просто для того, чтобы два процесса сказали «привет» друг другу! Но при этом вы реализовали двусторонний обмен данными между клиентом и сервером, а также получили возможность поработать с данными, переданными методом `POST` (вместо простого использования `GET`). И заодно вы познакомились со всеми классами модуля `HTTP`, кроме одного: `http.ClientRequest`, `http.Server`, `http.IncomingMessage` и `http.ServerResponse`.

В программе не встречается только класс `http.Agent`, используемый для опроса сокетов. Node поддерживает пул сокетов для обработки запросов, выданных вызовами `http.request()` или `http.get()` (во втором случае — упрощенный запрос `GET` без тела). Если приложение выдает много запросов, ограниченный размер пула может стать его «узким местом». Проблему можно обойти запретом использования пула подключений, присвоив свойству `agent` в свойствах исходящего запроса значение `false`. В листинге 5.2 для этого в объект `options` необходимо внести следующее изменение (выделенное жирным шрифтом):

```
var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  },
  agent: false
};
```

Также можно изменить максимальный размер пула сокетов в свойстве `agent.maxFreeSockets`. По умолчанию он равен `256`. Однако учтите, что изменение пула подключений исходящих запросов может отрицательно повлиять на использование памяти и других ресурсов.

В главе 7 вам представится возможность поработать с еще более сложными коммуникациями. А пока разберемся, что необходимо для создания сервера HTTP, возможности которого не ограничиваются простым возвращением строки «привет».

Что необходимо учесть при создании статического веб-сервера

Вся функциональность, необходимая для построения простого маршрутизатора запросов или предоставления статических файлов, встроена прямо в Node. Однако «возможно» и «удобно и просто» — далеко не одно и то же.

Если подумать над тем, что необходимо для построения простого, но функционального сервера со статическими файлами, можно предложить следующую последовательность шагов:

1. Создать сервер HTTP и прослушивать запросы.
2. При поступлении запроса разобрать URL для определения местонахождения файла.
3. Проверить, что файл существует.
4. Если файл не существует, отреагировать соответствующим образом.
5. Если файл существует, открыть его для чтения.
6. Подготовить заголовок ответа.
7. Записать файл в ответ
8. Ожидать следующего запроса.

Для выполнения всех этих функций понадобятся только базовые модули (с одним исключением, как будет показано позднее в этом разделе). Для создания сервера HTTP и чтения файлов потребуются модули `HTTP` и `File System`. Кроме того, следует определить глобальную переменную для базового каталога или использовать заранее определенную переменную `__dirname` (подробнее о ней рассказано на врезке «Когда не используется __dirname?», с. 150).

На данной стадии приложение должно начинаться со следующего фрагмента:

```
var http = require('http'),
    fs = require('fs'),
    base = '/home/examples/public_html';
```

Веб-сервер был создан в примере «Hello, World» главы 1; мы возьмем его за основу для кода этой главы. Функция `http.createServer()` создает сервер с функцией обратного вызова, получающей два значения: веб-запрос и ответ, который мы создадим. Чтобы получить запрашиваемый документ, приложение может напрямую обратиться к свойству `url` объекта запроса HTTP. Чтобы убедиться в том, что ответы соответствуют запросам, мы также вставим вызов `console.log` с путем к запрашиваемому файлу. Это сообщение добавляется к сообщению `console.log`, выводимому при запуске сервера:

```
var http = require('http'),
    fs = require('fs'),
    base = '/home/examples/public_html';

http.createServer(function (req, res) {
```

```
    pathname = base + req.url;
    console.log(pathname);

}).listen(8124);

console.log('Server web running at 8124');
```

При запуске приложения стартует веб-сервер, прослушивающий запросы на порте 8124.

ОБ ИСПОЛЬЗОВАНИИ ПОРТОВ

Как правило, при обращении к сайту в браузере номер порта указывать не обязательно. Дело в том, что два стандартных номера портов — 80 для запросов HTTP, 443 для HTTPS — встраиваются браузерами в запрос автоматически. В Википедии приведен список всех стандартных номеров портов.

Мы ведем прослушивание на порте 8214, потому что портом по умолчанию 80 обычно управляет основной веб-сервер вашей среды разработки, будь то Ngnix, Apache, lighttpd, более новый Caddy или приложение Node.

Также можно воспользоваться стандартным портом 80, но порты с номерами ниже 1024 требуют привилегий суперпользователя (root). Даже Apache не предоставляет экземпляры серверов, работающие с привилегиями суперпользователя. Сам Apache работает с привилегиями суперпользователя, но порождает дочерние потоки, выполняемые на более ограниченных уровнях. Вообще говоря, выполнять веб-серверы с привилегиями суперпользователя не рекомендуется.

Вы можете настроить внутренние механизмы сервера (например, воспользоваться iptables) для перенаправления обращений к порту 1024 на порт 80, но я не рекомендовала бы действовать так. Лучше настроить прокси-сервер (Apache, Ngnix, lighttpd или Caddy), который будет передавать запросы веб-серверу. О том, как использовать Apache в качестве прокси-сервера для приложений Node, более подробно рассказывается в разделе «Использование Apache в качестве прокси-сервера для приложений Node», с. 151.

Примерный результат тестирования приложения с файлом index.html (например, *http://blipdebit.com:8124/index.html*) будет выглядеть примерно так (в зависимости от конкретной среды):

```
/home/examples/public_html/index.html
```

Конечно, браузер «зависает», потому что мы не создали ответ, но сейчас мы займемся и этим.

Также перед открытием и чтением файла можно проверить его на доступность. Функция `fs.stat()` возвращает ошибку, если файл не существует.

УДАЛЕНИЕ ФАЙЛА ПОСЛЕ ПРОВЕРКИ

Одна из потенциальных опасностей решения с fs.stat() заключается в том, что файл по каким-то причинам может исчезнуть между проверкой и его открытием. Другое решение — просто открыть файл напрямую; если он отсутствует, вы получите сообщение об ошибке. Однако у использования чего-то вроде fs.open() с последующей передачей дескриптора файла fs.createReadStream() тоже есть недостаток: функция не предоставляет информации о том, является файл каталогом или файлом (а также о том, что он отсутствует или заблокирован). Поэтому я использую fs.stat(), а также проверяю ошибку при открытии потока для чтения, чтобы лучше контролировать процесс обработки ошибок.

Раз уж речь зашла о чтении из файла, вы можете попробовать использовать `fs.readFile()` для чтения содержимого файла. Проблема в том, что `fs.readFile()` пытается полностью прочитать файл в память, прежде чем предоставить доступ к нему. Документы, предоставляемые веб-серверами, могут быть довольно большими. Кроме того, в любой момент времени для каждого документа может существовать много запросов. Решения с `fs.readFile()` не масштабируются.

Вместо того чтобы использовать `fs.readFile()`, приложение создает поток для чтения методом `fs.createReadStream()` с настройками по умолчанию. Далее остается просто направить содержимое файла прямо в объект ответа HTTP. Так как поток отправляет сигнал о достижении конца данных, использовать метод `end` с потоком не нужно:

```
res.setHeader('Content-Type', 'test/html');

// Создание и перенаправление потока для чтения
var file = fs.createReadStream(pathname);
file.on("open", function() {
   // Код 200 - файл найден, ошибок нет
   res.statusCode = 200;
   file.pipe(res);
});
file.on("error", function(err) {
  res.writeHead(403);
  res.write('file missing, or permission problem');
  console.log(err);
});
```

Поток для чтения поддерживает два интересных события: `open` и `error`. Событие `open` генерируется при готовности потока, а `error` — при возникновении ошибки. Какой именно? Например, если файл исчез после проверки состояния, если доступ невозможен из-за разрешений или файл оказался подкаталогом... Так как в этот момент неизвестно, что произошло, программа просто выдает общую ошибку 403, которая объединяет большую часть потенциальных проблем. Мы записываем сообщение, которое может отображаться или не отображаться в зависимости от того, как браузер реагирует на ошибку.

Приложение вызывает метод `pipe` в функции обратного вызова для события `open`.

На данный момент веб-сервер, предоставляющий статические файлы, выглядит так, как показано в листинге 5.3.

Листинг 5.3. Простой веб-сервер, предоставляющий статические файлы

```
var http = require('http'),
    fs   = require('fs'),
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   fs.stat(pathname, function(err,stats) {
      if (err) {
        console.log(err);
        res.writeHead(404);
        res.write('Resource missing 404\n');
        res.end();
      } else {
         res.setHeader('Content-Type', 'text/html');

         // Создание и перенаправление потока для чтения
         var file = fs.createReadStream(pathname);

         file.on("open", function() {
            res.statusCode = 200;
            file.pipe(res);
         });

         file.on("error", function(err) {
           console.log(err);
           res.writeHead(403);
           res.write('file missing or permission problem');
```

```
          res.end();
        });

      }
    });
}).listen(8124);

console.log('Server running at 8124');
```

Я протестировала приложение на простом HTML-файле, содержащем всего один элемент `img`. Файл загрузился и отображался корректно:

```
<!DOCTYPE html>
<head>
   <title>Test</title>
   <meta charset="utf-8" />
</head>
<body>
<img src="./phoenix5a.png" />
</body>
```

Также приложение было протестировано на несуществующем файле. Ошибка была обнаружена при использовании `fs.stat()`; браузеру было возвращено сообщение 404, а на консоли появилось сообщение об отсутствии файла или каталога.

Затем я протестировала его на файле, для которого были сняты все разрешения на чтение. На этот раз ошибка была обнаружена потоком для чтения; браузеру было отправлено сообщение о невозможности чтения из файла, а на консоль была выведена ошибка разрешений. Также для файла с отсутствием разрешений был возвращен более подходящий код состояния HTTP 403.

Наконец, я протестировала серверное приложение с другим файлом, содержащим элемент HTML5 `video`:

```
<!DOCTYPE html>
<head>
   <title>Video</title>
   <meta charset="utf-8" />
</head>
<body>
   <video id="meadow" controls>
      <source src="videofile.mp4" />
      <source src="videofile.ogv" />
      <source src="videofile.webm" />
   </video>
</body>
```

При попытке открыть страницу в Chrome файл открывался, а видео воспроизводилось, но при тестировании страницы в Internet Explorer 10 элемент `video` не работал. Причина прояснилась при взгляде на консольный вывод:

```
Server running at 8124/
/home/examples/public_html/html5media/chapter1/example2.html
/home/examples/public_html/html5media/chapter1/videofile.mp4
/home/examples/public_html/html5media/chapter1/videofile.ogv
/home/examples/public_html/html5media/chapter1/videofile.webm
```

Хотя IE10 может воспроизводить видео MP4, он проверяет все три формата, потому что для каждого в заголовке ответа указан тип контента `text/html`. И хотя другие браузеры игнорируют неправильный тип контента и отображают видео правильно, IE10 так не поступает — и правильно, на мой взгляд, потому что иначе мне не удалось бы так быстро обнаружить ошибку в приложении.

В своем новом браузере Edge компания Microsoft изменила обработку типа контента, и в нем отображается подходящее видео. Тем не менее приложение должно работать правильно. Необходимо изменить его так, чтобы оно проверяло расширение каждого файла и возвращало подходящий тип MIME в заголовке ответа. Эту функциональность можно реализовать самостоятельно, но я воспользуюсь существующим модулем `Mime`.

Модуль Mime устанавливается с использованием npm: npm install mime. Этот модуль доступен на сайте GitHub (*https://github.com/broofa/node-mime*).

Модуль `Mime` может вернуть правильный тип MIME по имени файла (с путем или без), а также возвращает расширения файлов для заданного типа контента. Модуль `Mime` включается командой:

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

Возвращаемый тип контента используется в заголовке ответа, а также выводится на консоль, чтобы значение можно было проверить в ходе тестирования приложения:

```
// Тип контента
var type = mime.lookup(pathname);
console.log(type);
res.setHeader('Content-Type', type);
```

Теперь при обращении к файлу с элементом `video` в IE10 файл нормально работает. Для всех браузеров возвращается правильный тип контента.

Впрочем, кое-что все же не работает: программа дает сбой при обращении к каталогу вместо файла. В таком случае на консоль выводится ошибка:

```
{ [Error: EISDIR, illegal operation on a directory] errno: 28, code:
          'EISDIR' }
```

Но браузеры ведут себя по-разному. Edge выводит обобщенную ошибку 403 (рис. 5.1), а Firefox и Chrome выводят обобщенное сообщение об ошибке. Однако невозможность обращения к подкаталогу и проблемы с файлом — не одно и то же.

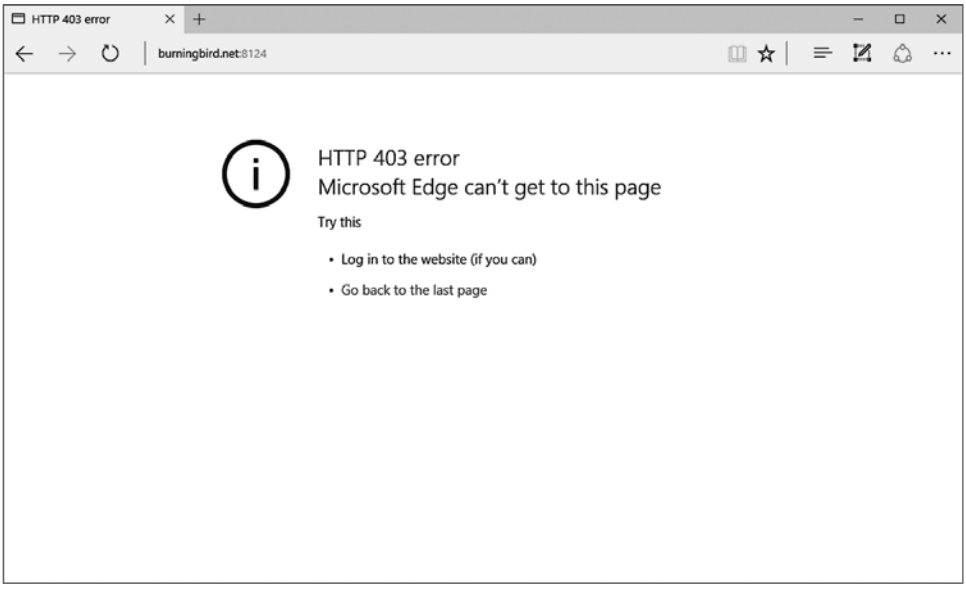


Рис. 5.1. Невозможность обращения к каталогу в Edge

Нужно не только убедиться в том, что ресурс существует, но и проверить, является он файлом или каталогом. Если программа обращается к каталогу, можно либо вывести его содержимое, либо выдать сообщение об ошибке — это решает разработчик.

Остается внести еще одно изменение. Базовый путь, который я использую, прекрасно работает на моем сервере с системой Linux, но при попытке использовать его на компьютере с Windows начнутся проблемы. Во-первых,

на машине с Windows нет каталога /home/examples/public_html. Во-вторых, на компьютере с Windows вместо слеша используется обратный слеш (\).

Чтобы приложение работало в обеих средах, я сначала создаю структуру каталога в Windows. Затем базовый модуль `Path` используется для нормализации строки пути, чтобы она работала в обеих средах. Функция `path.normalize()` получает строку и возвращает либо ту же строку, если она уже нормализована для существующего окружения, либо преобразованную строку.

```
var pathname = path.normalize(base + req.url);
```

Теперь приложение работает на обеих машинах.

Модуль Path подробно рассматривается в главе 6.

Последняя версия минимального сервера со статическими файлами в листинге 5.4 использует `fs.stats` для проверки существования запрашиваемого объекта и проверки того, является ли он файлом. Если ресурс не существует, то возвращается код состояния HTTP 404. Если ресурс существует, но является каталогом, то возвращается код состояния HTTP 403 (доступ запрещен). То же происходит, если файл заблокирован, но сообщение изменяется. Во всех случаях выдается более уместный ответ.

Листинг 5.4. Последняя версия минимального сервера для статических файлов

```
var http = require('http'),
    url =  require('url'),
    fs   = require('fs'),
    mime = require('mime'),
    path = require('path');

var base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = path.normalize(base + req.url);
   console.log(pathname);

   fs.stat(pathname, function(err, stats) {
      if (err) {
        res.writeHead(404);
```

```
        res.write('Resource missing 404\n');
        res.end();
      } else if (stats.isFile()) {
         // Тип контента
         var type = mime.lookup(pathname);
         console.log(type);
         res.setHeader('Content-Type', type);

         // Создание и перенаправление потока для чтения
         var file = fs.createReadStream(pathname);
         file.on("open", function() {
            // Код 200 - файл найден, ошибок нет
            res.statusCode = 200;
            file.pipe(res);
         });
         file.on("error", function(err) {
           console.log(err);
           res.statusCode = 403;
           res.write('file permission');
           res.end();
         });
       } else {
        res.writeHead(403);
        res.write('Directory access is forbidden');
        res.end();
       }
    });
}).listen(8124);
console.log('Server running at 8124');
```

Ниже приведен консольный вывод при обращении к веб-странице, содержащей ссылки на графику и видео, при использовании Firefox:

```
/home/examples/public_html/video.html
text/html
/home/examples/public_html/phoenix5a.png
image/png
/home/examples/public_html/videofile.mp4
video/mp4
/home/examples/public_html/favicon.ico
image/x-icon
/home/examples/public_html/favicon.ico
image/x-icon
```

Обратите внимание на правильность обработки типов контента — не говоря уже об автоматических обращениях к значкам сайтов, которые Firefox и Chrome присоединяют к запросам страниц.

Чтобы лучше понять, как работает поток для чтения, попробуйте загрузить страницу с элементом `video` и начните воспроизводить его. Браузер получает вывод потока для чтения с той скоростью, которую он может обеспечить, заполняет свой внутренний буфер, а затем приостанавливает вывод. Если вы закроете сервер во время воспроизведения видеоконтента, то видео продолжит воспроизводиться вплоть до опустошения буфера. После этого изображение пропадает, потому что поток для чтения становится недоступным. Просто поразительно, насколько хорошо все работает при минимуме усилий с нашей стороны.

КОГДА НЕ ИСПОЛЬЗУЕТСЯ __DIRNAME?

В некоторых примерах из книги базовое местонахождение веб-документов жестко запрограммировано: как правило, в виде /home/examples/public_html. Спрашивается, почему я не использую __dirname?

В Node вы можете использовать заранее определенную переменную __dirname для обозначения текущего рабочего каталога в приложении Node. Однако в примерах этой главы я обращаюсь к файлам, хранящимся отдельно от моего приложения Node; именно поэтому я не использую __dirname. Во всех остальных случаях это следует делать. Это позволяет протестировать приложения, а затем переместить их в среду реальной эксплуатации без изменения значения переменной базового каталога.

Пример использования переменной __dirname:

```
var pathname = __dirname + req.url;
```

Обратите внимание на двойной символ подчеркивания в __dirname.

Хотя приложение работает при тестировании с несколькими разными документами, оно не идеально. Оно не обрабатывает много других типов веб-запросов, оно не поддерживает средства безопасности и кэширование, и оно некорректно обрабатывает запросы видео. Одно из приложений веб-страниц с использованием видео HTML, с которым я тестировала приложение, также применяло API элемента HTML5 `video` для вывода состояния процесса загрузки видео. Это приложение не могло получить информацию, необходимую ему для работы.

При создании сервера для статических файлов существует великое множество мелочей, на которых можно споткнуться. По этой причине разработчики предпочитают использовать более сложные системы, такие как Express (см. главу 9).

Использование Apache в качестве прокси-сервера для приложения Node

Вряд ли можно ожидать, что Node в недалеком будущем заменит более традиционные веб-серверы (такие, как Apache). Тем не менее реализация части функциональности Node в то время, как сервер делает все остальное, — вполне приемлемый вариант. Так как Apache остается наиболее популярным веб-сервером, я сосредоточусь на нем в этом разделе.

Вопрос в том, как запустить сервер Node одновременно с выполнением другого сервера (того же Apache), не заставляя пользователя указывать порт? Только один веб-сервер может отвечать на запросы к порту 80, используемому по умолчанию. Правда и то, что, если мы предоставляем API для браузеров, номер порта может быть скрыт, — но что, если вы хотите использовать и Apache и Node одновременно без конфликта портов?

Другие аспекты реальной эксплуатации приложения — неограниченно долгое выполнение Node и восстановление после сбоя — рассматриваются в главе 11.

Простейший подход к выполнению приложения Node с Apache — использование Apache в качестве прокси-сервера для запросов приложения Node. Это означает, что все запросы к приложению Node будут сначала проходить через Apache.

У такого решения есть как достоинства, так и недостатки. К достоинствам можно отнести то, что оно чрезвычайно просто, а запросы перед попаданием к приложению Node будут обрабатываться исключительно надежным и популярным веб-сервером. Apache предоставляет средства безопасности и другую функциональность, которую было бы чрезвычайно трудно реализовать в приложении Node. Недостаток заключается в том, что Apache порождает новый программный поток для каждого запроса, а количество потоков не бесконечно.

Тем не менее многие сайты работают на базе Apache, притом с вполне приличной скоростью. Если вы не собираетесь конкурировать с отраслевыми гигантами, это хороший вариант.

Чтобы Apache выполнял функции прокси-сервера для Node, сначала необходимо включить проксирование в Apache. Введите в командной строке следующие команды:

```
a2enmod proxy
a2enmod proxy_http
```

Затем добавьте проксирование в поддомене. Например, при использовании моего сервера я указываю *shelleystoybox.com* в качестве своего приложения Node:

```
<VirtualHost ipaddress:80>
    ServerAdmin shelleyp@burningbird.net
    ServerName shelleystoybox.com

    ErrorLog path-to-logs/error.log
    CustomLog path-to-logs/access.log combined

    ProxyRequests off

    <Location />
            ProxyPass http://ipaddress:2368/
            ProxyPassReverse http://ipaddress:2368/
    </Location>
</VirtualHost>
```

Измените поддомен, адрес электронной почты администратора, порт и IP-адрес в соответствии с конфигурацией вашей среды. Далее остается только загрузить новый поддомен:

```
a2ensite shelleystoybox.com
service apache2 reload
```

ОБРАЩЕНИЕ К САЙТУ ЧЕРЕЗ ПОРТ

Использование прокси-сервера не помешает людям явно обращаться к сайту с указанием порта. Вы можете блокировать такую возможность, но это потребует нетривиальной настройки сервера. Например, на своем сервере с Ubuntu необходимые изменения вносятся при помощи iptables:

iptables -A input -i eth0 -p tcp --dport 2368 -j DROP

Тем не менее для такой настройки потребуются навыки администрирования сервера.

Разбор строки запроса с использованием Query String

Ранее в этой главе у вас уже была возможность поработать с модулем `Query String`. Его единственная цель — подготовка и обработка строк запросов.

Полученная строка запроса может быть преобразована в объект вызовом `querystring.parse()`, как показано в листинге 5.1. Разделитель строки запроса по умолчанию (`'&'`) может переопределяться необязательным вторым параметром функции, а присваивание по умолчанию (`'='`) может переопределяться третьим параметром. Четвертый необязательный параметр содержит метод `decodeURIComponent`; по умолчанию используется `querystring.unescape()`. Измените его, если строка запроса не будет передаваться в кодировке UTF-8. Тем не менее с большой вероятностью строка запроса будет использовать стандартный разделитель и присваивание и будет храниться в UTF-8, поэтому вы можете принять значения по умолчанию.

Чтобы вы лучше поняли, как работает функция `querystring.parse()`, возьмем следующий запрос:

```
somedomain.com/?value1=valueone&value1=valueoneb&value2=valuetwo
```

Функция `querystring.parse()` вернет следующий объект:

```
{
   value1: ['valueone','valueoneb'],
   value2: 'valuetwo'
}
```

При подготовке строки запроса для отправки, как показано в листинге 5.2, используется вызов `querystring.stringify()` с передачей кодируемого объекта. Таким образом, если у вас имеется объект вроде того, который мы только что создали, передайте его `querystring.stringify()`, и вы получите отформатированную строку запроса, готовую для передачи. В листинге 5.2 `querystring.stringify()` возвращает строку:

```
msg=Hello%20World!
```

Обратите внимание на экранирование (escaping) пробела. Функция `querystring.stringify()` получает те же необязательные параметры, за исключением последнего. В данном случае передается реализация `encodeURIComponent`, а по умолчанию используется `stringify.escape()`.

Преобразование DNS

Вряд ли вашим приложениям часто потребуется напрямую пользоваться сервисом DNS, но если такая необходимость все же возникнет, нужная функциональность предоставляется в базовом модуле `DNS`.

Мы рассмотрим две функции модуля DNS: `dns.lookup()` и `dns.resolve()`. Функция `dns.lookup()` возвращает первый полученный IP-адрес для доменного имени. В следующем примере возвращается первый найденный IP-адрес для *oreilly.com*:

```
dns.lookup('oreilly.com', function(err,address,family) {
     if (err) return console.log(err);

     console.log(address);
     console.log(family);
});
```

В параметре `address` содержится полученный IP-адрес, а параметр `family` равен 4 или 6 в зависимости от типа адреса (IPv4 или IPv6). Вы также можете задать объект `options`:

- `family` — число (4 или 6), представляющее тип адреса (IPv4 или IPv6).
- `hints` — поддерживаемые флаги `getaddrinfo` (число).
- `all` — если значение равно `true`, возвращаются все адреса (по умолчанию `false`).

Мне захотелось увидеть все IP-адреса, поэтому я внесла изменения в код. При этом мне пришлось исключить параметр `family`, который при получении всех адресов принимает значение `undefined`. Я получаю массив объектов IP-адресов, с полями `address` и `family`:

```
dns.lookup('oreilly.com', {all: true}, function(err,family) {
     if (err) return console.log(err);

     console.log(family);
});
```

Возвращаемый массив:

```
[ { address: '209.204.146.71', family: 4 },
  { address: '209.204.146.70', family: 4 } ]
```

Функция `dns.resolve()` возвращает тип записи для имени хоста. Поддерживаются следующие типы (в виде строк):

- `A` — адрес IPv4 (по умолчанию).
- `AAAA` — адрес IPv6.
- `MX` — адрес почтового шлюза.

- `TXT` — произвольный текст.
- `SRV` — запись SRV.
- `PTR` — используется для обратного преобразования DNS.
- `NS` — сервер имен.
- `CNAME` — запись канонического имени.
- `SOA` — начало указания на авторитетность информации.

В следующем примере я использую `dns.resolve()` для получения всех записей MX для *oreilly.com*:

```
dns.resolve('oreilly.com','MX', function(err,addresses) {
   if (err) return err;
   console.log(addresses);
});
```

Возвращаемый массив выглядит так:

```
[ { exchange: 'aspmx.l.google.com', priority: 1 },
  { exchange: 'alt1.aspmx.l.google.com', priority: 5 },
  { exchange: 'aspmx2.googlemail.com', priority: 10 },
  { exchange: 'alt2.aspmx.l.google.com', priority: 5 },
  { exchange: 'aspmx3.googlemail.com', priority: 10 } ]
```

6 Node и локальная система

Модуль файловой системы (`File System`) Node постоянно используется в книге. Мало найдется ресурсов, более важных для приложений, чем ресурсы файловой системы. Пожалуй, чаще используются только сетевые ресурсы, рассмотренные в главах 5 и 7.

Одна из самых замечательных особенностей Node заключается в том, что модуль `File System` работает более или менее одинаково в разных операционных системах. Разработчики Node стараются сделать так, чтобы вся функциональность, строящаяся на основе технологии, не зависела от операционной системы. Иногда у них это получается, иногда требуется небольшая помощь от сторонних модулей.

В этой главе модуль `File System` описан более подробно. Кроме того, в ней рассматривается функциональность, специфическая для конкретной ОС, и различия между системами. Напоследок мы рассмотрим два модуля, `ReadLine` и `Zlib`, обеспечивающие интерактивную передачу данных в режиме командной строки и средствами сжатия соответственно.

Знакомство с операционной системой

Некоторым технологиям удается полностью маскировать различия между операционными системами. В других случаях для того, чтобы справиться со спецификой отдельных ОС, разработчику приходится основательно потрудиться. Node занимает промежуточное положение. Обычно приложения, созданные вами, работают в любых системах. Однако существует ряд функциональных областей, в которых проявляются различия между ОС. Как упоминалось

в начале главы, иногда Node справляется с ними хорошо, а иногда разработчику приходится прибегать к услугам удобных сторонних модулей.

Для непосредственного получения информации об операционной системе используется базовый модуль `OS`. Это один из полезных инструментов, упрощающих построение кросс-платформенных приложений. Кроме того, он предоставляет информацию об использовании ресурсов и возможностях текущей среды.

Обращение к модулю `OS` начинается с команды `require`:

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

Вся функциональность модуля `OS` направлена только на получение информации. Например, если вы хотите обеспечить кросс-платформенную поддержку, можно проверить, какой завершитель строки поддерживается текущей системой, использует она прямой (little-endian) или обратный (big-endian) порядок байтов, а также узнать временный и домашний каталог:

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

console.log('Using end of line' + os.EOL + 'to insert a new line');
console.log(os.endianness());
console.log(os.tmpdir());
console.log(os.homedir());
```

На моем сервере с Ubuntu и компьютере с Windows 10 используется прямой порядок байтов, а завершитель строки (EOL, End-Of-Line) в обеих системах работает так, как и следовало ожидать (вторая часть текста начинается с новой строки). Различаются только временный и домашний каталоги, что вполне естественно.

ВРЕМЕННЫЙ КАТАЛОГ

Временный каталог используется для хранения временных файлов. Его содержимое удаляется при перезапуске системы или с заданной периодичностью.

Модуль `OS` также предоставляет средства для проверки доступных ресурсов текущей машины.

```
console.log(os.freemem());
console.log(os.loadavg());
console.log(os.totalmem());
```

Функция `os.loadavg()` специфична для Unix; в Windows она просто возвращает нули. Она возвращает показатель средней загрузки, отражающий текущую интенсивность работы системы, за 1, 5 и 15 минут. Чтобы получить значения в процентах, умножьте числа на 100. Функции `os.freemen()` и `os.totalmem()` возвращают объем памяти в байтах.

Другая функция, `os.cpus()`, возвращает информацию о процессорах машины. Возвращается количество миллисекунд, проведенных процессором в разных режимах: `user`, `nice`, `sys`, `idle` и `irq`. Если вы не знакомы с этими концепциями: значение `user` определяет время, проведенное процессором за выполнением процессов пользовательского пространства, `idle` — время бездействия, а `sys` — время, проведенное за выполнением системных процессов (режим ядра). Значение `nice` отражает величину динамической регулировки приоритета, предотвращающей слишком частое его выполнение. Значение `irq` описывает прерывания — запросы на обслуживание на аппаратном уровне.

Время в миллисекундах не так удобно, как проценты. Чтобы определить их, можно просуммировать все значения, а затем вычислить проценты. Также можно воспользоваться модулями независимых разработчиков, которые возвращают значения в процентах (наряду с другой информацией).

Microsoft предоставляет хорошее описание того, как это работает в Azure (*https://blogs.msdn.microsoft.com/azureossds/2015/08/23/finding-memory-leaks-and-cpu-usage-in-azure-node-js-web-app/*), но информация и перечисленные модули должны работать во всех средах.

Потоки и pipe()

Потоковая технология проявляется во всех базовых аспектах Node; она предоставляет функциональность для HTTP, а также для других форм сетевой передачи данных. Кроме того, она предоставляет функциональность для файловой системы, поэтому я рассматриваю ее до того, как мы займемся более глубоким изучением модуля `File System`.

Поток представлен абстрактным интерфейсом, это означает, что вы не будете создавать потоки напрямую. Вместо этого вы будете работать с различными объектами, реализующими интерфейс `Stream`, — запросами HTTP, потоками для чтения или записи модуля `File System`, объектами сжатия `Zlib` или `process.stdout`. Непосредственно реализовать Stream API вам придется только в одном случае: при создании собственной реализации потока. Так как эта тема выходит за рамки учебника начального уровня, я оставлю ее вам для

самостоятельного изучения. А пока мы сосредоточимся на общем поведении потоков, к которым вы получаете доступ при использовании другой функциональности.

Так как многие объекты в Node реализуют потоковый интерфейс, все потоки в Node обладают базовой функциональностью:

- Изменение кодировки потоковых данных вызовом `setEncoding`.
- Проверка возможности чтения и (или) записи данных в поток.
- Перехват событий потоков (например, *получения данных* или *закрытия подключения*) с назначением функций обратного вызова для каждого события.
- Приостановка и возобновление потока.
- Перенаправление данных из потока для чтения в поток для записи.

Обратите внимание на пункт с проверкой чтения и (или) записи. Поток с поддержкой чтения и записи называется *дуплексным*. Также существует подвид дуплексных потоков, называемый *потоком преобразования данных*, в котором ввод и вывод связаны причинной зависимостью. Такая разновидность потоков будет описана позднее в этой главе, когда я буду рассматривать сжатие `Zlib`.

Поток для чтения начинает работу в приостановленном состоянии; это означает, что никакие данные не будут отправляться до того момента, пока не будет явно выполнена операция чтения (`stream.read()`) или команда возобновления работы потока (`stream.resume()`). Однако используемые нами реализации потоков, такие как поток для чтения модуля `File System`, переключаются в рабочий режим сразу же при программировании события данных (механизм получения доступа к данным в потоке для чтения). В рабочем режиме данные передаются приложению сразу же при их появлении.

Потоки для чтения поддерживают несколько событий, но на практике нас обычно интересуют три события: `data`, `end` и `error`. Событие `data` отправляется при получении нового фрагмента данных, готового к использованию, а событие `end` — при потреблении всех данных. Событие `error`, естественно, отправляется при возникновении ошибки. Пример использования потока для чтения был представлен в листинге 5.1 в главе 5. Ниже вы увидите в примерах использование модуля `File System`.

Поток для записи представляет собой приемник, в который передаются (записываются) данные. Среди прослушиваемых событий можно выделить

error и событие finish, происходящее при вызове end() и сбросе всех данных. Также встречается событие drain, генерируемое в тот момент, когда попытка записи данных возвращает false. Мы использовали поток для записи при создании клиента HTTP в листинге 5.2 в главе 5; также вы увидите его в действии в примерах File System в разделе «Знакомство с модулем File System», с. 161.

Дуплексный поток обладает качествами потоков как для чтения, так и для записи. Поток преобразования данных представляет собой дуплексный поток, в котором — в отличие от обычных дуплексных потоков, где внутренние входные и выходные буферы существуют независимо друг от друга, — эти два буфера связаны напрямую через промежуточный этап преобразования данных. Во внутренней реализации поток преобразования данных должен реализовать функцию _transform(), которая получает входные данные, что-то с ними делает, а затем записывает в вывод.

Чтобы лучше понять суть потока преобразования данных, необходимо поближе познакомиться с функциональностью, поддерживаемой всеми потоками: функцией pipe(). Мы видели пример ее использования в листинге 5.1, где поток для чтения напрямую связывал содержимое файла с объектом ответа HTTP:

```
// Создание и перенаправление потока для чтения
         var file = fs.createReadStream(pathname);
         file.on("open", function() {
            file.pipe(res);
         });
```

Вызов pipe() извлекает данные из файла (поток) и выводит их в объект http.ServerResponse. В документации Node указано, что этот объект реализует интерфейс потока для записи и, как будет показано позднее, fs.createReadStream() возвращает fs.ReadStream — реализацию потока для чтения. В число методов, поддерживаемых потоком для чтения, входит и pipe() с потоком для записи.

Позднее в книге будет приведен пример использования модуля Zlib для сжатия файла, а пока ограничимся кратким примером, отлично демонстрирующим применение потока преобразования данных:

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

На вход поступает поток для чтения, на выходе находится поток для записи. Содержимое одного потока передается в другой, но сначала проходит через процедуру сжатия (это и есть преобразование).

Знакомство с модулем File System

Модуль Node `File System` (`fs`) предоставляет всю функциональность, необходимую для работы с файловой системой независимо от операционной системы. Он уже использовался во многих примерах книги, а теперь пора более систематично познакомиться с его функциональностью.

Во-первых, как указано в документации Node, модуль `File System` представляет собой набор оберток, работающих с функциями POSIX. По сути это означает, что модуль поддерживает POSIX-стандартизированную (кросс-платформенную) функциональность обращения к файловым системам, которая работает во всех поддерживаемых операционных системах. Таким образом, ваше приложение может работать в OS X, Linux и Windows, а также в некоторых новых средах, таких как Android, или микрокомпьютерах вроде Raspberry Pi.

Модуль `File System` предоставляет как синхронные версии функций, так и традиционные для Node асинхронные версии. Спорить о том, хорошо это или плохо, бессмысленно; функции существуют, и мы можем использовать или не использовать их.

Асинхронные функции получают в последнем аргументе функцию обратного вызова, получающую ошибку в первом аргументе, тогда как синхронные функции немедленно генерируют ошибку при ее возникновении. С синхронными функциями `File System` можно использовать традиционную конструкцию `try…catch`, а в асинхронных версиях используется объект `error`. В остальных примерах этого раздела я сосредоточусь исключительно на асинхронных функциях; тем не менее помните, что синхронные версии тоже существуют.

Кроме многочисленных функций, модуль `File System` поддерживает четыре класса:

- `fs.FSWatcher` — поддержка событий для отслеживания изменений в файле.
- `fs.ReadStream` — поток для чтения.
- `fs.WriteStream` — поток для записи.
- `fs.Stats` — информация, возвращаемая функциями `*stat`.

Класс fs.Stats

Объект `fs.Stats` возвращается при использовании функций `fs.stat()`, `fs.lstat()` и `fs.fstat()`. Он может использоваться для проверки существования файла (или каталога), но также возвращает информацию о том, является ли объект файловой системы файлом/каталогом/сокетом домена UNIX, какие разрешения связаны с файлом, время последнего обращения или модификации объекта и т. д. Node предоставляет специализированные функции для обращения к информации, например функции `fs.isFile()` и `fs.isDirectory()` для определения того, является ли объект файлом или каталогом. Также к данным можно обращаться напрямую:

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

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);
   console.log(util.inspect(stats));
});
```

В Linux вы получите структуру данных, которая выглядит примерно так:

```
{ dev: 2048,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 1419012,
  size: 219840,
  blocks: 432,
  atime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  mtime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  ctime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC),
  birthtime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC) }
```

Фактически это вывод функции POSIX `stat()`, возвращающей информацию о статусе файла. Даты понятны без объяснений, но с другими значениями все не так очевидно. Размер (`size`) задан в байтах, в поле `blksize` хранится размер блока операционной системы, а в поле `blocks` — количество блоков. Два последних значения в Windows не определены.

Среди более интересных компонентов обратите внимание на поле `mode`: в нем хранятся разрешения объекта. Проблема в том, что с ходу расшифровать его содержимое довольно трудно.

Здесь на помощь приходит вспомогательная функция или, в случае Node, вспомогательный модуль. Специализированный модуль `stat-mode` получает объект `stat`, возвращенный функцией (такой, как `fs.stat()`), и позволяет напрямую запрашивать нужные значения. В листинге 6.1 продемонстрировано, как с помощью этого модуля получить полезную информацию о разрешениях файла.

Листинг 6.1. Получение файловых разрешений с использованием модуля stat-mode

```
var fs = require('fs');
var Mode = require('stat-mode');

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);

   // Получение разрешений
   var mode = new Mode(stats);

   console.log(mode.toString());
   console.log('Group execute ' + mode.group.execute);
   console.log('Others write ' + mode.others.write);
   console.log('Owner read ' + mode.owner.read);
});
```

Результат для конкретного файла выглядит так:

```
-rw-rw-r--
Group execute false
Others write false
Owner read true
```

Отслеживание изменений в файловой системе

На практике приложениям нередко приходится «прослушивать» изменения в файлах или каталогах и выполнять некоторую задачу при обнаружении изменений. В Node для этой цели используется интерфейс `fs.FSWatcher`. К сожалению, как замечают разработчики Node, он непоследовательно ведет себя на некоторых платформах и пользы от него немного.

Мы не будем использовать его и связанную функцию `fs.watch()`, которая возвращает объект. Вместо этого мы обратимся к модулю независимых разработчиков. Модуль `Chokidar` (*https://github.com/paulmillr/chokidar*) ежемесячно загружается более двух миллионов раз, что делает его одним из наиболее популярных (не говоря уже о том, что он встроен в популярное приложение Gulp).

Установите модуль следующей командой (добавьте `-g` для глобальной установки):

```
npm install chokidar
```

Следующий код добавляет наблюдателя (watcher) для текущего каталога. Наблюдатель отслеживает изменения в каталоге, включая изменения в файлах. Он выполняет рекурсивный поиск, включая новые подкаталоги, содержащиеся в родительском каталоге, и новые файлы в этих подкаталогах. Событие `raw` перехватывает все события, а другие обработчики предназначены для более высокоуровневых событий.

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

var watcher = chokidar.watch('.', {
  ignored: /[\/\\]\./,
  persistent: true
});

var log = console.log.bind(console);

watcher
  .on('add', function(path) { log('File', path, 'has been added'); })
  .on('unlink', function(path) { log('File', path, 'has been removed'); })
  .on('addDir', function(path) { log('Directory', path, 'has been added');
})
  .on('unlinkDir', function(path) {
       log('Directory', path, 'has been removed'); })
  .on('error', function(error) { log('Error happened', error); })
  .on('ready', function() { log('Initial scan complete. Ready for
changes.'); })
  .on('raw', function(event, path, details) {
       log('Raw event info:', event, path, details); });

watcher.on('change', function(path, stats) {
  if (stats) log('File', path, 'changed size to', stats.size);
});
```

Информация о добавлении или удалении файлов или каталогов, как и информация об изменении размеров файлов, появляется на консоли. Имена функций `unlink()` и `unlinkDir()` отражают тот факт, что под «удалением» понимается разрыв связи объекта с текущим каталогом. При исчезновении последней ссылки файлы/подкаталоги перестают существовать.

При перехвате всех низкоуровневых событий `raw` выходные данные могут занимать много места. Тем не менее эксперименты с `Chokidar` стоит начинать именно с них.

Чтение и запись файлов

Включите модуль перед использованием:

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

В большинстве примеров, в которых используется модуль `File System`, задействованы непотоковые методы чтения и записи. Существует два основных способа чтения или записи в файл с применением непотоковой функциональности.

В первом варианте используются очень простые методы: `fs.readFile()` или `fs.writeFile()` (или их синхронные аналоги). Эти функции открывают файл, выполняют чтение или запись, после чего закрывают файл. В следующем фрагменте файл открывается для записи (при этом его текущее содержимое, если оно есть, теряется). После завершения записи файл открывается для чтения, его содержимое читается и выводится на консоль.

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

fs.writeFile('./some.txt','Writing to a file',function(err) {
      if (err) return console.error(err);
      fs.readFile('./some.txt','utf-8', function(data,err) {
         if (err) return console.error(err);
         console.log(data);
      });
});
```

Так как файловые операции ввода и вывода по умолчанию осуществляются через буфер, при чтении файла во втором аргументе функции `fs.readFile()` передается значение `'utf-8'`. Также можно было преобразовать буфер в строку.

Во втором способе чтения/записи файл открывается с назначением файлового дескриптора (`fd`). Файловый дескриптор используется для записи и (или) чтения из файла. К преимуществам данного способа следует отнести то, что вы можете более точно управлять режимом открытия файла и тем, что с ним можно сделать.

Следующий пример создает файл, записывает в него данные, а затем читает их. Второй параметр функции `fs.open()` содержит флаг, определяющий, какие действия могут выполняться с файлом. В данном случае передается значение `'a+'`, при котором файл открывается для присоединения и (или) чтения; если файл не существует, то он создается. Третий параметр задает разрешения доступа к файлу (разрешены чтение и запись).

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0x666, function(err, fd) {
     if (err) return console.error(err);
     fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
        if (err) return console.error(err);
        var buf = new Buffer(written);
        fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
           if (err) return console.error(err);
           console.log(buf.toString('utf8'));
        });
      });
});
```

Файловый дескриптор возвращается в функции обратного вызова, а затем используется в функции `fs.write()`. Строка записывается в файл с позиции 0. Однако следует учесть, что, согласно документации Node, при открытии файла в режиме присоединения в Linux данные всегда записываются в конец файла (позиционный индикатор игнорируется). Функция обратного вызова для `fs.write()` возвращает ошибку (если она происходит), количество записанных байтов и записанную строку. Наконец, функция `fs.read()` используется для чтения строки в буфер, после чего строка выводится на консоль.

Конечно, на практике читать только что записанную строку не нужно, однако этот пример демонстрирует три основных типа методов, используемых в этом способе чтения/записи в файл. Кроме файлов вы также можете напрямую работать с каталогами.

Работа с каталогами

Основные операции с каталогами — создание, удаление и чтение содержащихся в них файлов. Также существует возможность создания символических ссылок или удаления ссылок, что приводит к уничтожению файла (при условии, что он не открыт ни в одной программе). Для усечения файла (сокращения его до нуля байтов) используется функция `truncate()`. В этом случае сам файл остается, но его содержимое пропадает.

Следующий пример демонстрирует работу с каталогами: он выводит список файлов в текущем каталоге, и если какие-либо из этих файлов хранятся в сжатом виде (расширение `.gz`), они удаляются. Для простоты мы воспользуемся модулем `Path`, описанным в разделе «Обращение к ресурсам с модулем Path», с. 170.

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

Файловые потоки

Мы рассмотрели примеры работы с потоками для чтения и записи, но я хочу ненадолго остановиться и привести более подробное изложение.

Поток для чтения создается вызовом `fs.createReadStream()` с передачей пути и объекта `options` или же с включением описания файла в `options` без указания пути. То же можно сказать о потоках для записи, создаваемых вызовом `fs.createWriteStream()`. В обоих случаях поддерживается объект `options`. По умолчанию поток для чтения создается со следующими параметрами:

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

Если вы хотите использовать файловый дескриптор, задайте его в объекте `options`. Свойство `autoClose` автоматически закрывает файл при завершении чтения. Если вы хотите прочитать часть файла, задайте начальную и конечную позицию (в байтах) в полях `start` и `end`. Вы можете выбрать `'utf8'` или другую кодировку, но это можно сделать и позднее вызовом `setEncoding()`.

ПОТОКИ ДОКУМЕНТИРУЮТСЯ В РАЗНЫХ МЕСТАХ

Функция setEncoding(), которую вы можете вызвать с потоком для чтения, созданным вызовом fs.createReadStream(), на самом деле документирована в разделе с описанием потоков документации Node. Если вы работаете с потоками Node, учтите, что документация разбросана по разным местам и вам, возможно, придется потратить некоторое время на поиски.

Вскоре я приведу пример использования потоков для чтения модуля `File System`, но сначала рассмотрим набор параметров потока для записи `File System`, созданного вызовом `fs.createWriteStream()`. По умолчанию объект `options` содержит следующие значения:

```
{ flags: 'w',
  defaultEncoding: 'utf8',
  fd: null,
  mode: 0o666 }
```

И снова вместо пути можно использовать файловый дескриптор. Кодировка потока для записи задается свойством `defaultEncoding` (вместо `encoding`). Если вы хотите начать запись с определенного смещения от начала файла, задайте свойство `start`. Свойство `end` не задается, потому что оно определяется после записи нового содержимого.

А теперь объединим все вместе. В листинге 6.2 я открываю файл для изменения, используя поток для записи. В этом примере строка просто вставляется в заданную позицию файла. В этом примере используется файловый дескриптор, поэтому при вызове `fs.createWriteStream()` приложение не инициирует открытие файла одновременно с созданием потока для записи.

Листинг 6.2. Модификация существующего файла

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

fs.open('./working.txt', 'r+',function (err, fd) {
   if (err) return console.error(err);

   var writable = fs.createWriteStream(null,{fd: fd, start: 10,
                                        defaultEncoding: 'utf8'});

   writable.write(' inserting this text ');

});
```

Обратите внимание: файл открывается с флагом `r+`. Это позволяет приложению как читать, так и изменять файл.

В листинге 6.3 открывается тот же файл, но на этот раз для чтения содержимого. При открытии используется флаг `r`, так как файл будет использоваться только для чтения. Приложение читает все содержимое файла. При этом кодировка изменяется на `utf8` функцией `setEncoding()`. В листинге 6.2 кодировка для записи изменялась флагом `defaultEncoding`.

Листинг 6.3. Чтение содержимого файла с использованием потока

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

var readable =
  fs.createReadStream('./working.txt').setEncoding('utf8');

var data = '';
readable.on('data', function(chunk) {
   data += chunk;
});

readable.on('end', function() {
   console.log(data);
});
```

Если запустить приложение, читающее данные, до и после приложения, изменяющего файл, вы наглядно увидите изменения. При первом запуске читающего приложения возвращается содержимое working.txt:

```
Now let's pull this all together, and read and write with a stream.
```

Во второй раз возвращается новое содержимое:

```
Now let's  inserting this text  and read and write with a stream.
```

Было бы намного быстрее и удобнее, если бы мы могли просто открыть файл для чтения и направить результаты в поток для чтения. Сделать это нетрудно — достаточно воспользоваться функцией `pipe()` потока для чтения. Однако результаты не удастся изменить в ходе операции, потому что поток для записи делает только то, что следует из названия: принимает записанные данные. Это не дуплексный поток, а еще конкретнее — не поток преобразования данных, способный изменить содержимое. Однако содержимое можно скопировать из одного файла в другой.

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

var readable =
  fs.createReadStream('./working.txt');

var writable = fs.createWriteStream('./working2.txt');

readable.pipe(writable);
```

Поток преобразования данных будет представлен позднее, в разделе «Сжатие/восстановление данных с использованием Zlib», с. 174.

Обращение к ресурсам с модулем Path

Вспомогательный модуль Node `Path` предоставляет средства преобразования и извлечения данных с использованием путей файловой системы. Он также предоставляет механизм работы с путями файловой системы, не зависящий от среды выполнения, так что вам не придется программировать разные модули для Linux и Windows. Операция получения расширения файла уже встречалась ранее, когда мы извлекали расширения при переборе файлов в каталоге:

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

Для получения базового имени файла используйте следующий код:

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      let ext = path.extname(file);
      let base = path.basename(file, ext);
      console.log ('file ' + base + ' with extension of ' + ext);
   }
});
```

Второй аргумент функции `path.basename()` приводит к тому, что возвращается только имя без расширения.

В качестве примера платформенной независимости модуля `Path` можно привести свойство `path.delimeter`. В нем хранится разделитель, действующий в конкретной системе. В Linux это двоеточие (`:`), а в Windows точка с запятой (`;`).

Если вы захотите разбить значения переменной среды PATH способом, работающим в обеих операционных системах, используйте path.delimeter:

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

console.log(process.env.PATH);
console.log(process.env.PATH.split(path.delimiter));
```

Последнее решение работает в обеих системах и возвращает массив переменных PATH.

Другое различие между системами — использование слеша (/) или обратного слеша (\) в путях. В главе 5 я создала простой веб-сервер, использовавший пути файловой системы для предоставления ресурсов по веб-запросам. На моей машине с Linux файловые пути разделяются слешем, а на машине с Windows — обратным слешем. Чтобы приложение работало на обеих платформах, я использовала вызов path.normalize():

```
pathname = path.normalize(base + req.url);
```

Ключевая особенность модуля Path заключается вовсе не в том, что он выполняет удивительные преобразования, которые невозможно проделать с использованием объекта String или RegExp. Главное в другом: преобразования путей файловой системы выполняются независимо от операционной системы.

Если вам потребуется разобрать путь файловой системы на компоненты, используйте функцию path.parse(). Результаты довольно серьезно различаются в зависимости от операционной системы. В Windows при использовании require.main.filename или сокращения __filename (путь и имя выполняемого приложения) в аргументе я получаю следующий результат:

```
{ root: 'C:\\',
  dir: 'C:\\Users\\Shelley',
  base: 'work',
  ext: '.js',
  name: 'path1' }
```

На сервере с Ubuntu результат выглядит так:

```
{ root: '/',
  dir: '/home/examples/public_html/learnnode2',
  base: 'path1.js',
  ext: '.js',
  name: 'path1' }
```

Создание программы командной строки

В системе Unix разработчик может легко создать приложение Node, которое запускается напрямую без использования команды `node`.

ПРОГРАММЫ КОМАНДНОЙ СТРОКИ В СИСТЕМЕ WINDOWS

Чтобы создать программу командной строки для Windows, вам придется создать пакетный файл с вызовом Node и именем приложения.

Для демонстрации этой возможности я воспользуюсь модулем `Commander` (см. главу 3) и *дочерним процессом* для запуска ImageMagick — мощной графической программы.

В своем приложении я использую ImageMagick для применения к существующему изображению эффекта Polaroid и сохранения результата в новом файле. Как видно из листинга 6.4, модуль `Commander` используется для обработки аргументов командной строки и предоставления справки по использованию программы.

Листинг 6.4. Приложение Node в формате программы командной строки

```
#!/usr/bin/env node

var spawn = require('child_process').spawn;
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source graphic file name')
   .option ('-f, --file [file name]', 'Resulting file name')
   .parse(process.argv);

if ((program.source === undefined) || (program.file === undefined)) {
   console.error('source and file must be provided');
   process.exit();
}

var photo = program.source;
var file = program.file;

// Массив преобразования
var opts = [
photo,
```

```
"-bordercolor", "snow",
"-border", "20",
"-background","gray60",
"-background", "none",
"-rotate", "6",
"-background", "black",
"(", "+clone", "-shadow", "60x8+8+8", ")",
"+swap",
"-background", "none",
"-thumbnail", "240x240",
"-flatten",
file];

var im = spawn('convert', opts);

im.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

im.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

Чтобы преобразовать этот код в программу командной строки, я добавила в начало файла следующую строку:

```
#!/usr/bin/env node
```

За символами `#!` следует приложение, которое должно использоваться для выполнения файла, в данном случае Node. Подкаталог определяет путь, по которому находится приложение.

Файл сохраняется без расширения .js. Чтобы разрешить его исполнение, воспользуйтесь программой `chmod`:

```
chmod a+x polaroid
```

Теперь программу можно запустить следующей командой:

```
./polaroid -h
```

для получения справки по ее использованию (благодаря `Commander`), или

```
./polaroid -s phoenix5a.png -f phoenix5apolaroid.png
```

для создания нового изображения с примененным эффектом. Программа не работает в Windows, но в остальных средах проблем нет.

Создание программы командной строки не то же самое, что создание автономного приложения. Второе подразумевает, что приложение может быть установлено без предварительной установки Node (или других зависимостей).

СОЗДАНИЕ АВТОНОМНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ NW.JS

На момент написания книги мне известна только одна функциональность, позволяющая создавать автономные приложения Node: Intel NW.js (ранее node-webkit). Вы можете воспользоваться ею для упаковки своих файлов с последующим запуском пакета при помощи библиотеки NW.js. Библиотека предоставляет все необходимое для работы приложения.

Сжатие/восстановление данных с использованием Zlib

Модуль `Zlib` предоставляет функциональность сжатия/восстановления данных. В его основу заложен поток преобразования данных, что становится очевидным после знакомства с примером сжатия файла, приведенным в документации Node. Я слегка изменила этот пример для работы с большим файлом.

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

var gzip = zlib.createGzip();

var inp = fs.createReadStream('test.png');
var out = fs.createWriteStream('test.png.gz');

inp.pipe(gzip).pipe(out);
```

Входной поток напрямую связывается с выходным, а передаваемые между ними данные подвергаются сжатию gzip, то есть происходит преобразование данных, в данном случае файла PNG.

`Zlib` предоставляет поддержку сжатия `zlib` и `deflate` — более сложного и управляемого алгоритма сжатия. Если файлы, созданные с применением `zlib`, могут быть распакованы программой командной строки `gunzip` (или `unzip`), у файлов, созданных с применением `deflate`, такая возможность отсутствует. Для восстановления файла, сжатого в режиме `deflate`, придется использовать Node или другую функциональность.

Чтобы продемонстрировать функциональность сжатия и восстановления файлов, мы создадим две программы командной строки: `compress` и `uncompress`.

Первая программа сжимает файл с выбором алгоритма `gzip` или `deflate` в параметрах командной строки. Так как мы собираемся работать с параметрами командной строки, для них также следует включить модуль `Commander`:

```
var zlib = require('zlib');
var program = require('commander');
var fs = require('fs');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source File Name')
   .option ('-f, --file [file name]', 'Destination File name')
   .option ('-t, --type <mode>', /^(gzip|deflate)$/i)
   .parse(process.argv);

var compress;
if (program.type == 'deflate')
   compress = zlib.createDeflate();
else
   compress = zlib.createGzip();

var inp = fs.createReadStream(program.source);
var out = fs.createWriteStream(program.file);

inp.pipe(compress).pipe(out);
```

Такие приложения интересны и полезны (особенно в среде Windows, в которой нет встроенной реализации сжатия), но технология сжатия особенно популярна в веб-запросах. Документация Node содержит примеры функциональности Zlib с веб-запросами. Также приведены примеры получения сжатых файлов с использованием модуля `Request` (см. главу 5) и функции `http.request()`.

Вместо получения сжатых файлов я покажу, как отправить сжатый файл на сервер, где он затем будет восстановлен. Я адаптирую код сервера и клиента из листингов 5.1 и 5.2, но изменю код для сжатия большого файла PNG и отправки его с запросом HTTP. Далее сервер восстанавливает данные и сохраняет файл.

Код сервера приведен в листинге 6.5. Обратите внимание: передаваемые данные читаются в массив фрагментов, который затем используется для создания нового объекта `Buffer` вызовом `buffer.concat()`. Так как мы работаем с буфером, а не с потоком, использовать функцию `pipe()` не удастся. Вместо этого я буду использовать вспомогательную функцию Zlib `zlib.unzip`, которой передается объект `Buffer` и функция обратного вызова. Аргументы функции

обратного вызова содержат ошибку и результат. Результат также представляет собой объект `Buffer`, который записывается во вновь созданный поток для записи функцией `write()`. Чтобы программа создавала разные экземпляры файла, имя файла дополняется временной меткой.

Листинг 6.5. Создание веб-сервера, который получает сжатые данные и распаковывает их в файл

```
var http = require('http');
var zlib = require('zlib');
var fs = require('fs');

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
      var chunks = [];

      request.on('data', function(chunk) {
         chunks.push(chunk);
      });

      request.on('end', function() {
         var buf = Buffer.concat(chunks);
         zlib.unzip(buf, function(err, result) {
            if (err) {
               response.writeHead(500);
               response.end();
               return console.log('error ' + err);
            }
            var timestamp = Date.now();
            var filename = './done' + timestamp + '.png';
            fs.createWriteStream(filename).write(result);
         });
         response.writeHead(200, {'Content-Type': 'text/plain'});
         response.end('Received and undecompressed file\n');
      });
   }
});

console.log('server listening on 8214');
```

Ключевой момент клиентского кода в листинге 6.6 — назначение правильной кодировки `Content-Encoding` в заголовке. Заголовок должен содержать значение `'gzip,deflate'`. Также заголовок `Content-Type` меняется на `'application/javascript'`.

Листинг 6.6. Клиент сжимает файл и передает его с веб-запросом

```
var http = require('http');
var fs = require('fs');
var zlib = require('zlib');

var gzip = zlib.createGzip();

var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/javascript',
    'Content-Encoding': 'gzip,deflate'
  }
};

var req = http.request(options, function(res) {
  res.setEncoding('utf8');
  var data = '';
  res.on('data', function (chunk) {
      data+=chunk;
  });

  res.on('end', function() {
    console.log(data)
  })

});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// Потоковая передача сжатого файла серверу
var readable = fs.createReadStream('./test.png');
readable.pipe(gzip).pipe(req);
```

Клиент открывает файл для сжатия и передает его в поток преобразования данных, выполняющий сжатие Zlib; результат передается в веб-запрос (который представляет собой поток для записи). В коде мы работаем исключительно с потоками, что позволяет нам использовать функциональность `pipe()`, которая уже использовалась нами ранее. Использовать ее с сервером не удастся, потому что данные передаются в виде буферных фрагментов.

Буферизация файла в памяти может создать проблемы с масштабированием, поэтому возможно другое решение: сохранить несжатый файл, распаковать

его и затем удалить временный несжатый файл. Оставляю его читателям для самостоятельной работы.

Каналы и модуль ReadLine

Мы использовали каналы (pipes) в главе 5 и в этой главе. Один из простейших способов продемонстрировать использования каналов — откройте сеанс REPL и введите следующие команды:

```
> process.stdin.resume();
> process.stdin.pipe(process.stdout);
```

Все данные, которые вы будете вводить с этого момента, будут повторяться эхо-выводом.

Если вы хотите оставить выходной поток открытым для других данных, передайте параметр `{ end: false }` в выходной поток:

```
process.stdin.pipe(process.stdout, { end : false });
```

Построчная обработка данных в REPL на самом деле реализована с использованием последнего базового модуля, который будет рассмотрен в этой главе: `Readline`. Простое импортирование `Readline` также запускает бесконечный коммуникационный программный поток. Модуль `Readline` включается командой следующего вида:

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

Учтите, что при включении этого модуля программа Node не завершится, пока вы не закроете интерфейс.

В документации сайта Node представлен пример запуска и завершения интерфейса `Readline`; адаптированная версия этого кода приведена в листинге 6.7. Приложение задает вопрос сразу же после запуска, а затем выводит ответ. Оно также прослушивает «команды», которые в действительности представляют собой произвольные строки, завершающиеся символом `\n`. При вводе команды `.leave` приложение завершается; в противном случае приложение повторяет команду и запрашивает у пользователя новый ввод. Комбинации клавиш Ctrl+C и Ctrl+D тоже завершают приложение, хотя и менее корректно.

Листинг 6.7. Использование Readline для создания простого интерфейса, управляемого командами

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

// Создание нового интерфейса
var rl = readline.createInterface(process.stdin, process.stdout);

// Вопрос
rl.question(">>What is the meaning of life?  ", function(answer) {
   console.log("About the meaning of life, you said " + answer);
   rl.setPrompt(">> ");
   rl.prompt();
});

// Функция для закрытия интерфейса
function closeInterface() {
   rl.close();
   console.log('Leaving Readline');
}

// Прослушивание .leave
rl.on('line', function(cmd) {
   if (cmd.trim() == '.leave') {
      closeInterface();
      return;
   }
   console.log("repeating command: " + cmd);
   rl.prompt();
});

rl.on('close', function() {
    closeInterface();
});
```

Пример сеанса:

```
>>What is the meaning of life?  ===
About the meaning of life, you said ===
>>This could be a command
repeating command: This could be a command
>>We could add eval in here and actually run this thing
repeating command: We could add eval in here and actually run this thing
>>And now you know where REPL comes from
repeating command: And now you know where REPL comes from
>>And that using rlwrap replaces this Readline functionality
```

```
repeating command: And that using rlwrap replaces this Readline
                   functionality
>>Time to go
repeating command: Time to go
>>.leave
Leaving Readline...
Leaving Readline...
```

Что-то показалось вам знакомым? Вспомните, о чем говорилось в главе 4: программа `rlwrap` может использоваться для переопределения функциональности командной строки для REPL. Для активизации ее использования применяется следующая команда:

```
env NODE_NO_READLINE=1 rlwrap node
```

Теперь вы понимаете, что делает флаг, — он приказывает REPL не использовать модуль Node `Readline` для обработки командной строки и вместо него выбрать `rlwrap`.

7 Сети, сокеты и безопасность

Основа приложения Node неизменно зависит от двух главных компонентов инфраструктуры: сетевой поддержки и безопасности. А о сетях невозможно говорить без упоминания сокетов.

Я объединяю темы сетей и безопасности, потому что с выходом за пределы одной изолированной машины безопасность должна постоянно оставаться вашей первоочередной заботой. Каждый раз, когда вы завершаете работу над новым компонентом приложения, прежде всего следует спросить себя: насколько он безопасен? Никакое элегантное программирование не поможет, если в вашу систему проникнет вредоносный код.

Серверы, потоки и сокеты

Большая часть базовых API Node относится к созданию сервисов, прослушивающих конкретные типы коммуникаций. В главах 1 и 5 мы использовали модуль `HTTP` для создания веб-серверов, прослушивающих запросы HTTP. Другие модули могут создавать серверы TCP (Transmission Control Protocol), серверы TLS (Transport Layer Security) и сокеты UDP (User Datagram Protocol). О TLS я расскажу позднее в этой главе, а сейчас хочу познакомить вас с базовой функциональностью TCP и UDP в Node. Впрочем, сначала необходимо сказать пару слов о сокетах.

Сокеты и потоки

Сокет (socket) представляет собой конечную точку обмена данными, а *сетевой сокет* — конечную точку обмена данными между приложениями, работающими

на двух разных компьютерах в сети. Данные, передаваемые между сокетами, образуют *поток* (stream). Данные в потоке могут передаваться либо в виде двоичных данных в буфере, либо в виде строки в кодировке Юникод. Оба типа данных передаются в форме *пакетов*: частей данных, разделенных на блоки сходного размера. Также существует специальная разновидность пакетов — завершающий пакет (FIN), отправляемый сокетом как сигнал о завершении передачи.

Представьте двух людей, говорящих по портативной радиосвязи. Рации соответствуют конечным точкам передачи данных — сокетам. Когда два (или более) человека хотят поговорить друг с другом, они должны настроиться на одну частоту. Затем один участник нажимает кнопку на своей рации и соединяется с другим участником, используя некую форму идентификации. Каждый участник произносит слово «прием», сообщая, что он закончил говорить и теперь перешел к прослушиванию. Другой участник нажимает кнопку разговора на своей рации, произносит свою реплику и тоже говорит «прием», сообщая, что он переходит в режим прослушивания. Разговор продолжается до тех пор, пока одна из сторон не скажет «конец связи», сообщая о завершении сеанса. В любой момент времени говорить может только один человек.

Такие коммуникационные потоки называются *полудуплексными*, потому что передача в любой момент осуществляется только в одном направлении. *Дуплексные* потоки поддерживают двустороннюю передачу данных.

Эти концепции также относятся к потокам Node. Мы также работали с дуплексными и полудуплексными потоками в главе 6. Потоки, использованные для записи и чтения файлов, были полудуплексными: потоки поддерживали либо интерфейс чтения, либо интерфейс записи, но не оба сразу. Поток сжатия zlib является примером дуплексного потока, так как он поддерживает одновременно и чтение и запись. А теперь мы возьмем всю эту информацию и применим ее к сетевым (TCP) и зашифрованным (`Crypto`) потокам. Мы рассмотрим модуль `Crypto` позднее в этой главе, но сначала займемся TCP.

Серверы и сокеты TCP

Протокол TCP предоставляет коммуникационную платформу для большинства интернет-приложений, включая веб-службы и электронную почту. Он предоставляет механизм надежной передачи данных между клиентскими и серверными сокетами. TCP обеспечивает инфраструктуру, на которой строится прикладной уровень (например, протокол HTTP).

Клиент и сервер TCP создаются точно так же, как это делалось с HTTP, с небольшими различиями. При создании сервера TCP вместо передачи функции создания сервера `requestListener` с отдельными объектами ответа и запроса в единственном аргументе функции обратного вызова TCP передается экземпляр сокета, способного к отправке и получению данных.

Чтобы вы лучше поняли, как работает TCP, в листинге 7.1 представлен код создания сервера TCP. После создания серверный сокет прослушивает два события: получение данных и закрытие соединения клиентом. Затем полученные данные выводятся на консоль и отправляются обратно клиенту.

Сервер TCP также присоединяет обработчик для событий `listening` и `error`. В предыдущих главах я просто выводила сообщение `console.log()` после создания сервера, следуя сложившейся практике в Node. Но поскольку событие `listen()` асинхронно, с технической точки зрения эта практика неверна — сообщение выводится еще перед возникновением события. Вместо этого можно встроить сообщение в функцию обратного вызова функции `listen` или же сделать то, что я делаю здесь: связать обработчик с событием `listening` и правильно предоставить обратную связь.

Кроме того, я также реализую более сложную обработку ошибок по образцу, представленному в документации Node. Приложение обрабатывает событие `error`; если ошибка произошла из-за того, что порт в настоящее время используется, то приложение некоторое время ожидает, а потом пробует снова. Для других ошибок, например обращений к порту 80, требующему специальных привилегий, на консоль выводится полное сообщение об ошибке.

Листинг 7.1. Простой сервер TCP с сокетом, прослушивающим клиентскую передачу данных на порте 8124

```
var net = require('net');
const PORT = 8124;

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
```

```
    });

}).listen(PORT);

server.on('listening', function() {
    console.log('listening on ' + PORT);
});

server.on('error', function(err){
  if (err.code == 'EADDRINUSE') {
    console.warn('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT);
    }, 1000);
  }
  else {
    console.error(err);
  }
});
```

При создании сокета TCP можно передать дополнительный объект с параметрами, состоящий из двух значений: `pauseOnConnect` и `allowHalfOpen`. По умолчанию оба свойства имеют значение `false`:

```
{ allowHalfOpen: false,
pauseOnConnect: false }
```

Присваивание `allowHalfOpen` значения `true` запрещает сокету отправлять пакет FIN при получении пакета FIN от клиента. В этом случае сокет остается открытым для записи (не для чтения), и для закрытия сокета необходимо использовать функцию `end()`. Присваивание `pauseOnConnect` значения `true` позволяет принять соединение без чтения данных. Чтобы приступить к чтению данных, следует вызвать метод `resume()` для сокета.

Для тестирования сервера используйте клиентское приложение TCP, например утилиту `netcat (nc)` в Linux или OS X или эквивалентное приложение Windows. Следующая команда подключается к серверному приложению на порте 8124 и записывает данные из текстового файла на сервер:

```
nc burningbird.net 8124 < mydata.txt
```

В Windows для тестирования можно воспользоваться такими инструментами TCP, как SocketTest (*http://sockettest.sourceforge.net/*).

Вместо того чтобы использовать программу для тестирования сервера, вы можете создать собственное клиентское приложение TCP. Как видно из листинга 7.2, клиент TCP создается так же просто, как и сервер. Данные передаются в буфере, но мы можем использовать `setEncoding()` для чтения данных в виде строки `utf8`. Метод `write()` сокета используется для передачи данных. Клиентское приложение также назначает функции прослушивания для двух событий: `data` для получения данных и `close` на случай, если сервер закроет соединение.

Листинг 7.2. Клиентский сокет отправляет данные серверу TCP

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

// Подключение к серверу
client.connect ('8124','localhost', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// Полученные данные отправляются серверу
process.stdin.on('data', function (data) {
   client.write(data);
});

// Возвращаемые данные выводятся на консоль
client.on('data',function(data) {
   console.log(data);
});

// При закрытии сервера
client.on('close',function() {
   console.log('connection is closed');
});
```

Данные, передаваемые между двумя сокетами, вводятся в терминале и передаются при нажатии клавиши Enter. Клиентское приложение отправляет введенный текст, а сервер TCP выводит его на консоль при получении. Сервер возвращает полученное сообщение клиенту, который в свою очередь выводит сообщение на свою консоль. Сервер также выводит IP-адрес и порт клиента, используя свойства `remoteAddress` и `remotePort` сокета. Ниже приведен консольный вывод сервера после получения нескольких строк от клиента:

```
Hey, hey, hey, hey-now.
 from ::ffff:127.0.0.1 57251
Don't be mean, we don't have to be mean.
 from ::ffff:127.0.0.1 57251
Cuz remember, no matter where you go,
 from ::ffff:127.0.0.1 57251
there you are.
 from ::ffff:127.0.0.1 57251
```

Соединение между клиентом и сервером поддерживается до того момента, пока вы не уничтожите одного из участников клавишами Ctrl+C. Сокет, оставшийся открытым, получает событие `close`, сообщение о котором выводится на консоль. Сервер может обслуживать несколько соединений от нескольких клиентов, поскольку все соответствующие функции асинхронны.

ПРЕОБРАЗОВАНИЕ АДРЕСОВ IPV4 В АДРЕСА IPV6

Вывод, полученный при запуске пары приложений TCP в этом разделе, показывает, что адрес IPv4 преобразуется в адрес IPv6 добавлением ::ffff.

Вместо того чтобы связывать порт с сервером TCP, мы можем связать его напрямую с сокетом. Для демонстрации этой возможности я изменила сервер TCP из предыдущих примеров, но новый сервер связывается с *сокетом Unix* (листинг 7.3). Сокет Unix соответствует некоторому пути на вашем сервере. Для более точного управления доступом к сокету могут использоваться разрешения на чтение и запись, вследствие чего этот метод является более предпочтительным, чем интернет-сокеты.

Мне также пришлось внести изменения в обработку ошибок, чтобы сокет Unix удалялся при перезапуске приложения, если сокет уже используется. В реальном приложении перед выполнением столь радикальных действий следует убедиться в том, что сокет не используется другими клиентами.

Листинг 7.3. Связывание сервера TCP с сокетом Unix

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

const unixsocket = '/somepath/nodesocket';

var server = net.createServer(function(conn) {
   console.log('connected');
```

```
   conn.on('data', function (data) {
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
   });

}).listen(unixsocket);

server.on('listening', function() {
    console.log('listening on ' + unixsocket);
});

// При выходе с перезапуском сервера следует удалить сокет
server.on('error',function(err) {
   if (err.code == 'EADDRINUSE') {
      fs.unlink(unixsocket, function() {
          server.listen(unixsocket);
      });
   } else {
      console.log(err);
   }
});

process.on('uncaughtException', function (err) {
    console.log(err);
});
```

Я также использую `process` для дополнительной защиты от исключений, не обработанных приложением.

ПРОВЕРКА НАЛИЧИЯ ДРУГОГО ЭКЗЕМПЛЯРА СЕРВЕРА

Прежде чем удалять сокет, необходимо проверить, не выполняется ли другой экземпляр сервера. Решение, опубликованное на сайте Stack Overflow (*http://stackoverflow.com/questions/16178239/gracefully-shutdown-unix-socket-server-on-nodejs-running-under-forever*), представляет альтернативный способ освобождения ресурсов для подобных ситуаций.

Клиентское приложение приведено в листинге 7.4. Оно принципиально не отличается от более ранней версии клиента, работавшей с портом. Изменения относятся только к точке соединения.

Листинг 7.4. Подключение к сокету Unix и вывод полученных данных

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

// Подключение к серверу
client.connect ('/somepath/nodesocket', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// Полученные данные отправляются серверу
process.stdin.on('data', function (data) {
   client.write(data);
});

// Возвращаемые данные выводятся на консоль
client.on('data',function(data) {
   console.log(data);
});

// При закрытии сервера
client.on('close',function() {
   console.log('connection is closed');
});
```

В разделе «Защита передаваемых данных», с. 190, рассматривается HTTPS — SSL-версия HTTP, а также `Crypto` и TLS/SSL.

Сокет UDP

TCP требует выделенного соединения между двумя конечными точками. UDP — протокол, не требующий соединения; это означает, что соединение между двумя конечными точками не гарантировано. По этой причине протокол UDP по надежности и степени защиты от ошибок уступает TCP. С другой стороны, UDP обычно работает быстрее TCP, что делает его более популярным для задач реального времени и таких технологий, как VoIP (Voice over Internet Protocol), в которых требования к соединению TCP могут отрицательно повлиять на качество сигнала.

Базовая функциональность Node поддерживает оба типа сокетов. Функциональность TCP уже была продемонстрирована ранее, теперь очередь UDP.

Для модуля UDP используется идентификатор `dgram`:

```
require ('dgram');
```

Чтобы создать сокет UDP, вызовите метод `createSocket` и передайте ему тип сокета — `udp4` или `udp6`. Также можно передать методу функцию обратного вызова для прослушивания событий. В отличие от сообщений, отправляемых по протоколу TCP, сообщения UDP должны отправляться в виде буферов, но не в виде строк.

В листинге 7.5 приведен демонстрационный код клиента UDP. В нем клиент обращается к данным через `process.stdin` и отправляет их через сокет UDP. Указывать кодировку строки при этом необязательно, потому что сокет UDP принимает только буфер, а данные `process.stdin` представляют собой буфер. Однако нам приходится преобразовывать буфер в строку методом `toString()` буфера, чтобы получить осмысленную строку для эхо-вывода данных вызовом `console.log()`.

Листинг 7.5. Клиент UDP для отправки сообщений, вводимых с терминала

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

var client = dgram.createSocket("udp4");

process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
        if (err)
          console.error('error: ' + err);
        else
          console.log('successful');
   });
});
```

Сервер UDP, приведенный в листинге 7.6, еще проще клиента. По сути, он всего лишь создает сокет, связывает его с конкретным портом (8124) и прослушивает событие `message`. При поступлении сообщения приложение выводит его, а также IP-адрес и порт отправителя с использованием `console.log`. Для вывода сообщения кодировка не обязательна — оно автоматически преобразуется из буфера в строку.

Связывать сокет с портом не обязательно. Тем не менее без связывания сокет будет пытаться вести прослушивание на каждом порте.

Листинг 7.6. Сервер UDP для прослушивания сообщений, связанный с портом 8124

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

var server = dgram.createSocket("udp4");

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

server.bind(8124);
```

Ни в клиенте, ни в сервере после отправки/получения сообщения метод `close` не вызывается: между клиентом и сервером не поддерживается соединение — есть только сокеты, способные отправлять сообщения и получать данные.

Защита передаваемых данных

Безопасность в веб-приложениях не ограничивается простым предотвращением несанкционированного доступа к серверу приложения. Безопасность — сложная область, которая иногда наводит ужас на разработчиков. К счастью, в том, что касается разработки приложений Node, некоторые компоненты, необходимые для безопасности, уже были созданы за вас. Вам остается лишь подключить их в нужный момент.

Настройка TLS/SSL

Безопасный, защищенный от несанкционированного вмешательства обмен данными между клиентом и сервером осуществляется через протокол SSL (Secure Sockets Layer) и его обновленный вариант TLS (Transport Layer Security). Уровень TLS/SSL реализует шифрование данных для протокола HTTPS, который будет рассмотрен в следующем разделе. Но прежде чем браться за программирование для HTTPS, необходимо подготовить среду разработки.

Соединение TLS/SSL требует *согласования* (handshake) между клиентом и сервером. В процессе согласования клиент (обычно браузер) сообщает серверу, какие виды безопасности он поддерживает. Сервер выбирает функцию и отправляет *сертификат SSL*, включающий открытый ключ. Клиент подтверждает сертификат и генерирует случайное число с использованием ключа

сервера, отправляя его обратно серверу. Сервер использует свой закрытый ключ для расшифровки числа, которое в свою очередь используется для активизации безопасной передачи данных.

Чтобы эта схема работала, необходимо сгенерировать открытый и закрытый ключи, а также сертификат. В реальных условиях сертификат будет подписываться *доверенным центром сертификации* (например, регистратором доменных имен), но для разработки можно воспользоваться *самозаверяющим сертификатом*. При этом в браузере выводится предупреждение для всех пользователей приложения, но поскольку сайт в процессе разработки еще недоступен для пользователей, проблем с этим быть не должно.

ПОДАВЛЕНИЕ ПРЕДУПРЕЖДЕНИЙ О САМОЗАВЕРЯЮЩИХ СЕРТИФИКАТАХ

При использовании самозаверяющих сертификатов можно отключить предупреждения браузера, если вы обращаетесь к приложению Node через localhost (то есть https://localhost:8124). Также можно обойтись как без самозаверяющих сертификатов, так и без оплаты коммерческих центров сертификации; для этого можно воспользоваться сервисом Lets Encrypt (*https://letsencrypt.org/*), в настоящее время находящимся в фазе открытого бета-тестирования. На сайте имеется документация по созданию сертификатов (*https://letsencrypt.org/howitworks/*).

Необходимые файлы генерируются программой OpenSSL. Если вы работаете в Linux, программа должна быть уже установлена в системе; для Windows существует исполняемая двоичная версия, а Apple продвигает свою библиотеку `Crypto`. В этом разделе я рассматриваю только настройку в среде Linux.

Для начала введите в командной строке следующую команду:

```
openssl genrsa -des3 -out site.key 2048
```

Команда генерирует закрытый ключ, зашифрованный с применением алгоритма Triple-DES и хранящийся в формате PEM (Privacy-Enhanced Mail), что позволяет читать его в кодировке ASCII.

Вам будет предложено ввести пароль, который понадобится для следующей задачи — создания запроса на подпись сертификата (CSR, Certificate-Signing RequestCSR).

При генерировании CSR запрашивается только что созданный пароль. Вам также будет задано множество вопросов: код страны (например, US для Соединенных Штатов), штат или провинция, город, название компании или

организации, адрес электронной почты и т. д. Самым важным является вопрос об общем имени (Common Name). Введите имя хоста своего сайта, например *burningbird.net* или *yourcompany.com*. Укажите хост, на котором работает приложение. В своем примере я создала сертификат для хоста *examples.burningbird.net*.

```
openssl req -new -key site.key -out site.csr
```

Закрытому ключу необходима *парольная фраза* (passphrase). К сожалению, при каждом запуске сервера вам придется вводить эту парольную фразу. В среде реальной эксплуатации это создаст проблемы. На следующем шаге мы удалим парольную фразу из ключа. Сначала переименуйте ключ:

```
mv site.key site.key.org
```

Затем введите следующую команду:

```
openssl rsa -in site.key.org -out site.key
```

Если вы удалите парольную фразу, проследите за безопасностью вашего сервера и убедитесь в том, что файл доступен для чтения только для доверенных пользователей/групп.

Следующая задача — генерирование самозаверяющего сертификата. Следующая команда создает сертификат, действующий в течение 365 дней:

```
openssl x509 -req -days 365 -in site.csr -signkey site.key -out final.crt
```

Теперь у вас имеются все компоненты, необходимые для использования TLS/SSL и HTTPS.

Работа с HTTPS

Веб-страницы, на которых пользователь вводит учетные данные или информацию своей кредитной карты, должны предоставляться по протоколу HTTPS. В противном случае ваши данные будут передаваться в открытом виде и могут быть легко перехвачены. HTTPS — модификация протокола HTTP, объединенная с SSL; HTTPS обеспечивает проверку подлинности веб-сайта, шифрует данные в ходе передачи и проверяет, что отправленные данные были получены без промежуточного несанкционированного вмешательства.

Добавление поддержки HTTPS напоминает добавление поддержки HTTP с включением объекта `options`, содержащего открытый ключ шифрования

и подписанный сертификат. Отличается и порт по умолчанию для сервера HTTPS: протокол HTTP по умолчанию работает на порте 80, а HTTPS — на порте 443.

ПРЕДОТВРАЩЕНИЕ КОНФЛИКТА ПОРТОВ

У использования SSL в приложениях Node есть одно преимущество: по умолчанию HTTPS использует порт 443, а это означает, что вам не придется указывать номер порта при обращении к приложению и это не создаст конфликтов с Apache или другим веб-сервером (если, конечно, вы также не используете HTTPS с другим веб-сервером).

В листинге 7.7 представлен очень простой сервер HTTPS. В сущности, его возможности ограничиваются отправкой браузеру традиционного сообщения «Hello, World».

Листинг 7.7. Создание очень простого сервера HTTPS

```
var   fs = require("fs"),
      https = require("https");

var privateKey = fs.readFileSync('site.key').toString();
var certificate = fs.readFileSync('final.crt').toString();

var options = {
   key: privateKey,
   cert: certificate
};

https.createServer(options, function(req,res) {
   res.writeHead(200);
   res.end("Hello Secure World\n");
}).listen(443);
```

Программа открывает сертификат и открытый ключ и синхронно читает их содержимое. Данные присоединяются к объекту `options`, переданному в первом параметре метода `https.createServer`. Функция обратного вызова того же метода выглядит уже знакомо: запрос сервера и объект ответа передаются в параметрах.

Приложения Node должны запускаться с разрешениями суперпользователя. Дело в том, что сервер по умолчанию связывается с портом 443. Связывание с любым портом, номер которого меньше 1024, требует привилегий суперпользователя. Сервер также можно запустить на другом порте (например, 3000),

и все будет работать нормально, но при обращении к сайту придется указывать порт:

```
https://examples.burningbird.net:3000
```

Обращение к странице демонстрирует, что происходит при использовании самозаверяющего сертификата (рис. 7.1). Легко увидеть, почему самозаверяющий сертификат должен использоваться только во время тестирования. Обращение к странице с использованием `localhost` также блокирует предупреждение безопасности.

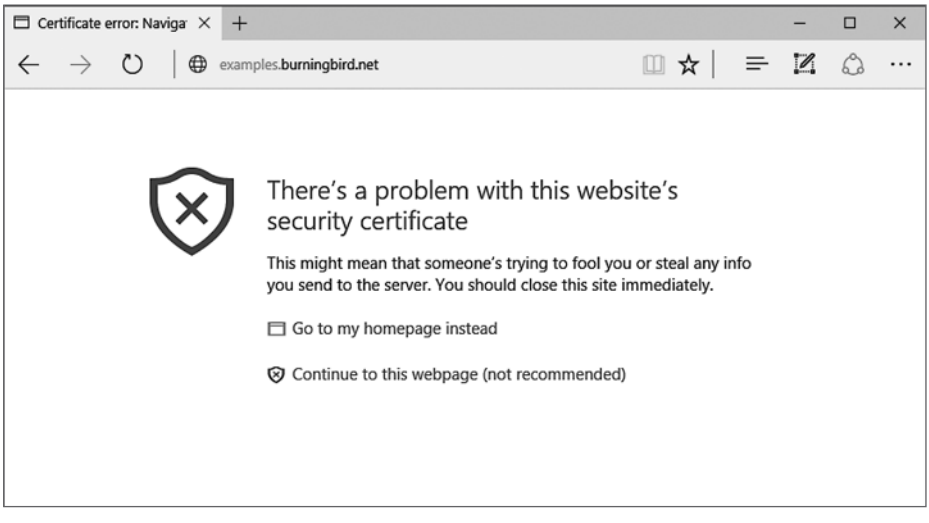


Рис. 7.1. Обращение к сайту с использованием HTTPS и самозаверяющего сертификата в Edge

Адресная строка браузера демонстрирует еще один способ, которым браузер сигнализирует о ненадежности сертификата сайта (рис. 7.2). Вместо замка (признак того, что браузер обращается к сайту через HTTPS) выводится замок с красным крестом — признак того, что сертификату нельзя доверять. Щелчок на замке открывает окно с более подробной информацией о сертификате.

Как уже говорилось ранее, использование доверенного центра сертификации для выдачи сертификата устраняет все эти неприятные предупреждения. А поскольку сейчас появилась возможность получения сертификата с нулевыми затратами, вы можете подумать о реализации HTTPS во всех своих приложениях с веб-интерфейсом, а не только в приложениях, работающих с платежами или паролями.

Рис. 7.2. Щелчок на значке с замком выводит более подробную информацию о сертификате (в браузере Chrome)

Модуль Crypto

Node предоставляет криптографический модуль `Crypto`, открывающий интерфейс к функциональности OpenSSL. В него включены обертки для функций хеширования OpenSSL, HMAC, шифрования, дешифрования, подписи и верификации. Этот компонент Node достаточно прост в использовании, но он основан на предположении, что разработчик Node хорошо знает OpenSSL и все используемые функции.

ЗНАКОМСТВО С OPENSSL

Трудно представить, насколько важно освоить азы OpenSSL перед тем, как браться за модуль Node Crypto. В вашем распоряжении документация OpenSSL (*https://www.openssl.org/docs/*), а также свободно распространяемая книга Айвена Ристича «OpenSSL Cookbook» (*https://www.feistyduck.com/books/openssl-cookbook/*).

Я рассматриваю относительно прямолинейный вариант использования модуля `Crypto` для создания хеша пароля с использованием функциональности хеширования OpenSSL. Та же функциональность может применяться для создания хеша, используемого как контрольная сумма (и гарантирующего, что данные не были повреждены в процессе сохранения или передачи).

MYSQL

В примере этого раздела используется MySQL. За дополнительной информацией о модуле node-mysql, используемом в этом разделе, обращайтесь к репозиторию модуля на сайте GitHub (*https://github.com/felixge/node-mysql*). Если у вас нет доступа к MySQL, сохраните имя пользователя, пароль и затравку в локальном файле и внесите соответствующие изменения в пример.

Метод `createHash` модуля `Crypto` может использоваться для создания хеша пароля, сохраняемого в базе данных. В следующем примере хеш создается с использованием алгоритма `sha1` и используется для кодирования пароля, после чего извлекается дайджест (digest) данных для сохранения в базе:

```
var hashpassword = crypto.createHash('sha1')
                                  .update(password)
                                  .digest('hex');
```

Для дайджеста назначается шестнадцатеричная кодировка. По умолчанию используется двоичная кодировка; также поддерживается кодировка `base64`.

ШИФРОВАНИЕ ПАРОЛЯ ИЛИ ХРАНЕНИЕ ХЕША

Хранение зашифрованных паролей лучше хранения паролей в виде простого текста, но пароль может быть взломан, если организация или некое лицо получит доступ к ключу шифрования. Хранение пароля в виде хеша — более безопасный, хотя и необратимый метод. Если владелец потеряет пароль, система предложит ему сбросить пароль (вместо того, чтобы пытаться его восстановить). За дополнительной информацией по этой теме обращайтесь к статье «Safely Storing User Passwords: Hashing vs. Encrypting» (*http://www.darkreading.com/safely-storing-user-passwords-hashing-vs-encrypting/a/d-id/1269374*).

Многие приложения применяют для этой цели хеширование. Однако при хранении простых хешированных паролей в базе данных возникает проблема, известная под красивым названием «радужная таблица».

Проще говоря, *радужная таблица* (rainbow table) представляет собой таблицу заранее вычисленных значений хеша для всех возможных комбинаций символов. Таким образом, даже если у вас имеется пароль, который, как вы уверены, взломать невозможно (будем откровенны — такая уверенность есть далеко не всегда), может оказаться, что такая последовательность символов встречается где-то в радужной таблице; это заметно упростит определение пароля.

Проблема радужной таблицы обходится с помощью *затравки* (salt) — уникального сгенерированного значения, которое присоединяется к паролю перед шифрованием. Это может быть одно значение, которое используется со всеми паролями и безопасно хранится на сервере. Впрочем, лучше генерировать уникальную затравку для каждого пароля и сохранять ее с паролем. Да, затравка может быть похищена вместе с паролем, но тогда злоумышленнику, пытающемуся взломать пароль, все равно придется генерировать радужную таблицу всего для одного пароля, что невероятно повышает сложность взлома любого конкретного пароля.

В листинге 7.8 приведено простое приложение, которое получает имя пользователя и пароль в аргументах командной строки, генерирует хеш пароля, после чего сохраняет оба значения в таблице базы данных MySQL. Я использую MySQL вместо любой другой базы данных, потому что MySQL присутствует в большинстве систем и эта база данных знакома наибольшему количеству пользователей.

Чтобы воспроизвести этот пример на своем компьютере, установите модуль `node-mysql` следующей командой:

```
npm install node-mysql
```

Таблица создается следующей командой SQL:

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT, PRIMARY KEY(userid),
username VARCHAR(400) NOT NULL, passwordhash VARCHAR(400) NOT NULL,
salt DOUBLE NOT NULL );
```

Затравка состоит из значения даты, умноженного на случайное число и округленного. Она присоединяется к паролю перед вычислением хеша пароля. Все данные пользователя вставляются в таблицу MySQL.

Листинг 7.8. Использование метода createHash и затравки для шифрования пароля

```
var mysql = require('mysql'),
    crypto = require('crypto');

var connection = mysql.createConnection({
   host: 'localhost',
   user: 'username',
   password: 'userpass'
  });

connection.connect();
```

```
connection.query('USE nodedatabase');

var username = process.argv[2];
var password = process.argv[3];

var salt = Math.round((Date.now() * Math.random())) + '';

var hashpassword = crypto.createHash('sha512')
                   .update(salt + password, 'utf8')
                   .digest('hex');
// Создание записи пользователя
connection.query('INSERT INTO user ' +
   'SET username = ?, passwordhash = ?, salt = ?
   [username, hashpassword, salt], function(err, result) {
      if (err) console.error(err);
      connection.end();
});
```

Сначала создается соединение с базой данных с последующим выбором вновь созданной таблицы. Имя пользователя и пароль извлекаются из командной строки, после чего начинается криптографическое «волшебство».

Программа генерирует затравку и передает ее функции для создания хеша с применением алгоритма `sha512`. Функции обновления хеша затравкой и выбора кодировки хеша объединяются в цепочку с функцией создания хеша. Только что зашифрованный хеш пароля вставляется в только что созданную таблицу вместе с именем пользователя.

Приложение для проверки имени пользователя и пароля, приведенное в листинге 7.9, запрашивает из базы данных хеш пароля и затравку для заданного имени пользователя. Затравка используется для повторного генерирования хеша. Заново сгенерированный хеш пароля сравнивается с паролем, хранящимся в базе данных. Если значения не совпадают, пользователь не проходит проверку. В случае совпадения пользователь допускается к работе с системой.

Листинг 7.9. Проверка имени пользователя и пароля

```
var mysql = require('mysql'),
    crypto = require('crypto');

var connection = mysql.createConnection({
   user: 'username',
   password: 'userpass'
  });

connection.query('USE nodedatabase');
```

```
var username = process.argv[2];
var password = process.argv[3];

connection.query('SELECT password, salt FROM user WHERE username = ?',
   [username], function(err, result, fields) {
   if (err) return console.error(err);

   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password, 'utf8')
                 .digest('hex');

   if (result[0].password === newhash) {
      console.log("OK, you're cool");
   } else {
      console.log("Your password is wrong. Try again.");
   }
   connection.end();
});
```

В качестве эксперимента мы сначала передадим имя `Michael` с паролем `apple*frk13*`:

```
node password.js Michael apple*frk13*
```

При последующей проверке имени пользователя и пароля:

```
node check.js Michael apple*frk13*
```

будет получен ожидаемый результат:

```
OK, you're cool
```

Повторим проверку с другим паролем:

```
node check.js Michael badstuff
```

И снова результат соответствует ожиданиям:

```
Your password is wrong. Try again
```

Криптографическое хеширование также может применяться к потоку. Для примера возьмем контрольную сумму — алгоритмический способ проверки правильности передачи данных. Вы создаете хеш файла и передаете его вместе с файлом при отправке. Получатель файла использует хеш для проверки правильности передачи. В следующем коде для создания такого хеша используется функция `pipe()` и дуплексная природа функций `Crypto`.

```
var crypto = require('crypto');
var fs = require('fs');
var hash = crypto.createHash('sha256');
hash.setEncoding('hex');

var input = fs.createReadStream('main.txt');
var output = fs.createWriteStream('mainhash.txt');

input.pipe(hash).pipe(output);
```

Также можно выбрать алгоритм `md5` для генерирования контрольной суммы MD5; этот алгоритм пользуется популярностью во многих средах и приложениях из-за своей быстроты (хотя при этом обеспечивает меньшую защищенность).

```
var hash = crypto.createHash('md5');
```

8 Дочерние процессы

Операционные системы предоставляют доступ к разнообразным функциям, большая часть которых доступна только в режиме командной строки. Было бы удобно иметь возможность обращаться к этим функциям из приложений Node. В этом вам помогут *дочерние процессы*.

Node позволяет выполнить системную команду в дочернем процессе и прослушать ее ввод/вывод. Команде можно передать аргументы и даже направить результаты одной команды в другую. В нескольких следующих разделах эта функциональность будет рассмотрена более подробно.

Во всех примерах этой главы, кроме последнего, используются команды Unix. Они работают в системе Linux и должны работать в OS X. Тем не менее в режиме командной строки Windows они работать не будут.

child_process.spawn

Существует четыре разных способа создания дочерних процессов. В самом распространенном используется метод `spawn`. Он запускает команду в новом процессе, передавая ей любые аргументы. Между родительским приложением и дочерним процессом создаются каналы (pipes) для `stdin`, `stdout` и `stderr`.

В следующем примере создается дочерний процесс для вызова команды Unix `pwd` для вывода текущего каталога. Команда вызывается без аргументов.

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd');

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

pwd.stderr.on('data', function (data) {
  console.error('stderr: ' + data);
});

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

Обратите внимание на события, обрабатываемые для потоков `stdout` и `stderr` дочернего процесса. При отсутствии ошибок весь вывод команды передается в `stdout` дочернего процесса, что приводит к генерированию события `data`. Если происходит ошибка, — как в следующем примере, в котором команде передается недействительный аргумент:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd', ['-g']);
```

то ошибка передается в поток `stderr`, выводящий информацию об ошибке на консоль:

```
stderr: pwd: invalid option -- 'g'
Try `pwd --help' for more information.

child process exited with code 1
```

Процесс завершается с кодом 1, обозначающим возникновение ошибки. Код завершения зависит от операционной системы и ошибки. При отсутствии ошибок дочерний процесс завершается с кодом 0.

Приведенный пример демонстрирует передачу вывода в потоки `stdout` и `stderr` дочернего процесса, но как насчет `stdin`? В документации Node, посвященной дочерним процессам, приведен пример отправки данных в `stdin`. Он используется для эмуляции каналов Unix (`|`), немедленно передающих результат выполнения одной команды на ввод другой команды. Я адаптировала пример для демонстрации одной возможности каналов Unix — поиска во всех подкаталогах, начиная с текущего каталога, файла, имя которого содержит заданное слово (в данном случае `test`):

```
find . -ls | grep test
```

В листинге 8.1 эта функциональность реализована в дочерних процессах. Обратите внимание: первая команда, выполняющая `find`, получает два аргумента, тогда как вторая получает всего один: условие поиска, переданное в пользовательском вводе из `stdin`. Также обратите внимание на смену кодировки `stdout` дочернего процесса `grep` вызовом `setEncoding`. Без этого данные будут выведены как содержимое буфера.

Листинг 8.1. Использование дочерних процессов для поиска в подкаталогах файлов, имена которых содержат слово «test»

```
var spawn = require('child_process').spawn,
    find = spawn('find',['.','-ls']),
    grep = spawn('grep',['test']);

grep.stdout.setEncoding('utf8');

// Вывод find направляется на ввод grep
find.stdout.pipe(grep.stdin);

// Выполнение grep с выводом результатов
grep.stdout.on('data', function (data) {
  console.log(data);
});

// Обработка ошибок для обеих команд
find.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

// Обработка завершения для обеих команд
find.on('close', function (code) {
  if (code !== 0) {
    console.log('find process exited with code ' + code);
  }
});

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

Запустив приложение, вы получите список всех файлов в текущем каталоге и во всех подкаталогах, имена которых содержат слово `test`.

Документация Node содержит предупреждение о том, что некоторые программы в своей внутренней работе используют ввод/вывод с построчной буферизацией. Это может привести к тому, что данные, отправленные программе, не будут потребляться немедленно. Для нас это означает то, что с некоторыми дочерними процессами данные буферизируются в блоках перед обработкой. Например, к их числу относится дочерний процесс `grep`.

В текущем примере размер вывода процесса `find` ограничен, поэтому ввод процесса `grep` не превышает размер блока (обычно 4096, но он может зависеть от конфигурации системы и индивидуальных настроек).

ПОДРОБНЕЕ О БУФЕРИЗАЦИИ STDIO

Более подробную информацию о буферизации можно найти в статьях «How to fix stdio buffering» (*https://www.perkin.org.uk/posts/how-to-fix-stdio-buffering.html*) и «Buffering in standard streams» (*http://www.pixelbeat.org/programming/stdio_buffering/*).

Построчную буферизацию для `grep` можно отключить параметром командной строки `--line-buffered`. В следующем приложении, использующем команду `ps` (Process Status) для анализа выполняемых процессов и последующего поиска экземпляров `apache2`, построчная буферизация отключается для `grep` и данные выводятся немедленно, а не при заполнении буфера:

```
var spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['--line-buffered', 'apache2']);

ps.stdout.pipe(grep.stdin);

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.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);
  }
});
```

Теперь вывод не буферизируется, а данные выводятся немедленно.

По умолчанию команда `child_process.spawn()` не выполняет команду в командном интерпретаторе. Однако начиная с Node 5.7.0 вы можете добавить параметр `shell`, и дочерний процесс создаст командный интерпретатор для процесса. В разделе «Выполнение приложения в дочернем процессе в Windows», с. 210, я покажу, как работает этот механизм в приложении Node, предназначенном для работы в Windows. Также поддерживаются другие параметры, включая следующие (список не полон; за подробным и актуальным списком обращайтесь к документации Node):

- `cwd`: смена рабочего каталога;
- `env`: массив пар «ключ/значение» с переменными среды;
- `detached`: подготовка дочернего процесса к выполнению отдельно от родителя;
- `stdio`: массив параметров `stdio` дочернего процесса.

У параметра `detached`, как и у `shell`, в среде Windows есть интересные отличия от других сред. Этот параметр тоже будет рассмотрен в разделе «Выполнение приложения в дочернем процессе в Windows», с. 210.

Функция `child_process.spawnSync()` — синхронная версия той же функции.

child_process.exec и child_process.execFile

Кроме порождения дочернего процесса функцией `child_process.spawn()`, для выполнения команды также можно воспользоваться функциями `child_process.exec()` и `child_process.execFile()`.

Метод `child_process.exec()` похож на `child_process.spawn()`, не считая того, что `spawn()` начинает возвращать поток сразу же после запуска программы, как видно из листинга 8.1. Функция `child_process.exec()`, как и `child_process.`

`execFile()`, буферизирует результаты. Однако `exec()` порождает командный интерпретатор для управления приложением, тогда как `child_process.execFile()` запускает процесс напрямую. Это делает функцию `child_process.execFile()` более эффективной по сравнению как с `child_process.spawn()` с включенным режимом `shell`, так и с `child_process.exec()`.

В первом параметре `child_process.exec()` или `child_process.execFile()` передается либо команда (для `exec()`), либо файл и его местоположение (`execFile()`); второй параметр содержит объект `options` для команды; в третьем параметре передается функция обратного вызова. Функция обратного вызова получает три аргумента: `error`, `stdout` и `stderr`. При отсутствии ошибок данные буферизируются в `stdout`.

Если исполняемый файл содержит строку

```
#!/usr/bin/node

console.log(global);
```

следующее приложение выводит буферизованные результаты:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./app', function(error, stdout, stderr) {
  if (error == null) {
    console.log('stdout: ' + stdout);
  }
});
```

То же самое можно сделать с использованием `child_process.exec()`:

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

child = exec('./app', function(error, stdout, stderr) {
   if (error) return console.error(error);
   console.log('stdout: ' + stdout);

});
```

Различие в том, что `child_process.exec()` порождает командный интерпретатор, тогда как `child_process.execFile()` этого не делает.

Функция `child_process.exec()` получает три параметра: команду, объект `options` и функцию обратного вызова. Объект `options` содержит несколько

значений, включая кодировку (`encoding`), идентификатор пользователя (`uid`) и идентификатор группы (`gid`) процесса. В главе 6 я создала приложение, которое копирует файл PNG и добавляет эффект Polaroid. Дочерний процесс использовался для обращения к ImageMagick — мощной программе обработки графики, работающей в режиме командной строки. Чтобы запустить его вызовом `child_process.exec()`, используйте следующий фрагмент с передачей аргумента командной строки:

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

child = exec('./polaroid -s phoenix5a.png -f phoenixpolaroid.png',
        {cwd: 'snaps'}, function(error, stdout, stderr) {
               if (error) return console.error(error);
               console.log('stdout: ' + stdout);
});
```

Функция `child_process.execFile()` получает дополнительный параметр: массив параметров командной строки, передаваемых приложению. Эквивалентное приложение с использованием этой функции выглядит так:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./snapshot',
                ['-s', 'phoenix5a.png', '-f', 'phoenixpolaroid.png'],
                 {cwd: 'snaps'}, function(error, stdout, stderr) {
   if (error) return console.error(error);
   console.log('stdout: ' + stdout);
});
```

Обратите внимание: аргументы командной строки разбиваются по разным элементам массива, при этом за каждым аргументом следует его значение.

Так как функция `child_process.execFile()` не порождает командный интерпретатор, она не применима в некоторых обстоятельствах. В документации Node указано, что вы не можете использовать перенаправление ввода/вывода и шаблоны имен файлов (с регулярными выражениями или метасимволами). Но если вы пытаетесь запустить дочерний процесс (или приложение) в интерактивном режиме, используйте `child_process.execFile()` вместо `child_process.exec()`. Это отлично демонстрирует следующий код, написанный членом Node Foundation Колином Иригом:

```
'use strict';
const cp = require('child_process');
```

```
const child = cp.execFile('node', ['-i'], (err, stdout, stderr) => {
  console.log(stdout);
});

child.stdin.write('process.versions;\n');
child.stdin.end();
```

Приложение запускает интерактивный сеанс Node, запрашивает версии процесса, после чего завершает ввод.

У обеих функций существуют синхронные версии — `child_process.execSync()` и `child_process.execFileSync()`.

child_process.fork

Последний метод дочерних процессов — `child_process.fork()`. Эта разновидность `spawn()` предназначена для порождения процессов Node.

Вызов `child_process.fork()` отличается от других тем, что он создает реальный канал передачи данных дочернему процессу. Однако обратите внимание на то, что каждому процессу требуется новый экземпляр V8, а это приводит к дополнительным затратам времени и памяти.

Метод `child_process.fork()` используется для выделения функциональности в полностью изолированные экземпляры Node. Допустим, у вас имеется сервер в одном экземпляре Node и вы хотите улучшить производительность, интегрировав второй экземпляр Node для ответа на запросы сервера. В документации Node приведен такой пример с сервером TCP. Может ли этот метод использоваться для создания параллельных серверов HTTP? Да, притом по той же схеме.

Я хочу поблагодарить Джайли Ху за идею, которая пришла мне в голову, когда я увидела его демонстрационный пример с запуском параллельных серверов HTTP в отдельных экземплярах. Джайли использует сервер TCP для передачи конечных точек сокетов в два отдельных дочерних сервера HTTP.

По аналогии с примером, приведенным в документации Node для параллельных серверов TCP в схеме «ведущий/ведомый», я создаю сервер HTTP, после чего использую функцию `child_process.send()` для отправки сервера дочернему процессу.

```
var cp = require('child_process'),
    cp1 = cp.fork('child2.js'),
    http = require('http');
var server = http.createServer();

server.on('request', function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by parent\n');
});

server.on('listening', function () {
    cp1.send('server', server);
});

server.listen(3000);
```

Дочерний процесс получает сообщение с сервером HTTP в объекте `process`. Он прослушивает событие соединения `connection`, и при получении инициирует событие `connection` для дочернего сервера HTTP, передавая ему сокет, становящийся конечной точкой соединения:

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

var server = http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by child\n');
});

process.on('message', function (msg, httpServer) {
   if (msg === 'server') {
      httpServer.on('connection', function (socket) {
          server.emit('connection', socket);
      });
   }
});
```

Если вы протестируете приложение, обратившись к домену с портом 3000, то увидите, что запрос иногда обрабатывается родительским сервером HTTP, а иногда дочерним. Проверка работающих процессов тоже обнаруживает два процесса: родительский и дочерний.

NODE CLUSTER

Модуль Node Cluster базируется на child_process.fork(), помимо другой функциональности.

Выполнение приложения в дочернем процессе в Windows

Ранее я предупреждала вас о том, что дочерние процессы, выполняющие системные команды Unix, не работают в Windows, и наоборот. Конечно, это звучит тривиально, но не все знают, что, в отличие от JavaScript в браузерах, приложения Node могут по-разному работать в разных средах.

Помимо различий в операционных системах и командах, при работе в Windows необходимо использовать либо метод `child_process.exec()`, запускающий командный интерпретатор для запуска приложения, либо параметр `shell` с новыми версиями `child_process.spawn()`. В противном случае необходимую команду следует выполнить через командный интерпретатор Windows cmd.exe.

Пример использования параметра `shell` с `child_process.spawn()`: следующее приложение выводит содержимое каталога на машине с системой Windows:

```
var spawn = require('child_process').spawn,
    pwd = spawn('echo', ['%cd%'], {shell: true});

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

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

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

Команда `echo` повторяет результат команды Windows `cd`, которая выводит содержимое текущего каталога. Если бы параметру `shell` не было присвоено значение `true`, произошла бы ошибка.

Аналогичного результата можно добиться вызовом `child_process.exec()`. Однако учтите, что с `child_process.exec()` команду `echo` использовать необязательно, так как эта функция буферизирует вывод:

```
var exec = require('child_process').exec,
    pwd = exec('cd');
```

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

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

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

Листинг 8.2 демонстрирует третий вариант: выполнение команды Windows через выполнение команды `cmd` (для приложения Windows cmd.exe); далее в аргументе следует то, что непосредственно выполняется в командном интерпретаторе. В приложении команда Windows cmd.exe используется для создания списка содержимого каталога, который затем выводится на консоль с использованием обработчика события `data`.

Листинг 8.2. Выполнение приложения в дочернем процессе в Windows

```
var cmd = require('child_process').spawn('cmd', ['/c', 'dir\n']);

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

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

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

Флаг `/c`, переданный в первом аргументе cmd.exe, приказывает выполнить команду и завершиться. Без этого флага приложение работать не будет. Особенно нежелателен флаг `/K`, который приказывает cmd.exe выполнить приложение и продолжить работу, потому что в этом случае ваше приложение не завершится.

Эквивалент с `child_process.exec()` выглядит так:

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

Для запуска cmd или bat-файла можно воспользоваться child_process.execFile(), как и при запуске файла в среде семейства Unix. Допустим, имеется следующий bat-файл (my.bat):

```
@echo off
REM Next command generates a list of program files
dir
```

Для его запуска можно воспользоваться следующим приложением:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('my.bat', function(error, stdout, stderr) {
  if (error == null) {
    console.log('stdout: ' + stdout);
  }
});
```

9 Node и ES6

В большинстве примеров, приведенных в книге, используется код JavaScript, с которым мы имеем дело уже много лет. И это абсолютно нормальный код, хорошо знакомый всем, кто работал на этом языке в среде браузера. Однако одно из преимуществ разработки в среде Node заключается в том, что вы можете использовать более современный вариант JavaScript, такой как ECMAScript 2015 (или ES6, как его часто называют), и даже более поздние версии, не беспокоясь о совместимости с браузером или операционной системой. Многие новые расширения языка стали неотъемлемой частью функциональности Node. В этой главе будут рассмотрены некоторые новые возможности JavaScript, которые по умолчанию реализуются версиями Node, упомянутыми в книге. Вы увидите, как они улучшают приложения Node; также будут рассмотрены нюансы, о которых необходимо знать при использовании новой функциональности.

СПИСОК ПОДДЕРЖИВАЕМЫХ ФУНКЦИЙ ES6

Я не буду рассматривать всю функциональность ES6, поддерживаемую в Node, — только текущие возможности, которые я часто вижу в приложениях Node, модулях и примерах. За полным списком реализованных возможностей ES6 обращайтесь к документации Node (*https://nodejs.org/en/docs/es6/*).

Строгий режим

Строгий (strict) режим JavaScript существует со времен ECMAScript 5, но его использование напрямую влияет на использование функциональности ES6, поэтому этот режим стоит рассмотреть поближе, прежде чем браться за описание возможностей ES6.

Строгий режим включается при добавлении следующей директивы в начало приложения Node:

```
"use strict";
```

Допускается использование как одиночных, так и двойных кавычек.

Также существуют другие способы включения строгого режима во всех зависимых модулях приложения, например флаг `--strict_mode`, но я ими пользоваться не рекомендую. Включение строгого режима для модуля с большой вероятностью приведет к появлению ошибок или непредвиденных последствий. Просто используйте его в своем приложении или модулях там, где код находится под вашим контролем.

Строгий режим сильно влияет на работу кода. В частности, он выдает ошибки, если переменная не определяется перед использованием; параметр функции может объявляться только единожды; переменная не может использоваться в `eval`-выражении на одном уровне с вызовом `eval`, и т. д. Но в этом разделе я хочу особо остановиться на запрете на использование восьмеричных литералов в строгом режиме.

В предыдущих главах при назначении разрешений доступа к файлу восьмеричная запись становится невозможной:

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0666, function(err, fd) {
     if (err) return console.error(err);
     fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
        if (err) return console.error(err);
        var buf = new Buffer(written);
        fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
           if (err) return console.error(err);
           console.log(buf.toString('utf8'));
        });
      });
});
```

В строгом режиме этот код приводит к синтаксической ошибке:

```
fs.open('./new.txt','a+',0666, function(err, fd) {
                          ^^^^
SyntaxError: Octal literals are not allowed in strict mode.
```

Восьмеричный литерал можно преобразовать в безопасный формат, заменив начальный нуль на `0o` — нуль, за которым следует буква `o` в нижнем регистре. Приложение в строгом режиме работает, если для назначения разрешений используется запись `0o666` вместо `0666`:

```
fs.open('./new.txt','a+',0o666, function(err, fd) {
```

Функция `fs.open()` также может получать восьмеричное значение в виде строки:

```
fs.open('./new.txt','a+','0666', function(err, fd) {
```

Впрочем, этот синтаксис многие не одобряют, поэтому лучше использовать предыдущий формат.

Строгий режим также обязателен для использования некоторых расширений ES6. Если вы хотите использовать классы ES6 (см. далее в этой главе), для этого необходимо включить строгий режим. Если вы хотите использовать `let` (см. далее), вам также понадобится строгий режим.

ВОСЬМЕРИЧНЫЕ ЛИТЕРАЛЫ

Если вас интересуют корни истории с преобразованием восьмеричных литералов, я рекомендую ознакомиться с обсуждением на сайте ES Discuss, «Octal literals have their uses (you Unix haters skip this one)» (*https://esdiscuss.org/topic/octal-literals-have-their-uses-you-unix-haters-skip-this-one*).

let и const

В прошлом в приложениях JavaScript действовало ограничение, которое делало невозможным объявление переменных на блочном уровне. Одним из самых долгожданных новшеств ES6 стала команда `let`. С ее помощью можно объявить внутри блока переменную, область видимости которой ограничивается этим блоком. В следующем примере при использовании `var` выводится значение 100:

```
if (true) {
  var sum = 100;
}

console.log(sum); // Выводит 100
```

Если же использовать `let`:

```
"use strict";

if (true) {
  let sum = 100;
}

console.log(sum);
```

то результат будет совершенно иным:

```
ReferenceError: sum is not defined
```

Для использования `let` приложение должно работать в строгом режиме.

Кроме области видимости блочного уровня `let` отличается от `var` тем, что переменные, объявленные с `var`, поднимаются в начало области видимости до выполнения каких-либо команд. Следующий фрагмент приводит к тому, что на консоль выводится `undefined`, но ошибки времени выполнения не происходит:

```
console.log(test);
var test;
```

С другой стороны, следующий код с `let` приводит к появлению ошибки `ReferenceError` с сообщением о том, что значение `test` не определено:

```
"use strict";

console.log(test);
let test;
```

Так стоит ли всегда использовать `let`? Одни программисты отвечают «да»; другие отвечают «нет». Вы также можете использовать обе конструкции и ограничить использование `var` переменными, которым необходима область видимости уровня приложения или функции, и применять `let` только для блочного уровня видимости. Впрочем, с большой вероятностью правила оформления кода, принятые в вашей организации, определят используемую конструкцию, или, если вы уверены в том, что в вашей среде не будет проблем, — используйте `let` и переходите на сторону ES6.

Команда `const` объявляет ссылку на значение, доступную только для чтения. Если значение относится к примитивному типу, оно неизменно. Если значение является объектом, вы не сможете присвоить по ссылке новый объект или примитив, но сможете изменить свойства объекта.

В следующем фрагменте при попытке присвоить новое значение по const-ссылке операция присваивания игнорируется без выдачи сообщений об ошибке:

```
const MY_VAL = 10;
MY_VAL = 100;
console.log(MY_VAL); // Выводится 10
```

Если присвоить по const-ссылке массив или объект, вы сможете изменять элементы объекта/массива:

```
const test = ['one','two','three'];

const test2 = {apples : 1, peaches: 2};

test = test2; //Не проходит

test[0] = test2.peaches;

test2.apples = test[2];

console.log(test); // [ 2, 'two', 'three' ]
console.log(test2); // { apples: 'three', peaches: 2 }
```

К сожалению, использование const порождает немало путаницы из-за различий в поведении между примитивными и объектными значениями, а также из-за самого названия, которое вроде бы подразумевает константное (статическое) присваивание. Впрочем, если вашей конечной целью является неизменность и по const-ссылке присваивается объект, вы можете вызвать object.freeze() для объекта, чтобы обеспечить по крайней мере поверхностную (shallow) неизменность.

Я заметила, что в документации Node представлено применение const при импортировании модулей. Хотя присваивание объекта по const-ссылке не предотвращает возможного изменения его свойств, оно формирует семантический уровень, на котором другой программист сразу же видит, что элементу позднее не будет присвоено новое значение.

ПОДРОБНЕЕ О CONST И НЕИЗМЕННОСТИ

У Матиаса Байненса, известного сторонника веб-стандартов, имеется хорошее обсуждение const и неизменности (*https://mathiasbynens.be/notes/es6-const*).

Как и let, const также обладает областью видимости блочного уровня. Однако, в отличие от let, const не требует обязательного включения строгого режима.

Вы можете использовать `let` и `const` в своих приложениях по тем же причинам, по каким используете их в коде браузера, однако я не нашла никаких дополнительных преимуществ, специфических для Node. Некоторые разработчики сообщают о повышении быстродействия при использовании `let` и `const`, другие отмечают снижение быстродействия. Лично я никаких изменений не заметила, но в вашей ситуации результат может быть другим. Как упоминалось ранее, в вашей группе разработки с большой вероятностью существуют требования по поводу того, когда использовать ту или иную конструкцию; придерживайтесь этих требований.

Стрелочные функции

Наиболее часто используемым усовершенствованием ES6 являются *стрелочные функции* (arrow functions); чтобы убедиться в этом, достаточно заглянуть в документацию Node API. Стрелочные функции решают две задачи. Во-первых, они упрощают синтаксис. Например, в предыдущих главах я использовала следующий фрагмент для создания сервера HTTP:

```
http.createServer(function (req, res) {
   res.writeHead(200);
   res.write('Hello');
   res.end();
}).listen(8124);
```

Со стрелочной функцией этот фрагмент можно переписать в следующем виде:

```
http.createServer( (req, res) => {
   res.writeHead(200);
   res.write('Hello');
   res.end();
}).listen(8124);
```

Ключевое слово `function` исчезло, а для обозначения анонимной функции с заданными параметрами используется стрелка `=>`. Упрощение можно продолжить; например, хорошо знакомый паттерн функции:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(function(element) {
   return element.toString(16);
});
console.log(hexArray); // ["17", "ff", "7a", "5", "10", "63"]
```

упрощается до следующего вида:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(element => element.toString(16));

console.log(hexArray); // ["17", "ff", "7a", "5", "10", "63"]
```

Фигурные скобки, команда `return`, ключевое слово `function` — все это исчезает, а функциональность урезается до минимума.

Стрелочные функции не ограничиваются упрощением синтаксиса; они также могут переопределять `this`. В JavaScript до появления стрелочных функций каждая функция определяла собственное значение `this`. Таким образом, в следующем примере вместо имени на консоль выводится `undefined`:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(function() {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

Дело в том, что в конструкторе объекта `this` определяется как объект, а в последующем экземпляре — как функция `setTimeout`. Проблему можно обойти использованием другой переменной (обычно `self`), которая может быть привязана к заданному окружению. Следующий фрагмент работает так, как ожидается, — он выводит имя:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   var self = this;
   setTimeout(function() {
      console.log(self.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

В стрелочной функции `this` всегда присваивается значение, которое переменная обычно имела бы в окружающем контексте, — в данном случае `new`-объект:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(()=> {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

СТРАННОСТИ СТРЕЛОЧНЫХ ФУНКЦИЙ

У стрелочных функций есть свои странности, например может быть непонятно, как вернуть пустой объект и где их аргументы? На сайте StrongLoop есть хорошая статья о стрелочных функциях (*https://strongloop.com/strongblog/an-introduction-to-javascript-es6-arrow-functions/*), в которой обсуждаются эти странности и обходные решения.

Классы

Язык JavaScript присоединился к своим «старшим товарищам»: в нем теперь тоже поддерживаются классы. Забудьте о всевозможных «творческих» способах моделирования знакомого поведения классов.

Ранее, в главе 3, я создала «класс» `InputChecker` с использованием старого синтаксиса:

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
                     {'flags' : 'a',
                      'encoding' : 'utf8',
                      'mode' : 0666});
```

```
};

util.inherits(InputChecker,eventEmitter);

InputChecker.prototype.check = function check(input) {
   var command = input.toString().trim().substr(0,3);
   if (command == 'wr:') {
      this.emit('write',input.substr(3,input.length));
   } else if (command == 'en:') {
      this.emit('end');
   } else {
      this.emit('echo',input);
   }
};
```

Я слегка изменила его, встроив функцию `check()` прямо в определение объекта. Затем результат был преобразован в класс ES6, начиная с включения строгого режима, необходимого для использования новой функциональности классов.

Вместо того чтобы использовать `util.inherits()` для наследования методов прототипа из суперконструктора в конструктор, новый класс расширяет методы объекта `EventEmitter`. Я добавила метод-конструктор для создания и инициализации нового объекта. В этом конструкторе вызывается метод `super()` для активизации функций родительского класса:

```
'use strict';

const util = require('util');
const eventEmitter = require('events').EventEmitter;
const fs = require('fs');

class InputChecker extends eventEmitter {

   constructor(name, file) {
      super()
      this.name = name;
      this.writeStream = fs.createWriteStream('./' + file + '.txt',
         {'flags' : 'a',
          'encoding' : 'utf8',
          'mode' : 0o666});
      }

   check (input) {
      let command = input.toString().trim().substr(0,3);
      if (command == 'wr:') {
        this.emit('write',input.substr(3,input.length));
      } else if (command == 'en:') {
```

```
        this.emit('end');
      } else {
        this.emit('echo',input);
      }
   }
};

exports.InputChecker = InputChecker;
```

Метод `check()` добавляется не с использованием объекта-прототипа, а просто включается в класс как метод. Ни `var`, ни `function` с `check()` не используются — вы просто определяете метод. Однако вся рабочая логика остается неизменной или очень близкой к исходному объекту.

Приложение, использующее новый модуль, не изменяется:

```
var InputChecker = require('./class').InputChecker;

// Тестирование нового объекта и обработки событий
var ic = new InputChecker('Shelley','output');

ic.on('write', function (data) {
   this.writeStream.write(data, 'utf8');
});
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

Как показывает этот код, мне не пришлось переводить приложение в строгий режим для использования модуля, определенного в строгом режиме.

ПОЛЕЗНАЯ ДОКУМЕНТАЦИЯ MOZILLA

Mozilla Developer Network всегда остается первым источником информации обо всем, что связано с JavaScript, и новая функциональность классов тоже не является исключением. В документации имеется очень хороший справочник (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes*) по этой функциональности.

Обещания и Bluebird

На ранней стадии разработки Node создатели спорили о том, какой путь выбрать: функции обратного вызова или обещания (promises). Функции обратного вызова выиграли, что обрадовало одних и огорчило других.

Обещания теперь стали частью ES6, и вы, безусловно, можете использовать их в своих приложениях Node. Но если вы хотите сочетать обещания ES6 с базовой функциональностью Node, вам придется либо реализовать их поддержку «с нуля», либо воспользоваться модулем, предоставляющим эту поддержку. Хотя в книге я старалась по возможности избегать сторонних модулей, в данном случае я все же рекомендую воспользоваться модулем. В этом разделе будет рассмотрен чрезвычайно популярный модуль обещаний — Bluebird.

BLUEBIRD И ПОВЫШЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ

Другая причина для использования Bluebird — производительность. Автор Bluebird объясняет, почему Bluebird превосходит «встроенные» обещания ES6 по эффективности (*http://programmers.stackexchange.com/questions/278778/why-are-native-es6-promises-slower-and-more-memory-intensive-than-bluebird*).

Вместо вложенных обратных вызовов обещания ES6 используют логику условного ветвления: успех обрабатывается одной ветвью, неудача — другой. Чтобы продемонстрировать эту схему, проще всего взять типичное приложение Node, работающее с файловой системой, и преобразовать структуру с обратными вызовами в обещания.

При использовании встроенных обратных вызовов следующее приложение открывает файл, читает его содержимое, вносит изменение, после чего записывает результат в другой файл:

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
      console.error(err.stack);
   } else {

     var adjData = data.replace(/apple/g,'orange');

     fs.writeFile('./oranges.txt', adjData, function(err) {

        if (err) console.error(err);
     });
   }
});
```

Даже в этом простом примере используется двухуровневое вложение: чтение файла с последующей записью измененного содержимого.

А теперь воспользуемся Bluebird для перевода примера на обещания.

В следующем коде функция Bluebird `promisifyAll()` используется для преобразования всех функций `File System` на обещания. Вместо `readFile()` будет использоваться `readFileAsync()` — версия функции с поддержкой обещаний:

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.writeFileAsync('./oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

В этом примере при чтении содержимого файла успешная операция с данными обрабатывается функцией `then()`. Если операция завершилась неудачей, функция `catch()` обрабатывает ошибку. Если чтение прошло успешно, то данные изменяются и вызывается «обещательная» версия `writeFile()` — `writeFileAsync()`, которая записывает данные в файл. Из предыдущего примера с вложенными обратными вызовами мы знаем, что `writeFile()` просто возвращает ошибку. Эта ошибка также будет обрабатываться единой функцией `catch()`.

Хотя пример с вложенными обратными вызовами невелик, вы видите, насколько четче выглядит версия кода с обещаниями. Также становится понятно, как решается проблема с вложением, особенно с единой функцией обработки ошибок для любого количества вызовов.

Как насчет более сложного примера? Я изменила приведенный выше код и добавила дополнительный шаг, на котором создается новый подкаталог с файлом oranges.txt. В этом коде вы видите две функции `then()`. Первая обрабатывает успешный ответ на создание подкаталога, а вторая создает новый файл с измененными данными. Новый каталог создается «обещательной» функцией `mkdirAsync()`, которая возвращается в конце процесса. Эта схема играет ключевую роль в работе обещаний, потому что следующая функция `then()` на самом деле присоединяется к возвращаемой функции. Измененные данные передаются функции, в которой эти данные записываются. Любые ошибки в процессе чтения файла или создания каталога обрабатываются единой функцией `catch()`:

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.mkdirAsync('./fruit/');
   })
   .then(function(adjData) {
      return fs.writeFileAsync('./fruit/oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

Как насчет ситуаций, в которых возвращается массив результатов, например при использовании функции `File System readdir()` для получения содержимого каталога?

В таких ситуациях могут пригодиться такие функции обработки массивов, как `map()`. В следующем коде содержимое каталога возвращается в виде массива, каждый файл в этом каталоге открывается, его содержимое изменяется и записывается в файл с аналогичным именем в другом каталоге. Внутренняя функция `catch()` обрабатывает ошибки чтения и записи файлов, а внешняя — ошибки обращения к каталогам:

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readdirAsync('./apples/').map(filename => {
   fs.readFileAsync('./apples/'+filename,'utf8')
      .then(function(data) {
         var adjData = data.replace(/apple/g, 'orange');
         return fs.writeFileAsync('./oranges/'+filename, adjData);
      })
      .catch(function(error) {
         console.error(error);
      })
   })
   .catch(function(error) {
      console.error(error);
   })
```

Эта глава дает лишь начальное представление о возможностях Bluebird и преимуществах обещаний в Node. Не пожалейте времени и разберитесь в том, как использовать их (помимо других возможностей ES6) в ваших приложениях Node.

10 Комплексная разработка приложений Node

Большая часть книги посвящена базовым модулям и основной функциональности Node. Я старалась избегать описания сторонних модулей, потому что среда Node динамично развивается и поддержка сторонних модулей может быстро и радикально меняться.

Но мне кажется, что Node невозможно описать без хотя бы краткого упоминания о широком контексте приложений Node, а это означает, что вы должны ориентироваться в вопросах комплексной (full-stack) разработки приложений Node. Другими словами, вы должны ориентироваться в системах данных, API, разработке на стороне клиента — целом диапазоне технологий, у которых есть только один общий аспект — Node.

Самая распространенная форма комплексной разработки в Node обозначается сокращением MEAN — MongoDB, Express, AngularJS и Node. Тем не менее комплексная разработка может включать другие инструменты, например MySQL или Redis для разработки баз данных, или клиентские фреймворки в дополнение к AngularJS. Впрочем, Express используется практически повсеместно. Обязательно освойте Express, если вы собираетесь работать с Node.

ДАЛЬНЕЙШЕЕ ИЗУЧЕНИЕ MEAN

Если вас заинтересует тема MEAN, комплексной разработки и Express, я могу порекомендовать книгу Итана Брауна «Web Development with Node and Express: Leveraging the JavaScript Stack» (O'Reilly, 2014); книгу Шьяма Сешадри и Брэда Грина «AngularJS: Up and Running» (O'Reilly, 2014); и видео Скотта Дэвиса «Architecture of the MEAN Stack» (O'Reilly, 2015) (*http://shop.oreilly.com/product/0636920039495.do*).

Express — фреймворк для приложений Node

В главе 5 было описано небольшое подмножество функциональности, необходимой для реализации веб-приложений Node. Задача создания веб-приложений Node в лучшем случае не проста. Этим и объясняется популярность таких фреймворков для построения приложений, как Express: они предоставляют большую часть функциональности с минимальными усилиями со стороны разработчика.

Express почти повсеместно используется в мире Node, поэтому вам стоит поближе познакомиться с ним. В этой главе будет рассмотрено минимальное приложение Express, но вам, безусловно, следует заняться углубленным изучением этой темы по другим источникам.

EXPRESS — ЧАСТЬ NODE.JS FOUNDATION

История Express (*http://expressjs.com/*) начиналась не лучшим образом, но теперь Express является частью Node.js Foundation. Можно ожидать, что будущее развитие Express станет более предсказуемым, а поддержка — более надежной.

Express предоставляет хорошую документацию. Мы реализуем основную последовательность действий, описанную в документации, а затем расширим базовое приложение. Для начала создайте новый подкаталог для приложения; присвойте ему любое имя на свое усмотрение. Воспользуйтесь `npm` для создания файла package.json, укажите `app.js` как точку входа приложения. Наконец, установите фреймворк Express и сохраните его в зависимостях в файле package.json следующей командой:

```
npm install express --save
```

В документации Express представлено минимальное приложение «Hello World Express», которое следует сохранить в файле app.js:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Функция `app.get()` обрабатывает все веб-запросы GET, передавая объекты запроса и ответа, уже знакомые по предыдущим главам. По общепринятым соглашениям в приложениях Express используются сокращенные формы `req` и `res`. Эти объекты обладают той же функциональностью, что и стандартные объекты запроса и ответа, с добавлением новой функциональности, представленной Express. Например, для реакции на веб-запрос можно использовать `res.write()` и `res.end()`, как в предыдущих главах, но вы также можете использовать `res.send()` — усовершенствование Express — для реализации той же функциональности в одной строке.

Вместо того чтобы строить приложение «вручную», можно воспользоваться генератором приложений Express для построения заготовки приложений. Сейчас мы так и поступим, потому что генератор строит более подробные и полноценные приложения Express.

Сначала выполните глобальную установку генератора приложений Express:

```
sudo npm install express-generator -g
```

Затем запустите приложение с именем, которое вы хотите присвоить своему приложению. Для демонстрационных целей будет использоваться имя `bookapp`:

```
express bookapp
```

Генератор приложений Express создает необходимые подкаталоги. Перейдите в подкаталог `bookapp` и установите зависимости:

```
npm install
```

Вот и все, вы только что создали свою первую заготовку приложения Express. В среде OS X и Linux вы можете запустить его следующей командой:

```
DEBUG=bookapp:* npm start
```

Если вы находитесь в режиме командной строки Windows, выполните следующую команду:

```
set DEBUG=bookapp:* & npm start
```

Также приложение можно запустить командой `npm start`, отказавшись от отладки.

Приложение запускается и прослушивает запросы на порте Express по умолчанию 3000. При обращении к приложению из Веб возвращается простая веб-страница с приветствием «Welcome to Express».

Приложение генерирует структуру подкаталогов и файлов:

```
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

Многие из этих компонентов будут рассмотрены более подробно, а пока упомяну, что статические файлы, предназначенные для внешнего доступа, находятся в подкаталоге public. Как нетрудно заметить, графика и файлы CSS находятся в этом каталоге. Файлы динамических шаблонов контента находятся в каталоге views. Подкаталог routes содержит конечные точки, которые прослушивают веб-запросы и генерируют веб-страницы.

JADE ТЕПЕРЬ НАЗЫВАЕТСЯ PUG

Из-за нарушений прав товарного знака создатели Jade не могут называть шаблонный движок, используемый в Express и других приложениях, именем «Jade». Однако процесс преобразования Jade в Pug все еще не завершен. На момент передачи книги в печать генератор Express продолжает использовать Jade, но при попытке установить Jade как зависимость выдается сообщение об ошибке с предложением установить последнюю версию Pug вместо Jade.

Веб-сайт Pug (*https://github.com/pugjs*) все еще называется Jade, но документация и функциональность остаются неизменными. Будем надеяться, что проблема будет решена в ближайшем будущем.

Файл www в подкаталоге bin содержит стартовый сценарий для приложения. Это файл Node, преобразованный в приложение командной строки. Если вы заглянете в сгенерированный файл package.json, вы увидите, что этот файл указан в качестве стартового сценария приложения:

```
{
  "name": "bookapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

Другие сценарии для тестирования, перезапуска или других управляющих операций с вашим приложением находятся в подкаталоге bin.

Наше углубленное изучение приложения начнется с точки входа приложения — файла app.js.

Открыв файл app.js, вы найдете в нем существенно больше кода, чем в простом приложении, которое было рассмотрено ранее. В нем импортируются другие модули, в основном осуществляющие поддержку программного обеспечения промежуточного звена (middleware), которую следует ожидать от приложения с выходом в Интернет. Импортируемые модули также включают директивы импортирования модулей, специфических для приложения, из подкаталога routes:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();
```

Модули и их назначение:

- `express` — приложение Express.
- `path` — базовый модуль Node для работы с путями к файлам.
- `serve-favicon` — предоставление файла favicon.ico с заданного пути или буфера.
- `morgon` — ведение журнала запросов HTTP.
- `cookie-parser` — разбор заголовка cookie и заполнение `req.cookies`.
- `body-parser` — предоставление четырех разных типов парсеров тела запроса (но не обработка запросов с многокомпонентным телом).

Каждый из этих промежуточных модулей работает как с простейшим сервером HTTP, так и с Express.

ЧТО ТАКОЕ «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРОМЕЖУТОЧНОГО ЗВЕНА»?

Программное обеспечение промежуточного звена занимает место между системой/операционной системой/базой данных и приложением. В случае с Express к этой категории относится часть цепочки приложений, каждое из которых выполняет определенную функцию, связанную с запросом HTTP, — обрабатывает его или обрабатывает запрос для последующих промежуточных приложений. Список программного обеспечения промежуточного звена, работающего с Express, весьма обширен (*http://expressjs.com/en/resources/middleware.html*).

Следующий раздел кода app.js *монтирует* промежуточные функции (делает их доступными в приложении) по заданному пути с использованием функции `app.use()`. Порядок монтирования этих промежуточных функций важен, поэтому при добавлении промежуточной функциональности убедитесь в том, что она находится с другими промежуточными продуктами в отношениях, рекомендованных разработчиком.

Фрагмент также включает код, определяющий настройку движка представлений (вскоре я расскажу об этом более подробно):

```
// Настройка движка представлений
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Снимите комментарий после размещения значка сайта в /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

В последнем вызове `app.use()` упоминается одна из немногочисленных встроенных промежуточных функций Express, `express.static`, предназначенная для обработки всех статических файлов. Если веб-пользователь запрашивает файл HTML, JPEG или другой статический файл, запрос будет обработан `express.static`. Все статические файлы предоставляются относительно пути, заданного при монтировании промежуточных функций, — в данном случае подкаталога public.

Вернемся к вызовам функций `app.set()`, определяющим движок представлений в этом фрагменте: мы используем шаблонный движок, помогающий связать данные с конечным результатом. Один из самых популярных движков, Jade, интегрирован по умолчанию, но вы с равным успехом можете использовать другие (такие, как Mustache и EJS). Настройка определяет, в каком подкаталоге находятся шаблонные файлы (представления) и какой движок представлений должен использоваться (Jade).

НАПОМИНАЮ: JADE ТЕПЕРЬ НАЗЫВАЕТСЯ PUG

Как упоминалось ранее в этой главе, Jade теперь называется Pug. Сверьтесь с документацией Express и Pug, чтобы привести примеры в соответствие с новым именем шаблонного движка.

Когда эта книга отправилась в печать, я изменила сгенерированный файл package.json для замены модуля Jade модулем Pug:

”pug”: “2.0.0-alpha8”,

В файле app.js замените упоминание jade на pug:

app.set('view engine', 'pug');

Похоже, приложение после этого работает без проблем.

В подкаталоге views находится три файла: error.jade, index.jade и layout.jade. Для начала этого достаточно, но когда вы будете интегрировать данные

в приложение, вам понадобится нечто большее. Содержимое сгенерированного файла index.jade выглядит так:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Строка `extends layout` включает синтаксис Jade из файла layout.jade. Обратите внимание на элементы заголовка HTML (`h1`) и абзаца (`p`). Заголовку `h1` присваивается значение, переданное шаблону под именем `title`; оно же используется в элементе абзаца. Способ визуализации этих значений в шаблоне требует вернуться к файлу app.js в следующем фрагменте:

```
app.use('/', routes);
app.use('/users', users);
```

Это конечные точки, специфические для приложения, — та функциональность, которая реагирует на запросы клиента. Запрос верхнего уровня (`'/'`) обслуживается файлом index.js в подкаталоге routes, запрос `users` — файлом users.js.

В файле index.js представлен *маршрутизатор* (router) Express, предоставляющий функциональность обработки ответов. Как указано в документации Express, поведение маршрутизатора строится по следующей схеме:

```
app.МЕТОД(ПУТЬ, ОБРАБОТЧИК)
```

Здесь *МЕТОД* — метод HTTP; Express поддерживает разные методы, включая хорошо знакомые `get`, `post`, `put` и `delete`, а также менее известные `merge`, `search`, `head`, `options` и т. д. *ПУТЬ* определяет веб-путь, а *ОБРАБОТЧИК* — функцию, которая обрабатывает запрос. В файле index.js указан метод `get`, путем является корень приложения, а обработчиком — функция обратного вызова, передающая запрос и ответ:

```
var express = require('express');
var router = express.Router();

/* Запрос GET к домашней странице. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Данные (локальные переменные) и представление встречаются в вызове функции `res.render()`. Используемое представление содержится в файле index.jade, который был рассмотрен выше; как видите, значение атрибута `title` в шаблоне передается функции как данные. Попробуйте в своей копии заменить «Express» чем-нибудь другим, перезагрузите страницу и посмотрите, что при этом изменится.

Оставшаяся часть файла app.js посвящена обработке ошибок; я оставлю ее вам для самостоятельного изучения. Наше знакомство с Express было очень быстрым и непродолжительным, но я надеюсь, что даже этот простой пример даст вам представление о структуре приложений Express.

ВСТРАИВАНИЕ ДАННЫХ

Если вы хотите больше узнать о встраивании данных в приложениях Express, я нескромно порекомендую свою книгу «JavaScript Cookbook» (O'Reilly, 2010). В главе 14 описано расширение существующих приложений Express для внедрения данных MongoDB, а также встраивание контроллеров для реализации полноценной архитектуры MVC (Model-View-Controller).

Системы управления баз данных MongoDB и Redis

В листинге 7.8 главы 7 был представлен пример вставки в базу данных MySQL. Поддержка реляционных баз данных в приложениях Node, поверхностная на первых порах, существенно усилилась; в ней появились такие мощные решения, как драйвер MySQL для Node, и новые модули, такие как пакет Tedious для доступа к SQL Server в среде Microsoft Azure.

Для приложений Node также доступны некоторые другие системы баз данных. В этом разделе будут кратко рассмотрены две такие системы: MongoDB, чрезвычайно популярная в разработке Node, и Redis (моя любимая).

MongoDB

MongoDB — самая популярная база данных, используемая в приложениях Node. Эта система относится к категории *документных* баз данных. Документы кодируются в формате BSON — двоичной разновидности JSON (что, вероятно, объясняет его популярность среди разработчиков JavaScript). В MongoDB место записи таблицы занимает документ BSON, а место таблицы — коллекция.

MongoDB не единственная документная база данных. Другие популярные разновидности хранилищ данных такого типа — CouchDB (Apache), SimpleDB (Amazon), RavenDB и даже легендарная система Lotus Notes. Node в той или иной степени поддерживает большинство современных хранилищ данных, но MongoDB и CouchDB пользуются самым высоким уровнем поддержки.

СУБД MongoDB не тривиальна, и вам придется потратить некоторое время на изучение ее функциональности, прежде чем встраивать ее в приложения Node. Но когда фаза изучения будет завершена, вы сможете пользоваться превосходной поддержкой MongoDB в Node, представленной драйвером MongoDB Native NodeJS Driver (*https://github.com/mongodb/node-mongodb-native*) и дополнительной объектно-ориентированной поддержкой в Mongoose (*http://mongoosejs.com/*).

И хотя я не собираюсь подолгу задерживаться на работе с MongoDB в Node, я все же приведу один пример, чтобы вы получили представление о том, как это работает. Хотя по используемой структуре данных она отличается от реляционных баз данных, концепции остаются прежними: вы создаете базу данных, затем создаете коллекции записей и добавляете отдельные записи. После этого можно выполнять обновление, выборку или удаление записей. В примере MongoDB, приведенном в листинге 10.1, я подключаюсь к базе данных, обращаюсь к коллекции Widgets, удаляю все существующие записи, вставляю две новые записи, затем запрашиваю эти две записи и вывожу их.

Листинг 10.1. Работа с базой данных MongoDB

```
var MongoClient = require('mongodb').MongoClient;

// Подключение к базе данных
MongoClient.connect("mongodb://localhost:27017/exampleDb",
                                        function(err, db) {
   if(err) { return console.error(err); }

   // Обращение или создание коллекции widgets
   db.collection('widgets', function(err, collection) {
      if (err) return console.error(err);

      // Удаление всех документов
      collection.remove(null,{safe : true}, function(err, result) {
         if (err) return console.error(err);
         console.log('result of remove ' + result.result);

         // Создание двух записей
         var widget1 = {title : 'First Great widget',
```

```
                        desc : 'greatest widget of all',
                        price : 14.99};
        var widget2 = {title : 'Second Great widget',
                        desc : 'second greatest widget of all',
                        price : 29.99};

        collection.insertOne(widget1, {w:1}, function (err, result) {
           if (err) return console.error(err);
           console.log(result.insertedId);

           collection.insertOne(widget2, {w:1}, function(err, result) {
              if (err) return console.error(err);
              console.log(result.insertedId);

              collection.find({}).toArray(function(err,docs) {
                 console.log('found documents');
                 console.dir(docs);

                 // Закрытие базы данных
                 db.close();
              });
           });
        });
     });
  });
});
```

Да, типичная для Node мешанина обратных вызовов, но вы можете реализовать использование обещаний.

Объект MongoClient — тот объект, который вы будете использовать чаще всего для подключения к базе данных. Обратите внимание на заданный номер порта (27017): это порт по умолчанию для системы MongoDB. База данных `exampleDB` указывается в составе URL-адреса подключения, а коллекция называется `widgets` (в честь класса Widget, хорошо известного среди разработчиков).

Функции MongoDB асинхронны, как и следовало ожидать. Перед вставкой записей приложение сначала удаляет все существующие записи в коллекции, используя функцию `collection.remove()` без указания запроса. Если этого не сделать, в базе появятся дубликаты, так как MongoDB присваивает уникальные идентификаторы, сгенерированные системой, всем новым записям, а поле `title` (или другое поле) не назначается уникальным идентификатором явно. Новые записи добавляются вызовами `collection.insertOne()` с данными JSON, определяющими объект. Параметр `{w:1}` задает уровень подтверждения операции записи от MongoDB.

После того как записи будут вставлены, приложение использует `collection.find()` для поиска всех записей (снова без конкретного запроса). Функция создает курсор, а функция `toArray()` возвращает результаты курсора в виде массива для вывода на консоль вызовом `console.dir()`. Результат выглядит примерно так:

```
result of remove 1
56c5f535c51f1b8d712b6552
56c5f535c51f1b8d712b6553
found documents
[ { _id: ObjectID { _bsontype: 'ObjectID', id: 'VÅõ5Å\u001f\u001bq+eR' },
    title: 'First Great widget',
    desc: 'greatest widget of all',
    price: 14.99 },
  { _id: ObjectID { _bsontype: 'ObjectID', id: 'VÅõ5Å\u001f\u001bq+eS' },
    title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99 } ]
```

Идентификатор объекта в действительности является объектом; он представлен в формате BSON, из-за чего он выводится в нечитаемом виде. Если вы хотите получить более наглядный вывод, обратитесь к каждому отдельному полю и преобразуйте идентификатор BSON в шестнадцатеричную строку вызовом `toHexString()`:

```
 docs.forEach(function(doc) {
                    console.log('ID : ' + doc._id.toHexString());
                    console.log('desc : ' + doc.desc);
                    console.log('title : ' + doc.title);
                    console.log('price : ' + doc.price);
                });
```

Теперь результат принимает следующий вид:

```
result of remove 1
56c5fa40d36a4e7b72bfbef2
56c5fa40d36a4e7b72bfbef3
found documents
ID : 56c5fa40d36a4e7b72bfbef2
desc : greatest widget of all
title : First Great widget
price : 14.99
ID : 56c5fa40d36a4e7b72bfbef3
desc : second greatest widget of all
title : Second Great widget
price : 29.99
```

Для просмотра записей в базе данных MongoDB можно воспользоваться программой командной строки. Используйте следующую последовательность команд для ее запуска и просмотра записей:

1. Введите команду `mongo` для запуска интерфейса командной строки.
2. Введите команду `use exampleDb` для выбора базы данных `exampleDb`.
3. Введите команду `show collections` для просмотра всех коллекций.
4. Введите команду `db.widgets.find()` для просмотра записей Widget.

Если вы предпочитаете более объектно-ориентированный подход к внедрению MongoDB в ваши приложения, используйте Mongoose. Возможно, этот механизм также лучше подойдет для интеграции в Express.

Завершив эксперименты с MongoDB, не забудьте завершить работу программы.

MONGODB В ДОКУМЕНТАЦИИ NODE

У драйвера MongoDB для Node существует электронная документация, доступная в репозитории GitHub (*https://github.com/mongodb/node-mongodb-native*). Впрочем, документация драйвера также размещена на сайте MongoDB (*https://mongodb.github.io/node-mongodb-native/api-articles/nodekoarticle1.html*). Я рекомендую документацию сайта MongoDB, особенно для разработчиков, начинающих работу с системой.

Redis

Когда речь заходит о данных, в этой области обычно выделяют две основные категории: реляционные базы данных и Все Остальное, часто обозначаемое термином NoSQL. В категории NoSQL данные структурируются в виде пар «ключ/значение», особенно при хранении в памяти для очень быстрого доступа. Три самые популярные системы хранения пар «ключ/значение» — Memcached, Cassandra и Redis. Разработчикам Node повезло: в Node реализована поддержка всех трех систем хранения.

Memcached в основном используется для кэширования запросов к данным, чтобы ускорить последующие обращения к ним. Система также хорошо справляется с распределенным кэшированием, но поддержка более сложных данных в ней ограничена. Она хорошо работает в приложениях, выдающих большое количество запросов на выборку, но не столь эффективно проявляет себя в приложениях с многочисленными операциями чтения и записи. Redis — отличный

вариант для приложения второго типа. Кроме того, данные Redis могут сохраняться, и система обладает большей гибкостью, чем Memcached, — особенно в поддержке различных типов данных. Однако, в отличие от Memcached, Redis работает только на одной машине.

Те же факторы следует учитывать при сравнении Redis с Cassandra. В Cassandra, как и в Memcached, реализована поддержка кластеров. С другой стороны, как и в Memcached, поддержка структур данных в Cassandra ограничена. Система хорошо подходит для выдачи ситуативных запросов — сценарий использования, плохо подходящий для Redis. С другой стороны, система Redis проста в использовании, незатейлива и обычно работает быстрее Cassandra. По этим и другим причинам система Redis пользуется большей популярностью среди разработчиков Node.

EARN

Сокращение EARN означает «Express, AngularJS, Redis, Node». Пример EARN рассматривается в статье «The EARN Stack» (*https://www.airpair.com/express/posts/earn-stack*).

Мой любимый модуль Node для Redis устанавливается из `npm`:

```
npm install redis
```

Если вы собираетесь применять Redis в крупномасштабных проектах, я также рекомендую установить модуль `hiredis`, который работает без блокирования и повышает быстродействие:

```
npm install hiredis redis
```

Модуль `Redis` представляет собой относительно тонкую обертку для самой системы Redis. Это означает, что вам придется потратить время на изучение команд и особенностей работы системы Redis.

Чтобы использовать Redis в приложении Node, включите модуль:

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

Затем необходимо создать клиент Redis. Для этой цели используется метод `createClient`:

```
var client = redis.createClient();
```

Метод `createClient` может получать до трех необязательных параметров: номер порта (`port`), хост (`host`) и параметры (`options`). По умолчанию параметр `host` содержит значение 127.0.0.1, а `port` содержит значение 6379. Номер порта соответствует тому, который используется по умолчанию сервером Redis, поэтому стандартные настройки должны работать нормально, если сервер Redis размещен на одной машине с приложением Node.

В третьем параметре передается объект со значениями параметров, смысл которых подробно описан в документации модулей. Используйте настройки по умолчанию, пока вы не будете более уверенно чувствовать себя с Node и Redis.

После создания клиентского подключения к хранилищу данных Redis вы сможете отправлять команды серверу, пока не будет вызван метод `client.quit()`, закрывающий подключение к серверу Redis. Если вы хотите закрыть подключение, используйте метод `client.end()`. Учтите, что последний метод не ожидает, пока будут разобраны все ответы. Метод `client.end()` обычно вызывается в том случае, если приложение не знает, как продолжать работу, или вы хотите начать все заново.

Процесс выдачи команд Redis через клиентское подключение интуитивно понятен. Все команды оформлены в виде методов клиентского объекта, а аргументы команд передаются в параметрах. Как обычно в Node, в последнем параметре передается функция обратного вызова, которая возвращает ошибку или данные, полученные в ответ на команду Redis.

В следующем коде метод `client.hset()` используется для присваивания значения свойству-хешу. В Redis *хеш* представляет собой отображение между строковыми полями и значениями; например, «lastname» представляет фамилию, «firstname» — имя и т. д.:

```
client.hset("hashid", "propname", "propvalue", function(err, reply) {
   // Обработка ошибки или ответа
});
```

Команда `hset` присваивает значение, поэтому возвращаемых данных нет — только подтверждение Redis. Если вы вызываете метод, получающий несколько значений (такой, как `client.hvals`), во втором параметре функции обратного вызова передается массив — либо массив строк, либо массив объектов:

```
client.hvals(obj.member, function (err, replies) {
   if (err) {
      return console.error("error response - " + err);
   }
```

```
    console.log(replies.length + " replies:");
    replies.forEach(function (reply, i) {
      console.log("    " + i + ": " + reply);
    });
});
```

Так как обратные вызовы Node встречаются сплошь и рядом, а многие команды Redis соответствуют операциям, которые отвечают простым подтверждением успеха, модуль Redis предоставляет метод `redis.print`, который может передаваться в последнем параметре:

```
client.set("ключ", "значение", redis.print);
```

Метод `redis.print` выводит на консоль ошибку или ответ и возвращает управление.

Чтобы продемонстрировать использование Redis в Node, я создам *очередь сообщений* — приложение, получающее некоторые данные на входе и сохраняющее их в очереди. Сообщения хранятся до того момента, пока они не будут извлечены из очереди и переданы получателю (по одному или группой). Передача информации происходит асинхронно, потому что приложению, сохраняющему сообщения, не нужно сохранять постоянное соединение с получателем, а получателю не нужно постоянное соединение с приложением.

Среда Redis предоставляет идеальный механизм сохранения данных для приложений такого типа. Когда приложение принимает сообщения для сохранения, оно заносит их в конец очереди сообщений. Сообщения, принимаемые получателем, извлекаются им из начала очереди.

В примере использования Redis также задействован сервер TCP (отсюда работа с модулем Node Net), сервер HTTP и дочерний процесс. В главе 5 рассмотрен HTTP, в главе 7 — Net, а в главе 8 — дочерние процессы.

Чтобы продемонстрировать работу очереди сообщений, я создала приложение Node для обращения к файлам журналов нескольких разных поддоменов. Приложение использует дочерний процесс и команду Unix `tail -f` для обращения к последним записям журналов.

Приложение применяет к прочитанным данным журналов два объекта регулярных выражений: для извлечения ресурса и для проверки того, является ли ресурс графическим файлом. Если запрашиваемый ресурс является

графическим файлом, то приложение отправляет URL-адрес ресурса в сообщении TCP приложению очереди сообщений.

Функциональность приложения очереди сообщений ограничивается прослушиванием входящих сообщений на порте 3000 и занесением их в хранилище данных Redis.

Третья часть демонстрационного приложения — веб-сервер, прослушивающий запросы на порте 8124. С каждым запросом он обращается к базе данных Redis, извлекает начальный элемент из хранилища данных и возвращает его в объекте ответа. Если база данных Redis возвращает `null` вместо ресурса изображения, выводится сообщение о том, что приложение достигло конца очереди сообщений.

Первая часть приложения, обрабатывающая записи в журналах, приведена в листинге 10.2. Команда Unix `tail` выводит несколько последних строк текстового файла (или перенаправленных данных). С флагом `-f` команда выводит несколько строк файла и переходит в режим ожидания, прослушивая события появления новых записей. При появлении данных она возвращает новую строку. Команда `tail -f` может выполняться сразу для нескольких разных файлов; в этом случае она помечает данные, получаемые из разных источников. Приложение не интересует, в каком журнале был сгенерирован последний ответ `tail`, — ему нужно лишь содержимое записи.

После того как приложение получит запись журнала, оно применяет к ней пару регулярных выражений для поиска обращений к графическим ресурсам (файлы с расширением .jpg, .gif, .svg или .png). Если совпадение будет найдено, то приложение отправляет URL-адрес ресурса приложению очереди сообщений (сервер TCP). Приложение простое, и оно не проверяет, является совпадение строки расширением файла или просто строкой, встроенной в имя файла (например, this.jpg.html), поэтому не исключены ложные положительные срабатывания. Тем не менее приложение демонстрирует работу с Redis, а для нас это намного важнее.

Листинг 10.2. Приложение Node для обработки файлов журналов и отправки запросов графических ресурсов в очередь сообщений

```
var spawn = require('child_process').spawn;
var net = require('net');

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

```
// Соединение с сервером TCP
client.connect ('3000','examples.burningbird.net', function() {
    console.log('connected to server');
});

// Запуск дочернего процесса
var logs = spawn('tail', ['-f',
        '/home/main/logs/access.log',
        '/home/tech/logs/access.log',
        '/home/shelleypowers/logs/access.log',
        '/home/green/logs/access.log',
        '/home/puppies/logs/access.log']);

// Обработка данных дочернего процесса
logs.stdout.setEncoding('utf8');
logs.stdout.on('data', function(data) {

   // URL-адрес ресурса
   var re = /GET\s(\S+)\sHTTP/g;

   // Проверка графики
   var re2 = /\.gif|\.png|\.jpg|\.svg/;

   // Извлечение URL
   var parts = re.exec(data);
   console.log(parts[1]);

   // Поиск графического ресурса (и сохранение в случае обнаружения)
   var tst = re2.test(parts[1]);
   if (tst) {
      client.write(parts[1]);
   }
});
logs.stderr.on('data', function(data) {
   console.log('stderr: ' + data);
});

logs.on('exit', function(code) {
   console.log('child process exited with code ' + code);
   client.end();
});
```

Ниже показан типичный набор журнальных записей для этого приложения (интересующие нас строки — обращения к графическим файлам — выделены жирным шрифтом):

```
/robots.txt
/weblog
/writings/fiction?page=10
/images/kite.jpg
/node/145
/culture/book-reviews/silkworm
/feed/atom/
/images/visitmologo.jpg
/images/canvas.png
/sites/default/files/paws.png
/feeds/atom.xml
```

В листинге 10.3 приведен код очереди сообщений. Это простое приложение запускает сервер TCP и прослушивает входящие сообщения. Получив сообщение, оно извлекает данные из сообщения и сохраняет их в базе данных Redis. Приложение использует команду Redis `rpush` для занесения данных в конец списка изображений (строка, выделенная жирным шрифтом).

Листинг 10.3. Очередь сообщений принимает входящие сообщения и заносит их в список Redis

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // Создание клиента Redis
   var client = redis.createClient();

   client.on('error', function(err) {
     console.log('Error ' + err);
   });

   // Шестая база данных - очередь графических ресурсов
   client.select(6);
   // Прослушивание входящих данных
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);

      // Сохранение данных
      client.rpush('images',data);
   });

}).listen(3000);
server.on('close', function(err) {
```

```
   client.quit();
});

console.log('listening on port 3000');
```

Консольный вывод приложения очереди сообщений обычно выглядит примерно так:

```
listening on port 3000
connected
/images/venus.png from 173.255.206.103 39519
/images/kite.jpg from 173.255.206.103 39519
/images/visitmologo.jpg from 173.255.206.103 39519
/images/canvas.png from 173.255.206.103 39519
/sites/default/files/paws.png from 173.255.206.103 39519
```

Последний компонент демонстрационного приложения — сервер HTTP, прослушивающий запросы на порте 8124 (листинг 10.4). С получением очередного запроса сервер HTTP обращается к базе данных Redis, извлекает из списка следующую запись и выводит ее в ответ. Если данных в списке больше нет (то есть если Redis возвращает ответ `null`), выводится сообщение о том, что очередь сообщений пуста.

Листинг 10.4. Сервер HTTP извлекает сообщения из списка Redis и возвращает пользователю

```
var redis = require("redis"),
    http = require('http');

var messageServer = http.createServer();

// Прослушивание входящего запроса
messageServer.on('request', function (req, res) {

   // Сначала отфильтровывается запрос значка
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }

  // Создание клиента Redis
  var client = redis.createClient();

  client.on('error', function (err) {
    console.log('Error ' + err);
  });
```

```
  // Выбор базы данных 6 - очереди графических ресурсов
  client.select(6);
   client.lpop('images', function(err, reply) {
      if(err) {
        return console.error('error response ' + err);
      }

      // Если получены данные
      if (reply) {
         res.write(reply + '\n');
      } else {
         res.write('End of queue\n');
      }
      res.end();

   });
   client.quit();

});

messageServer.listen(8124);

console.log('listening on 8124');
```

Обращение к приложению сервера HTTP в браузере возвращает URL графического ресурса при каждом запросе (обновлении браузера), пока очередь сообщений не опустеет.

Используемые данные очень просты, а работать с ними приходится часто; именно поэтому Redis идеально подходит для приложений такого типа. Это быстрое, несложное хранилище данных, включение которого в приложение Node не требует особых усилий.

КОГДА СОЗДАЮТСЯ КЛИЕНТЫ REDIS

В работе с Redis я иногда создаю клиента Redis, который существует на протяжении всего жизненного цикла приложения, а в других случаях я создаю клиента Redis и освобождаю его сразу же после завершения команды Redis.

В каких случаях лучше создавать долгосрочное соединение с Redis, а в каких лучше создать соединение и немедленно освободить его?

Хороший вопрос.

Чтобы протестировать последствия двух разных решений, я создала сервер TCP, который прослушивал запросы и сохранял простой хеш в базе данных Redis. Затем я создала другое приложение в виде клиента TCP, который просто отправлял объект серверу в сообщении TCP.

При помощи приложения ApacheBench я запустила несколько параллельных итераций клиента и измерила, сколько времени заняло каждое выполнение. Первая группа запускалась с созданием соединения на время существования сервера, а вторая — с созданием соединения для каждого запроса и его немедленным освобождением.

Я ожидала, что приложение с долгосрочным соединением будет работать быстрее, и была права... в определенной степени. Где-то на середине тестирования с долгосрочным соединением приложение резко замедлилось на короткий промежуток времени, а затем продолжило выполняться в относительно быстром темпе.

Конечно, скорее всего, это происходило из-за того, что находящиеся в очереди запросы к базе данных Redis блокировали работу приложения Node, — по крайней мере временно, пока очередь не освободилась. С открытием и закрытием соединения для каждого запроса такая ситуация не возникала, потому что дополнительные затраты ресурсов замедляли работу приложения и оно не сталкивалось с порогом производительности, обусловленным запросами параллельных пользователей.

AngularJS и другие комплексные фреймворки

Прежде всего хочу сказать, что термином «фреймворк» часто злоупотребляют. Им обозначаются библиотеки, ориентированные на взаимодействие с пользователем (например, jQuery), графические библиотеки (например, D3), Express и ряд других, более современных комплексных приложений. В этой главе под термином «фреймворк» я подразумеваю комплексные инфраструктуры, такие как AngularJS, Ember и Backbone.

Чтобы освоить комплексные фреймворки, вам стоит посетить сайт TodoMVC (*http://todomvc.com/*). Этот сайт определяет требования к одному из основных типов приложений — списку задач и предлагает всем разработчикам фреймворков предоставить реализации такого приложения. Также на сайте размещена базовая реализация такого приложения, не использующая никакие фреймворки, и реализация на базе jQuery. Сайт предоставляет разработчикам возможность сравнить, как одна и та же функциональность реализуется в разных фреймворках. На сайте представлены все популярные фреймворки, не только AngularJS: Backbone.js, Dojo, Ember, React и др. Также представлены приложения, в которых задействовано несколько технологий, например AngularJS, Express и платформа Google Cloud.

В требованиях к списку задач приведена рекомендуемая структура каталогов и файлов:

```
index.html
package.json
node_modules/
css
└── app.css
js/
├── app.js
├── controllers/
└── models/
readme.md
```

В этой структуре нет ничего таинственного; она похожа на ту, которая использовалась в приложении ExpressJS. Однако способы выполнения этих требований разными фреймворками могут быть очень разными; вот почему приложение ToDo — отличный способ изучить различия в работе фреймворков.

В качестве примера рассмотрим часть кода для пары фреймворков: AngularJS и Backbone.js. Я не буду приводить большую часть кода, потому что она наверняка изменится к тому времени, когда вы будете читать эту книгу. Я начну с AngularJS и сосредоточусь на оптимизированном приложении — на сайте представлено несколько разных реализаций с AngularJS. На рис. 10.1 изображено приложение после включения трех задач в список.

Рис. 10.1. Список после добавления трех задач

Прежде всего заметим, что корень приложения — app.js — очень прост, как и следовало ожидать при разбиении всей функциональности по разным подгруппам схемы «модель-представление-контроллер».

```
/* jshint undef: true, unused: true */
/*global angular */
(function () {
'use strict';

/**
   * Главный модуль TodoMVC подключает все модули зависимостей
из одноименных файлов
   *
   * @type {angular.Module}
   */
angular.module('todomvc', ['todoCtrl', 'todoFocus', 'todoStorage']);
})();
```

Приложению присвоено имя `todomvc`, и оно включает три модуля: `todoCtrl`, `todoFocus` и `todoStorage`. Пользовательский интерфейс содержится в файле index.html, расположенном в корневом каталоге. Файл довольно большой, поэтому я приведу лишь небольшой фрагмент. Тело главной страницы заключено в элемент `section` со следующим определением:

```
<section id="todoapp" ng-controller="TodoCtrl as TC">
...
</section>
```

AngularJS добавляет в HTML специальные аннотации, называемые *дериватами* (derivatives). Их легко узнать, так как все стандартные дериваты начинаются с префикса `ng-`: `ng-submit`, `ng-blur`, `ng-model` и т. д. Во фрагменте кода дериват `ng-controller` определяет контроллер для представления (`TodoCtrl`) и обозначение, которое будет использоваться для ссылки на него в шаблоне (`TC`):

```
<form ng-submit="TC.doneEditing(todo, $index)">
    <input class="edit"
       ng-trim="false"
       ng-model="todo.title"
       ng-blur="TC.doneEditing(todo, $index)"
       ng-keydown="($event.keyCode === TC.ESCAPE_KEY)
                   && TC.revertEditing($index)"
       todo-focus="todo === TC.editedTodo">
</form>
```

В этом фрагменте вы видите несколько дериватов; их смысл в основном интуитивно понятен. В деривате `ng-model` представление вступает в контакт

с моделью (данными), в этом случае `todo.title`. `TC.doneEditing` и `TC.revertEditing` — функции контроллера. Я вынесла их код из файла контроллера и продублировала ниже. Функция `TC.doneEditing` сбрасывает объект `TC.editedTodo`, усекает отредактированный заголовок задачи, а при отсутствии заголовка удаляет задачу. Функция `TC.revertEditing` также сбрасывает объект задачи и снова связывает исходную задачу с индексом исходной задачи в массиве задач:

```
TC.doneEditing = function (todo, index) {
      TC.editedTodo = {};
      todo.title = todo.title.trim();
      if (!todo.title) {
        TC.removeTodo(index);
      }
};

TC.revertEditing = function (index) {
      TC.editedTodo = {};
      todos[index] = TC.originalTodo;
};
```

В этом коде нет ничего особенно сложного. Найдите его копию на GitHub (*https://github.com/tastejs/todomvc/tree/gh-pages/examples/angularjs-perf*) и опробуйте сами.

Приложение на базе Backbone.js не отличается от приложения AngularJS ни внешним видом, ни поведением, но его исходный код выглядит совершенно иначе. Файл app.js для AngularJS был не очень большим, а файл для Backbone.js еще меньше:

```
/*global $ */
/*jshint unused:false */
var app = app || {};
var ENTER_KEY = 13;
var ESC_KEY = 27;

$(function () {
'use strict';

// Все начинается с создания приложения
new app.AppView();
});
```

По сути, `app.AppView()` запускает приложение. Файл app.js прост, но о реализации `app.AppView()` этого сказать нельзя. Вместо того чтобы размечать HTML дериватами, как в AngularJS, Backbone.js широко использует шаблоны

Userscore. В файле index.html они встречаются во встроенных сценарных элементах вроде следующего, который представляет шаблон каждой отдельной задачи. Разметка HTML чередуется с шаблонными тегами `<%=…%>` (`title`, признак состояния флажка):

```
<script type="text/template" id="item-template">
      <div class="view">
        <input class="toggle" type="checkbox" <%= completed ? 'checked'
             : '' %>>
        <label><%- title %></label>
        <button class="destroy"></button>
      </div>
      <input class="edit" value="<%- title %>">
    </script>
```

Визуализация элементов выполняется в файле todo-view.js, но ее движущей силой является файл app-view.js. Приведу часть этого файла:

```
// Добавление задачи в список. Для задачи создается
// представление, а в <ul> добавляется соответствующий элемент.
    addOne: function (todo) {
      var view = new app.TodoView({ model: todo });
      this.$list.append(view.render().el);
    },
```

Ниже приведен фрагмент файла todo-view.js, обеспечивающего визуализацию. В нем встречается ссылка на идентификатор элемента списка `item-template`, заданный ранее в сценарии, встроенном в файл index.html. Разметка HTML в сценарном элементе в файле index.html предоставляет шаблон для элементов, сгенерированных этим представлением. В шаблоне находится заполнитель для данных, предоставленных моделью приложения. В приложении списка задач эти данные состоят из заголовка элемента и признака его заполнения.

```
// Элементом DOM для элемента списка...
app.TodoView = Backbone.View.extend({
    //... является тег li.
    tagName:  'li',
    // Кэширование шаблонной функции для одного элемента.
    template: _.template($('#item-template').html()),
    ...
    // Повторная визуализация заголовков элементов списков.
    render: function () {
      // Backbone LocalStorage добавляет атрибут `id` сразу же после
      // создания модели.  В результате TodoView строит элемент дважды,
      // после создания модели и при изменении `id`.  Вторую лишнюю
      // визуализацию, обусловленную изменением `id`, нужно отфильтровать.
```

```
    // Это известная ошибка Backbone LocalStorage, для которой
    // приходится создавать обходное решение.
    // https://github.com/tastejs/todomvc/issues/469
    if (this.model.changed.id !== undefined) {
      return;
    }

    this.$el.html(this.template(this.model.toJSON()));
    this.$el.toggleClass('completed', this.model.get('completed'));
    this.toggleVisible();
    this.$input = this.$('.edit');
    return this;
  },
```

Разобраться в происходящем в случае с Backbone.js несколько труднее, чем с AngularJS, но, как и в предыдущем случае, изучение приложения ToDo сильно проясняет ситуацию. Я также рекомендую опробовать эту версию на практике.

Визуализация представления — всего лишь одно из различий между фреймворками. AngularJS предоставляет двустороннее связывание данных, что означает автоматическую синхронизацию изменений в пользовательском интерфейсе и модели. Backbone.js использует архитектуру MVP (Model-View-Presenter), тогда как в AngularJS используется MVC (Model-View-Controller); это означает, что Backbone.js не обеспечивает сходного связывания данных и вам придется реализовать его самостоятельно. С другой стороны, фреймворк Backbone.js проще и может работать быстрее, чем AngularJS, хотя AngularJS обычно более понятен для изучения разработчиками.

Два этих и другие комплексные фреймворки используются для динамического создания веб-страниц. Под этим термином я подразумеваю не динамическое генерирование страниц, рассмотренное ранее в разделе «Express — фреймворк для приложений Node» на с. 227, — эти фреймворки позволяют создавать так называемые *одностраничные приложения* (SPA, Single-Page Application). Вместо того чтобы генерировать разметку HTML на сервере и отправлять ее браузеру, они упаковывают данные, отправляют их браузеру, а затем форматируют веб-страницу средствами JavaScript.

Преимущество такой функциональности заключается в том, что она снимает необходимость в обновлении веб-страницы при каждом изменении представления данных или переходе к детализированному представлению.

Возьмем приложение Gmail: оно также относится к категории SPA. Когда вы открываете в приложении папку «Входящие» и обращаетесь к одному из

сообщений, страница не перезагружается. Вместо этого все данные, необходимые для отображения сообщения, загружаются с сервера и встраиваются в представление страницы. В результате приложение работает быстро и более понятно для пользователя. Но поверьте, на исходный код такой страницы лучше не смотреть. Если вы хотите сохранить рассудок, не пытайтесь открывать страницы Google в режиме исходного кода в браузере.

Что должен предоставлять хороший фреймворк? По моему мнению, одной из функций, которые должны поддерживаться фреймворками, является связывание данных между данными и представлением. Это означает, что при изменении данных пользовательский интерфейс должен автоматически обновиться. Также фреймворк должен поддерживать шаблонный движок, например Jade, использовавшийся ранее в примере Express. Также необходим механизм сокращения избыточного кода, поэтому фреймворк должен поддерживать многократное использование компонентов и (или) модуляризацию.

В приложении Express была продемонстрирована связь между маршрутизацией URL и функциями. URL-адрес становится однозначным способом обращения к группам данных или отдельным элементам. Чтобы найти конкретного студента, пользователь вводит URL вида */students/A1234*, а запрос маршрутизируется к странице с подробной информацией о студенте с идентификатором A1234. Фреймворк должен предоставлять поддержку маршрутизации такого типа.

Фреймворк также должен поддерживать схему MV*, что означает как минимум отделение бизнес-логики от логики визуализации. Это может быть разновидность MVC (Model-View-Controller), MVP (Model-View-Presenter), MVVM (Model-View-View-Model) и т. д., но как минимум должно поддерживаться отделение данных от интерфейса. И конечно, с учетом контекста книги, фреймворк должен интегрироваться с Node.

11 Node в разработке и эксплуатации приложений

Рождение Node совпало с появлением целого семейства новых инструментов и приемов для разработки, управления и сопровождения кода. Отладка, тестирование, управление задачами, развертывание и поддержка являются ключевыми элементами любого проекта Node; к счастью, большинство этих операций автоматизировано.

В этой главе представлены некоторые инструменты и концепции Node. Список не полон, но он станет хорошей отправной точкой для дальнейших исследований.

Отладка приложений Node

Признаюсь, я применяю вывод на консоль при отладке чаще, чем следовало бы. Это простой метод проверки значений переменных и результатов. Однако у консольного вывода есть свои недостатки: он влияет на динамику и поведение приложения, и сам факт его использования может маскировать — или создавать — проблемы. На самом деле лучше применять специализированные средства отладки, особенно если объем приложения превышает несколько простых блоков кода.

Node предоставляет встроенный отладчик, который может использоваться для расстановки точек прерывания в коде и добавления *наблюдателей* (watchers) для отслеживания промежуточных результатов. Это не самый изощренный отладчик в мире, но для выявления ошибок и потенциальных проблем его

достаточно. В следующем разделе будет рассмотрен более сложный инструмент — Node Inspector.

Отладчик Node

Если у меня есть выбор, я всегда отдаю предпочтение «родным» реализациям перед сторонней функциональностью. К счастью, Node предоставляет встроенную поддержку отладки. Она относительно проста, но может быть полезной.

Чтобы установить точку прерывания в коде, вставьте команду `debugger` прямо в код:

```
for (var i = 0; i <= test; i++) {
   debugger;
   second+=i;
}
```

Чтобы приступить к отладке приложения, укажите параметр `debug` при запуске приложения:

```
node debug application
```

Для демонстрации процесса отладки я создала приложение с двумя встроенными точками отладки:

```
var fs = require('fs');
var concat = require('./external.js').concatArray;

var test = 10;
var second = 'test';

for (var i = 0; i <= test; i++) {
   debugger;
   second+=i;
}

setTimeout(function() {
   debugger;
   test = 1000;
   console.log(second);
}, 1000);

fs.readFile('./log.txt', 'utf8', function (err,data) {
   if (err) {
      return console.log(err);
```

```
    }
    var arry = ['apple','orange','strawberry'];
    var arry2 = concat(data,arry);
    console.log(arry2);
});
```

Приложение запускается следующей командой:

```
node debug debugtest
```

Если приложение Node было запущено с флагом командной строки `--debug`, отладчик также можно запустить с присоединением к процессу с заданным идентификатором pid:

```
node debug -p 3383
```

Или отладчик запускается с подключением к работающему процессу по URI:

```
node debug http://localhost:3000
```

Когда отладчик откроется, приложение прерывает выполнение в строке 1 и выводит верхнюю часть кода:

```
< Debugger listening on port 5858
debug> . ok
break in debugtest.js:1
> 1 var fs = require('fs');
  2 var concat = require('./external.js').concatArray;
  3
```

Команда `list` используется для вывода строк исходного кода в заданном контексте. Команда вида `list(10)` выведет 10 предшествующих и 10 следующих строк кода. Команда `list(25)` в приложении debugtest выводит все строки в приложении и нумерует их. В этот момент можно добавить дополнительные строки `setBreakpoint` (в сокращенном виде `sb`). Мы установим точку прерывания в строке 19 тестового приложения (в функции обратного вызова `fs.readFile()`). Также точка прерывания будет установлена в строке 3 модуля external.js:

```
debug> sb(19)
debug> sb('external.js',3)
```

Вы получите предупреждение о том, что сценарий external.js еще не загружен. На функциональность это предупреждение не влияет.

Команда `watch('выражение')` в отладчике устанавливает наблюдателя для отслеживания значения переменной или выражения. Мы будем отслеживать переменные `test` и `second` variables, параметр `data` и массив `arry2`:

```
debug> watch('test');
debug> watch('second');
debug> watch('data');
debug> watch('arry2');
```

Наконец, все готово к началу отладки. Команда `cont` или `c` выполняет приложение до первой точки прерывания. Из выходных данных видно, что программа остановилась на первой точке прерывания, а также выводятся четыре отслеживаемых значения. Двум — `test` и `second` — присвоены реальные значения, два других содержат `<error>`. Дело в том, что приложение в данный момент находится за пределами области видимости функций, в которых определен параметр (`data`) и переменная (`arry2`). Пока не обращайте на них внимания.

```
debug> c
break in debugtest.js:8
Watchers:
  0: test = 10
  1: second = "test"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
> 8    debugger;
  9    second+=i;
 10 }
```

Есть еще несколько команд, которые вы можете опробовать, прежде чем переходить к следующей точке прерывания. Команда `scripts` выводит список загруженных сценариев:

```
debug> scripts
* 57: debugtest.js
  58: external.js
debug>
```

Команда `version` выводит версию V8. Снова введите команду `c`, чтобы перейти к следующей точке прерывания:

```
debug> c
break in debugtest.js:8
Watchers:
  0: test = 10
  1: second = "test0"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
> 8    debugger;
  9    second+=i;
 10 }
```

Обратите внимание на изменившееся значение переменной `second`. Дело в том, что переменная изменяется в цикле `for`, в котором находится отладчик. Если несколько раз ввести команду `c`, цикл продолжает выполняться и вы видите, как переменная раз за разом изменяется. К сожалению, сбросить точку, установленную командой отладчика, не удастся, но вы можете сбросить точку прерывания, установленную командой `setBreakpoint` или `sb`. Команда `clearBreakpoint` (сокращенно `cb`) получает имя сценария и номер строки, в которой устанавливается точка прерывания:

```
cb('debugtest.js',19)
```

Установленный наблюдатель сбрасывается командой `unwatch`:

```
debug> unwatch('second')
```

Команда `sb` без указания значения устанавливает точку прерывания в текущей строке:

```
debug> sb();
```

В нашем приложении отладчик выполняет приложение до следующей точки прерывания в функции обратного вызова `fs.readFile()`. Теперь мы видим, что параметр `data` изменился:

```
debug> c
break in debugtest.js:19
Watchers:
  0: test = 10
  1: second = "test012345678910"
  2: data = "test"
  3: arry2 = undefined
```

```
 17
 18 fs.readFile('./log.txt', 'utf8', function (err,data) {
>19   if (err) {
 20     return console.log(err);
 21   }
```

Кроме того, для переменной `arry2` вместо ошибки выводится значение `undefined`.

Вместо того чтобы продолжать выполнение программы командой `c`, перейдем к выполнению приложения в пошаговом режиме командой `next` или `n`. Когда выполнение доходит до строки 23, отладчик открывает модуль `external` и прерывает выполнение в строке 3 из-за ранее установленной точки прерывания:

```
debug> n
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: data = "<error>"
  3: arry2 = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3   return arry.map(function(element) {
  4         return str + ' ' + element;
  5   });
```

Также можно было дойти до строки 23 в приложении и воспользоваться командой `step` или `s` для захода в функцию модуля:

```
debug> s
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: arry2 = "<error>"
  3: data = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3   return arry.map(function(element) {
  4         return str + ' ' + element;
  5   });
```

Обратите внимание: теперь для всех отслеживаемых значений выводится ошибка. На этой стадии выполнение полностью вышло из контекста родительского приложения. Вы можете определять наблюдателей во время нахождения в функциях или во внешних модулях для предотвращения подобных ошибок или отслеживания переменных в контексте, в котором они были определены (на случай, если одна и та же переменная используется в приложении и модуле или в других функциях).

ОШИБКА CONTINUE

Если вы введете команду cont или c в самом конце приложения, отладчик зависнет и прервать его работу вы уже не сможете. Одна из известных ошибок Node.

Команда `backtrace` (сокращенно `bt`) предоставляет *обратную трассировку* текущего контекста выполнения. Результат, возвращаемый на этой стадии отладки, показан в следующем блоке:

```
debug> bt
#0 concatArray external.js:3:3
#1 debugtest.js:23:15
```

Выводятся две записи: для текущей строки приложения и для текущей строки в функции импортированного модуля.

Выполните внешнюю функцию в пошаговом режиме или вернитесь к приложению командой `out` (сокращенно `o`). Эта команда возвращает управление приложению из функции (будь то функция локального сценария или модуля).

Отладчик Node создан на основе REPL; вы даже можете вызвать сеанс REPL отладчика командой `repl`. Принудительное завершение сценария осуществляется командой `kill`, а перезапуск — командой `restart`. Учтите, что при перезапуске сценария будут сброшены все точки прерывания и наблюдатели.

Так как отладчик работает в REPL, приложение можно завершить нажатием клавиш Ctrl+C или командой `.exit`.

Node Inspector

Если вы готовы работать с более совершенными средствами отладки, вам стоит присмотреться к Node Inspector. Node Inspector включает функциональность отладки, которая, вероятно, знакома вам по отладчику Blink DevTools, работающему с Chrome или Opera.

Оборотная сторона более совершенного инструмента — повышенная сложность и наличие визуальной среды для отладки. Node Inspector предъявляет повышенные требования к системе. Например, когда я попыталась запустить его на одной из моих машин с Windows 10, приложение сообщило мне, что я должна установить .NET Framework 2.0 SDK или Microsoft Visual Studio 2005.

УСТАНОВКА NODE INSPECTOR

Если вы столкнулись с ошибкой Visual Studio, попробуйте изменить версию Visual Studio, используемую приложением, следующей командой:

npm install -g node-inspector --msvs_version=2013

(Или укажите другую версию, которая у вас установлена.)

Если ваша рабочая среда соответствует требованиям Node Inspector, установите программу следующей командой:

```
npm install -g node-inspector
```

Чтобы использовать Node Inspector, запустите свое приложение Node следующей командой (при запуске с Node Inspector необходимо добавить расширение .js):

```
node-debug application.js
```

Здесь может произойти одно из двух. Если в вашей системе по умолчанию выбран браузер Chrome или Opera, приложение откроется в инструментарии разработчика. Если по умолчанию используется другой браузер, запустите другой браузер вручную и введите URL *http://127.0.0.1:8080/?port=5858*.

ДОКУМЕНТАЦИЯ NODE INSPECTOR

В репозитории GitHub доступна документация по Node Inspector (*https://github.com/node-inspector/node-inspector*). StrongLoop — компания, обеспечивающая поддержку программы — также предоставляет документацию по использованию Node Inspector (http://bit.ly/1OQEMkB). Как указано в документации, полезную информацию также можно найти в документации Chrome Developer Tools (*https://developers.google.com/web/tools/chrome-devtools/*). Учтите, что Google часто изменяет местоположение своей документации и в любой момент времени ссылка может оказаться недействительной.

Я открыла свой файл debugtest.js, созданный в предыдущем разделе в Node Inspector. На рис. 11.1 изображен отладчик после загрузки файла и нажатия кнопки запуска, расположенной в правом верхнем углу инструмента. Node Inspector поддерживает команду `debugger`, которая становится первой точкой прерывания приложения. Отслеживаемые переменные выводятся под кнопками управления выполнением программы.

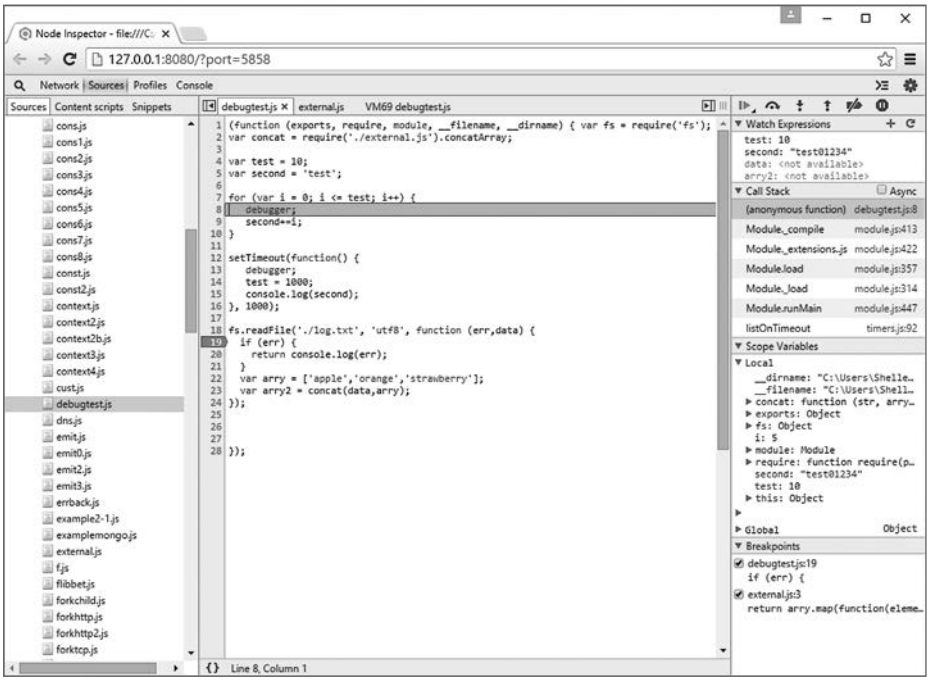


Рис. 11.1. Запуск debugtest.js в отладчике Node Inspector

Новая точка прерывания устанавливается щелчком на поле слева от строки, в которой она должна быть установлена. Чтобы добавить нового наблюдателя, щелкните на кнопке со знаком «плюс» (+) в заголовке Watch Expressions. Команды, приведенные выше, позволяют (слева направо) выполнить приложение до следующей точки прерывания и следующий вызов функции без захода в функцию, сбросить все точки прерывания и приостановить приложение. Новая особенность этого визуального интерфейса — список приложений/модулей в левом окне, стек вызовов, список переменных в области видимости (локальных и глобальных) и средства для установки точек прерывания на правой панели. Если вы хотите добавить точку прерывания в импортированном модуле external.js, для этого достаточно открыть файл в списке слева и вставить

точку прерывания. На рис. 11.2 показан отладчик с загруженным модулем и достигнутой точкой прерывания.

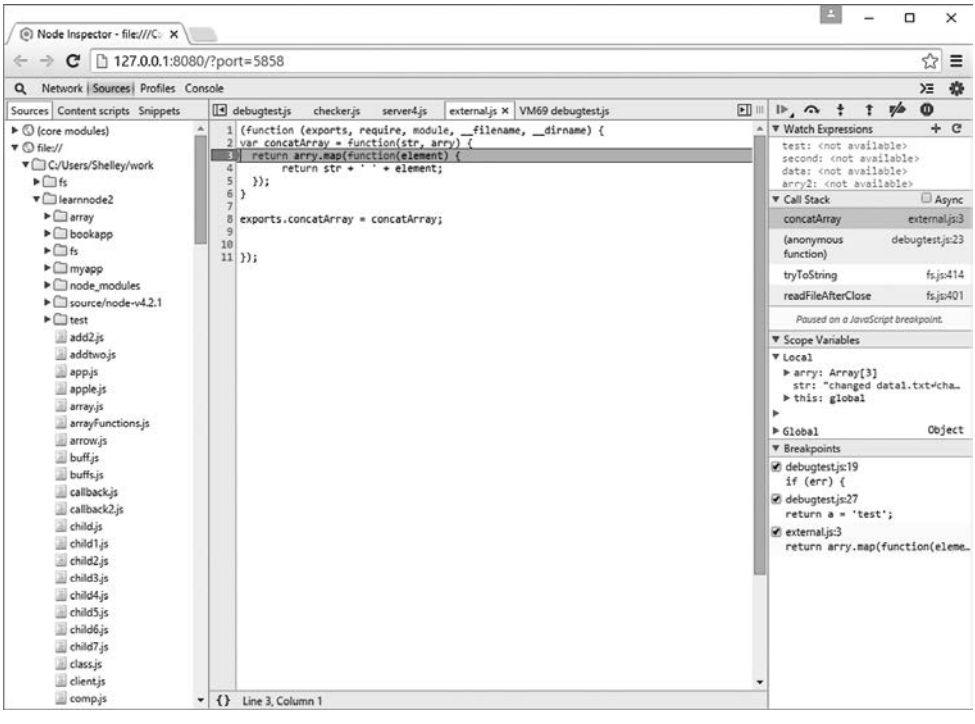


Рис. 11.2. Node Inspector с загруженным модулем external.js и достигнутой точкой прерывания

Интересный момент: при взгляде на приложение, загруженное в Node Inspector, становится видно, что приложение «завернуто» в анонимную функцию, как упоминалось в главе 3 («Как Node находит и загружает модуль», с. 77).

Завершив работу с Node Inspector, вы можете закрыть браузер и завершить пакетную операцию нажатием Ctrl+C.

БОЛЕЕ МОЩНЫЕ ИНСТРУМЕНТЫ

Как встроенный отладчик, так и Node Inspector предоставляет достойную функциональность отладки приложений Node. А если вы готовы выделить время на настройку и освоение более сложной интегрированной среды разработки (IDE), вы также можете воспользоваться отладчиком Nodeclipse (*http://www.nodeclipse.org/usage*) из популярной среды Eclipse.

Модульное тестирование

Модульное тестирование — метод изоляции компонентов приложения для тестирования. Многие тесты, содержащиеся в подкаталоге tests модулей Node, являются модульными. Все тесты в подкаталоге test установки Node тоже являются модульными. Многие из этих тестов были построены с использованием модуля `Assert`.

Модульное тестирование и модуль Assert

Тестовые утверждения проверяют значение некоторых выражений; конечным результатом такой проверки является логическое значение `true` или `false`. Например, если вы тестируете команду возврата из вызова функции, можно сначала проверить, что возвращаемое значение представляет собой массив (первое утверждение). Если содержимое массива должно иметь определенную длину, выполните условную проверку длины (второе утверждение) и т. д. В Node существует встроенный модуль `Assert`, упрощающий создание тестовых утверждений. Этот модуль предназначен для внутреннего использования в Node, но вы можете пользоваться им. Нужно лишь осознавать, что это не полноценный тестовый фреймворк.

Модуль `Assert` включается в приложение следующей командой:

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

Чтобы научиться использовать модуль `Assert`, посмотрим, как он используется в существующих модулях. Приложение Node использует модуль `Assert` в своих модульных тестах. Например, есть тестовое приложение test-util.js, предназначенное для тестирования модуля `Utilities`. Следующий фрагмент тестирует метод `isArray`:

```
// isArray
assert.equal(true, util.isArray([]));
assert.equal(true, util.isArray(Array()));
assert.equal(true, util.isArray(new Array()));
assert.equal(true, util.isArray(new Array(5)));
assert.equal(true, util.isArray(new Array('with', 'some', 'entries')));
assert.equal(true, util.isArray(context('Array')()));
assert.equal(false, util.isArray({}));
assert.equal(false, util.isArray({ push: function() {} }));
assert.equal(false, util.isArray(/regexp/));
assert.equal(false, util.isArray(new Error));
assert.equal(false, util.isArray(Object.create(Array.prototype)));
```

Методы `assert.equal()` и `assert.strictEqual()` имеют два обязательных параметра: ожидаемый результат и выражение, при вычислении которого он должен быть получен. Если выражение дает результат `true` и ожидаемый результат тоже равен `true`, метод `assert.equal` завершается успешно и ничего не выводит.

Но если при вычислении выражения будет получен ответ, отличный от ожидаемого, метод `assert.equal` выдает исключение. Если привести первую строку тестов `isArray` в исходном коде Node к следующему виду:

```
assert.equal(false, util.isArray([]));
```

то результат выглядит так:

```
assert.js:89
  throw new assert.AssertionError({
  ^
AssertionError: false == true
    at Object.<anonymous> (/home/examples/public_html/
                          learnnode2/asserttest.js:4:8)
    at Module._compile (module.js:409:26)
    at Object.Module._extensions..js (module.js:416:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Function.Module.runMain (module.js:441:10)
    at startup (node.js:134:18)
    at node.js:962:3
```

Методы `assert.equal()` и `assert.strictEqual()` также могут получать третий необязательный параметр — сообщение, которое выводится вместо стандартного при неудаче:

```
assert.equal(false, util.isArray([]), 'Test 1Ab failed');
```

Передача сообщения помогает точно определить, какой именно из нескольких тестов не прошел. Пример теста с передачей нестандартного сообщения встречается в тестовом коде `node-redis`:

```
assert.equal(str, results, label + " " + str +
               " does not match " + results);
```

Нестандартное сообщение выводится при перехвате исключения. Все методы `Assert`, перечисленные ниже, получают одинаковый набор из трех параметров, но с разными отношениями между ожидаемым результатом и выражением, как следует из имени:

- `assert.equal` — не проходит, если результат выражения и заданное значение не равны.
- `assert.strictEqual` — не проходит, если результат выражения и заданное значение не связаны строгим равенством.
- `assert.notEqual` — не проходит, если результат выражения и заданное значение равны.
- `assert.notStrictEqual` — не проходит, если результат выражения и заданное значение связаны строгим равенством.
- `assert.deepEqual` — не проходит, если результат выражения и заданное значение не равны.
- `assert.notDeepEqual` — не проходит, если результат выражения и заданное значение равны.
- `assert.deepStrictEqual` — аналог `assert.deepEqual()`, но примитивы сравниваются с использованием операции строгого равенства (`===`).
- `assert.notDeepStrictEqual` — проверка глубокого строгого неравенства.

Глубокие (`deep`) методы работают со сложными объектами, такими как массивы или объекты. Следующий тест проходит с `assert.deepEqual`:

```
assert.deepEqual([1,2,3],[1,2,3]);
```

но не пройдет с `assert.equal`.

Остальные методы `assert` получают разные наборы параметров. Вызов метода `assert` с передачей значения и сообщения эквивалентен вызову `assert.isEqual` с передачей `true` в первом параметре, выражения и сообщения. Следующий фрагмент:

```
var val = 3;
assert(val == 3, 'Test 1 Not Equal');
```

эквивалентен вызову:

```
assert.equal(true, val == 3, 'Test 1 Not Equal');
```

Также допустимо использование псевдонима `assert.ok`:

```
assert.ok(val == 3, 'Test 1 Not Equal');
```

Метод `assert.fail` выдает исключение. Он получает четыре параметра: значение, выражение, сообщение и оператор, который используется для разделения значения и выражения в генерируемом сообщении (при отсутствии явно заданного). Для следующего фрагмента:

```
try {
  var val = 3;
  assert.fail(val, 4, 'Fails Not Equal', '==');
} catch(e) {
  console.log(e);
}
```

консольный вывод выглядит так:

```
{ [AssertionError: Fails Not Equal]
  name: 'AssertionError',
  actual: 3,
  expected: 4,
  operator: '==',
  message: 'Fails Not Equal',
  generatedMessage: false }
```

Функция `assert.ifError` получает значение и генерирует исключение только в том случае, если значение не эквивалентно `false`. Как указано в документации Node, она хорошо подходит для проверки объекта ошибки из первого аргумента функции обратного вызова:

```
assert.ifError(err); // Выдается только при истинном значении
```

Последние методы assert — `assert.throws` и `assert.doesNotThrow`. Первый метод ожидает, что будет выдано исключение, второй — что исключения не будет. Оба метода получают программный блок в первом обязательном параметре и необязательную ошибку и сообщение во втором и третьем параметрах. Объект ошибки может быть конструктором, регулярным выражением или проверочной функцией. В следующем фрагменте кода будет выведено сообщение об ошибке, потому что регулярное выражение ошибки во втором параметре не соответствует сообщению:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /.../
);
```

С помощью модуля `Assert` можно создавать надежные модульные тесты. Единственным недостатком модуля можно считать то, что вам придется принимать меры к тому, чтобы неудачная проверка одного теста не приводила к сбою всего тестового сценария. В таких ситуациях на помощь приходят фреймворки модульного тестирования более высокого уровня, например Nodeunit.

Модульное тестирование с использованием Nodeunit

`Nodeunit` предоставляет способ сценарного программирования тестов. Все запрограммированные тесты выполняются последовательно, а вывод результатов координируется. Чтобы использовать модуль `Nodeunit`, установите его глобально с использованием `npm`:

```
[sudo] npm install nodeunit -g
```

Модуль `Nodeunit` позволяет легко выполнить серию тестов без того, чтобы упаковывать все подряд в блоки `try/catch`. Он поддерживает все тесты модуля `Assert` и предоставляет пару собственных методов для управления тестами. Тесты объединяются в *тестовые наборы*, каждый из которых экспортируется как метод объекта в тестовом сценарии. Каждый тестовый набор представляется управляющим объектом, которому обычно присваивается имя `test`. В тестовом наборе первым вызывается метод `expect`, который сообщает `Nodeunit`, сколько тестов ожидать в этом случае. Последним в тестовом наборе вызывается метод `done` элемента `test`, который сообщает `Nodeunit` о завершении тестового набора. Все, что находится между этими вызовами, составляет сам тестовый набор:

```
module.exports = {
   'Test 1' : function(test) {
      test.expect(3); // Три теста
      ... // Тесты
      test.done();
   },
   'Test 2' : function (test) {
      test.expect(1); // Только один тест
      ... // Тест
      test.done();
   }
};
```

Чтобы выполнить тесты, введите команды `nodeunit` с именем тестового сценария:

```
nodeunit thetest.js
```

В листинге 11.1 представлен небольшой, но законченный тестовый сценарий с шестью тестовыми утверждениями. Он состоит из двух тестовых наборов с метками `Test 1` и `Test 2`. В первом тестовом наборе выполняются четыре отдельных теста, а во втором — два. Вызов метода `expect` отражает количество тестов, выполняемых в блоке.

Листинг 11.1. Тестовый сценарий Nodeunit с двумя тестовыми наборами (всего шесть тестов)

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

module.exports = {
    'Test 1' : function(test) {
        test.expect(4);
        test.equal(true, util.isArray([]));
        test.equal(true, util.isArray(new Array(3)));
        test.equal(true, util.isArray([1,2,3]));
        test.notEqual(true, 1 > 2);
        test.done();
    },
    'Test 2' : function(test) {
        test.expect(2);
        test.deepEqual([1,2,3], [1,2,3]);
        test.ok('str' === 'str', 'equal');
        test.done();
    }
};
```

Результат выполнения тестового сценария из листинга 11.1 выглядит так:

```
thetest.js
✓ Test 1
✓ Test 2

OK: 6 assertions (12ms)
```

Символы перед тестами сообщают об успехе или неудаче теста: «галочка» — успех, «крест» — неудача. В этом сценарии все тесты прошли успешно, поэтому в вывод не включается информация об ошибке или данные трассировки стека.

Для поклонников CoffeeScript: Nodeunit поддерживает приложения CoffeeScript.

Другие фреймворки тестирования

Кроме фреймворка Nodeunit, упомянутого в предыдущем разделе, существует целый ряд тестовых фреймворков для разработчиков Node. Одни инструменты проще, другие сложнее, у каждого есть свои достоинства и недостатки. Сейчас я кратко представлю три таких фреймворка: Mocha, Jasmine и Vows.

Mocha

Установите Mocha при помощи `npm`:

```
npm install mocha -g
```

Mocha считается наследником другого популярного фреймворка тестирования, Espresso. Mocha работает как в браузерах, так и в приложениях Node. Функция `done` обеспечивает возможность асинхронного тестирования, хотя тестирование может производиться и в синхронном режиме без нее. Mocha может использоваться в сочетании с любой библиотекой поддержки тестовых утверждений.

Пример теста Mocha с использованием `Assert`:

```
assert = require('assert')
describe('MyTest', function() {
  describe('First', function() {
    it('sample test', function() {
      assert.equal('hello','hello');
    });
  });
});
```

Запустите тест следующей командой:

```
mocha testcase.js
```

Тест должен пройти успешно:

```
  MyTest
    First
      ✓ sample test
  1 passing (15ms)
```

Vows

У Vows — тестового фреймворка разработки через поведение (BDD, Behavior-Driven Development) — есть одно важное преимущество: более подробная документация. Процесс тестирования состоит из *комплексов тестов*, а эти комплексы в свою очередь состоят из пакетов последовательно выполняемых тестов. Пакет (batch) состоит из одного или нескольких контекстов, выполняемых параллельно; каждый составляет *тему* (topic). Тест в программном коде называется *обязательством* (vow). Отличительной особенностью Vows является четкое разделение между тем, что тестируется (тема), и самими тестами (обязательства).

Рассмотрим простой пример того, как работают тесты Vows. Впрочем, сначала необходимо установить Vows:

```
npm install vows
```

Для тестирования Vows я использую простой модуль для вычисления площади круга и длины окружности. Поскольку значения хранятся в вещественном формате, а я проверяю равенство, возвращаемые значения округляются до четырех знаков в дробной части:

```
const PI = Math.PI;

exports.area = function (r) {
  return (PI * r * r).toFixed(4);
};

exports.circumference = function (r) {
  return (2 * PI * r).toFixed(4);
};
```

В тестовом приложении Vows объект круга соответствует теме, а методы `area` и `circumference` — обязательствам. Все это инкапсулируется в контексте `Vows`. Комплекс тестов соответствует всему тестовому приложению, а пакет — тестовому экземпляру (объект круга и два метода).

В листинге 11.2 приведен весь тест.

Листинг 11.2. Тестовое приложение Vows с одним пакетом, одним контекстом, одной темой и двумя обязательствами

```
var vows = require('vows'),
    assert = require('assert');

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

```
var suite = vows.describe('Test Circle');

suite.addBatch({
    'An instance of Circle': {
        topic: circle,
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic.circumference(3.0), 18.8496);
        },
        'should be able to calculate area': function(topic) {
            assert.equal (topic.area(3.0), 28.2743);
        }
    }
}).run();
```

Запуск приложения из Node приводит к запуску теста из-за добавления метода `run` в конце метода `addBatch`:

```
node vowstest.js
```

В результате должно быть два успешных теста:

```
·· ✓ OK " 2 honored (0.012s)
```

Тема всегда представляет собой асинхронную функцию или значение. Вместо использования `circle` как темы я могла бы напрямую обращаться к методам объекта как к темам — с небольшой помощью от функциональных замыканий (closures):

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
    'Testing Circle Circumference': {
        topic: function() { return circle.circumference;},
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic(3.0), 18.8496);
        },
    },
    'Testing Circle Area': {
        topic: function() { return circle.area;},
        'should be able to calculate area': function(topic) {
            assert.equal (topic(3.0), 28.2743);
```

```
        }
    }
}).run();
```

В этой версии примера каждый контекст представляет собой объект с заданным заголовком: `Testing Circle Circumference` и `Testing Circle Area`. В каждом контексте присутствует одна тема и одно обязательство.

Вы можете включить в свой процесс тестирования несколько пакетов, каждый из которых содержит несколько контекстов, в свою очередь состоящих из нескольких тем и нескольких обязательств.

Обеспечение бесперебойной работы Node

Вы сделали все, что могли, с кодом приложения. Вы тщательно протестировали его и добавили обработку ошибок, чтобы приложение корректно восстанавливалось в случае сбоев. Тем не менее неприятности все же возможны — непредвиденные факторы могут вывести приложение из строя. Если это произойдет, необходимо позаботиться о том, чтобы приложение снова запустилось, даже если вас не будет рядом.

Программа Forever решает именно эту задачу — она обеспечивает перезапуск вашего приложения в случае сбоя. Также она позволяет запустить ваше приложение в виде демона, продолжающего существовать после завершения текущего терминального сеанса. Программа Forever выполняется из командной строки или интегрируется в приложение. Чтобы использовать ее из командной строки, установите ее глобально:

```
npm install forever -g
```

Далее, вместо того чтобы запускать приложение с помощью Node, запустите его из Forever:

```
forever start -a -l forever.log -o out.log -e err.log finalserver.js
```

Для двух параметров используются значения по умолчанию: `minUpTime` (1000 мс) и `spinSleepTime` (1000 мс).

Приведенная команда запускает сценарий finalserver.js и задает имена журнала Forever, выходного журнала и журнала ошибок. Она также приказывает

приложению присоединять записи к текущему содержимому журнала, если файлы уже существуют.

Если со сценарием происходит нечто такое, что приводит к его аварийному завершению, Forever перезапускает его. Также Forever следит за тем, чтобы приложение Node продолжало работать, даже если терминальное окно, использованное для запуска приложения, было закрыто.

В командной строке Forever могут передаваться как управляющие значения, так и команды. Примером команды служит значение `start` в только что приведенной командной строке. Полный список доступных действий:

- `start` — запуск сценария;
- `stop` — остановка сценария;
- `stopall` — остановка всех сценариев;
- `restart` — перезапуск сценария;
- `restartall` — перезапуск всех сценариев Forever;
- `cleanlogs` — удаление всех записей в журнале;
- `logs` — вывод списка журнальных файлов для всех процессов Forever;
- `list` — вывод списка всех выполняемых сценариев;
- `config` — вывод списка пользовательских конфигураций;
- `set <`*`ключ`*`> <`*`значение`*`>` — присваивание значения ключу конфигурации;
- `clear <`*`ключ`*`>` — сброс значения ключа конфигурации;
- `logs <`*`сценарий`*`|`*`индекс`*`>` — вывод завершающей части журналов для параметров `<`*`сценарий`*`|`*`индекс`*`>`;
- `columns add <`*`столбец`*`>` — добавление столбца в выходном списке Forever;
- `columns rm <`*`столбец`*`>` — удаление столбца из выходного списка Forever;
- `columns set <`*`столбец`*`>` — установка всех столбцов в выходном списке Forever.

Пример выходного списка после запуска httpserver.js в режиме демона Forever:

```
info:    Forever processes running
data:        uid  command         script         forever pid   id
logfile                            uptime
data:    [0] _gEN /usr/bin/nodejs serverfinal.js 10216   10225
/home/name/.forever/forever.log STOPPED
```

Вывод списка журналов командой `logs`:

```
info:    Logs for running Forever processes
data:        script         logfile
data:    [0] serverfinal.js /home/name/.forever/forever.log
```

Forever также поддерживает обширный набор управляющих значений, включая только что представленные настройки файлов журналов, режим запуска сценария (`-s` или `--silent`), режим подробного вывода Forever (`-v` или `--verbose`), назначение исходного каталога (`--sourceDir`) и множество других. Чтобы получить информацию о них, введите команду:

```
forever --help
```

Использование Forever можно встроить прямо в код приложения при помощи вспомогательного модуля `forever-monitor`, как показано в документации этого модуля:

```
var forever = require('forever-monitor');

  var child = new (forever.Monitor)('serverfinal.js', {
    max: 3,
    silent: true,
    args: []
  });

  child.on('exit', function () {
    console.log('serverfinal.js has exited after 3 restarts');
  });

  child.start();
```

Кроме того, Forever можно использовать с Nodemon не только для перезапуска приложения в случае неожиданного сбоя, но и для того, чтобы обеспечить обновление приложения в случае обновления исходного кода. Выполните глобальную установку Nodemon:

```
npm install -g nodemon
```

Nodemon создает обертку для вашего приложения. Вместо Node для запуска приложения следует использовать Nodemon:

```
nodemon app.js
```

Nodemon незаметно для пользователя отслеживает содержимое каталога, из которого было запущено приложение (и всех содержащихся в нем каталогов), наблюдая за изменением файлов. При обнаружении изменения Nodemon перезапускает приложение с учетом новейших изменений.

Приложению могут передаваться параметры:

```
nodemon app.js param1 param2
```

Также возможно использование модуля с CoffeeScript:

```
nodemon someapp.coffee
```

Если вы хотите, чтобы вместо текущего каталога отслеживался какой-то другой каталог, используйте флаг `--watch`:

```
nodemon --watch dir1 --watch libs app.js
```

Также поддерживаются другие флаги, описанные в документации модуля (*https://github.com/remy/nodemon/*).

Чтобы использовать Nodemon с Forever, создайте для Nodemon «обертку» Forever и укажите параметр `--exitcrash`. В этом случае при сбое приложения Nodemon завершится корректно и передаст управление Forever:

```
forever start nodemon --exitcrash serverfinal.js
```

Если вы получите сообщение о том, что Forever не находит Nodemon, укажите полный путь:

```
forever start /usr/bin/nodemon --exitcrash serverfinal.js
```

Если в ходе работы приложения произойдет сбой, Forever запустит Nodemon, а Nodemon, в свою очередь, запустит сценарий Node, который не только обеспечивает обновление выполняемого сценария при изменении исходного кода, но и гарантирует, что приложение продолжит работу и после незапланированного сбоя.

Эталонные тесты и нагрузочное тестирование с использованием Apache Bench

Даже у самого надежного приложения, удовлетворяющего всем потребностям пользователей, нет никаких перспектив, если оно работает с низкой производительностью. Разработчику необходимы средства для тестирования производительности приложений Node, — особенно если он вносит изменения, направленные на улучшение производительности. Мы не можем просто «подстроить» что-то в приложении, передать его пользователям и дожидаться сообщений о проблемах производительности.

Тестирование производительности состоит из эталонного тестирования и нагрузочного тестирования. *Эталонное тестирование* (benchmarking), также называемое сравнительным тестированием, основано на запуске нескольких версий или модификаций приложения и определении того, какая из них лучше. Это эффективный инструмент оптимизации приложения, призванной улучшить его быстродействие и масштабируемость. Вы создаете стандартизированный тест, запускаете его вместе с несколькими модификациями, а затем анализируете результаты.

С другой стороны, в ходе нагрузочного тестирования вы пытаетесь узнать, в какой момент ваше приложение начинает «тормозить» или сбоить из-за чрезмерно высокой нагрузки на ресурсы или количества одновременно работающих пользователей. Фактически приложение сознательно доводится до сбоя. В нагрузочном тестировании сбой становится признаком успеха.

Существует много готовых инструментов, реализующих обе разновидности тестирования производительности; особой популярностью пользуется ApacheBench. Отчасти это объясняется тем, что он по умолчанию присутствует на всех серверах с установкой Apache, то есть практически на любом сервере. Кроме того, это простой, мощный и компактный инструмент тестирования. В частности, когда я выясняла, что лучше — создать статическое подключение к базе данных или создать подключение и уничтожать его после каждого использования, для тестирования использовался пакет ApacheBench.

Название ApacheBench часто сокращается до «ab», и в дальнейшем я буду использовать именно этот вариант. ab — утилита командной строки, позволяющая задать количество запусков приложения и количество параллельных пользователей. Например, если вы хотите эмулировать 20 параллельных

пользователей, суммарно обращающихся к веб-приложению 100 раз, используйте следующую команду:

```
ab -n 100 -c 20 http://burningbird.net/
```

Слеш в конце команды важен, так как ab ожидает получить полный URL-адрес, включающий путь.

ab выводит достаточно разнообразную информацию. Пример выходных данных одного теста (без идентификационных данных):

```
Benchmarking burningbird.net (be patient).....done

Server Software:        Apache/2.4.7
Server Hostname:        burningbird.net
Server Port:            80

Document Path:          /
Document Length:        36683 bytes

Concurrency Level:      20
Time taken for tests:   5.489 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      3695600 bytes
HTML transferred:       3668300 bytes
Requests per second:    18.22 [#/sec] (mean)
Time per request:       1097.787 [ms] (mean)
Time per request:       54.889 [ms] (mean, across all concurrent requests)
Transfer rate:          657.50 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   2.3      0       7
Processing:   555 1049 196.9   1078    1455
Waiting:       53  421 170.8    404     870
Total:        559 1050 197.0   1080    1462

Percentage of the requests served within a certain time (ms)
  50%   1080
  66%   1142
  75%   1198
  80%   1214
  90%   1341
  95%   1392
  98%   1415
  99%   1462
 100%   1462 (longest request)
```

В этом выводе нас прежде всего интересуют строки с продолжительностью выполнения каждого теста и распределение вероятностей в конце теста (основанное на процентных соотношениях). В соответствии с приведенными данными среднее время обработки запроса (первая выделенная строка) составляет 1097,787 миллисекунды — столько времени пользователю придется в среднем дожидаться ответа. Вторая выделенная строка относится к пропускной способности сервера и, скорее всего, не так полезна, как первая.

Накопительное распределение позволяет понять, какой процент запросов обрабатывается за некоторый временной интервал. Еще раз подчеркну, что данные показывают, чего можно ожидать в среднем случае: время отклика лежит в интервале от 1080 до 1462 миллисекунд, причем в подавляющем большинстве случаев оно составляет 1392 миллисекунды и менее.

Остается еще одно значение, которое нас интересует, — количество запросов в секунду (в данном случае 18,22). По этому значению иногда можно спрогнозировать, как будет масштабироваться приложение, потому что оно дает представление о максимальном количестве запросов в секунду, то есть о верхней границе количества обращений. Однако тесты следует проводить в разное время и под разной сторонней нагрузкой, особенно если тестирование проводится в системе, выполняющей другие функции.

ПРИЛОЖЕНИЕ LOADTEST

Для проведения нагрузочного тестирования также можно воспользоваться приложением Loadtest:

```
npm install -g loadtest
```

Его преимущество перед Apache Bench заключается в том, что Loadtest позволяет настраивать не только пользователей, но и запросы:

```
loadtest [-n запросы] [-c параллелизм] [-k] URL
```

12 Node в других средах

Поддержка Node появилась во многих средах за пределами базовых серверов для Linux, OS X и Windows.

Node позволяет использовать JavaScript на микрокомпьютерах и микроконтроллерах, таких как Raspberry Pi и Arduino. Компания Samsung планирует интегрировать Node в свою концепцию «Интернета вещей» (IoT, Internet of Things), при том что купленная ею технология IoT — SmartThings — основана на разновидности Java (Groovy). Компания Microsoft тоже использовала разновидность Node в своем ядре Chakra.

Многогранность Node — то, что делает эту технологию такой интересной и занимательной.

NODE И МОБИЛЬНЫЕ ПЛАТФОРМЫ

Технология Node нашла место и в мобильном мире. Впрочем, даже если бы я постаралась сжать тему до минимума, мне все равно не удалось бы выделить под нее раздел книги. Вместо этого я порекомендую читателю книгу по этой теме: «Learning Node.js for Mobile Application Development» (Packt, 2015), авторы Стефан Баттиджиг и Милорад Евженич.

Samsung IoT и GPIO

Компания Samsung создала разновидность Node, называемую IoT.js, а также версию JavaScript для технологий IoT, которая называется JerryScript. Из документации следует, что это было сделано прежде всего для разработки инструментов и технологий, работающих на устройствах с меньшим объемом памяти по сравнению с традиционными средами JavaScript/Node.

В одной презентации работника Samsung (*http://www.soscon.net/download/day28/GB2/S_28_1050_%EC%9D%B4%EC%B6%98%EC%84%9D.pdf*) представлена полная реализация JerryScript с размером двоичного файла 200 Кбайт и затратами памяти от 16 до 64 Кбайт. Сравните с реализацией V8, требующей двоичного файла размером 10 Мбайт и затрат памяти 8 Мбайт. На устройствах IoT каждый байт имеет значение.

В документации IoT.js указано, что эта реализация поддерживает подмножество базовых модулей Node: в него входят такие модули, как `Buffer`, `HTTP`, `Net` и `File System`. С учетом условий применения вполне понятно отсутствие поддержки таких модулей, как `Crypto`. Также добавился новый базовый модуль `GPIO`, относящийся к специфике IoT: он представляет интерфейс между приложениями и физическим оборудованием.

GPIO — сокращение от «ввод/вывод общего назначения» (General-Purpose Input/Output). Представьте контакт на микросхеме, который может выдавать как входной, так и выходной сигнал и поведением которого управляют создаваемые нами приложения. Контакты GPIO образуют интерфейс к устройству. Входные контакты могут получать информацию с таких устройств, как датчики температуры или движения; выходные контакты могут управлять освещением, сенсорными экранами, моторами, вентиляторами и т. д.

На таких устройствах, как Raspberry Pi (см. «Node для микроконтроллеров и микрокомпьютеров», с. 284), с одной стороны размещается контактная панель. Многие контакты являются контактами GPIO, они чередуются с контактами питания и заземления. На рис. 12.1 представлена фотография самих контактов, а под ними — схема разводки с обозначением контактов GPIO, питания и заземления для Raspberry Pi 2 Model B.

Как можно заметить, номер контакта на схеме разводки не соответствует его физическому расположению на плате. Число рядом с контактом определяет его *номер GPIO*. Некоторые API, включая IoT.js компании Samsung, под номером контакта понимают номер GPIO.

Чтобы использовать Samsung IoT.js, разработчик инициализирует объект GPIO и вызывает одну из его функций, например функцию `gpio.setPen()`, которая получает в первом параметре номер контакта, направление (`'in'` для ввода, `'out'` для вывода, `'none'` для отпускания), необязательный спецификатор режима и функцию обратного вызова. Для отправки данных на контакт используется функция `gpio.writePin()` с номером контакта, логическим значением и функцией обратного вызова.

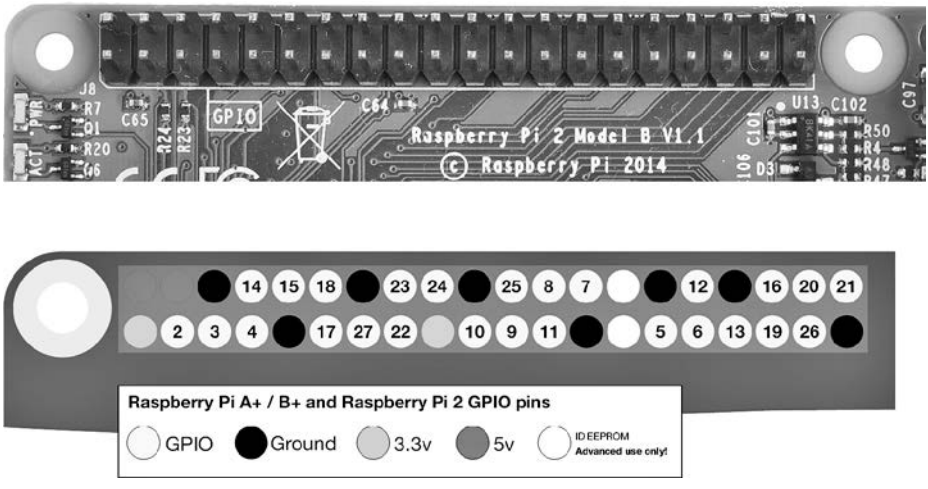


Рис. 12.1. Контакты Raspberry Pi 2 и схема разводки (с разрешения Raspberry Pi Foundation, используется на условиях лицензии CC CC-BY-SA)

Технология Samsung IoT.js определенно находится на стадии разработки. И похоже, она вступает в конфликт с другой технологией Samsung, известной как SAMIO (*https://developer.samsungsami.io/*); эта технология определяет платформу обмена данными, которая может (среди прочего) обеспечивать обмен данными между Arduino и Raspberry Pi при управлении пожарной сигнализацией (см. обучающее руководство *https://developer.samsungsami.io/sami/tutorials/your-first-iot-device.html*), и она тоже интегрирует Node. Тем не менее все эти технологии находятся в активной разработке. Вероятно, пока нам просто стоит следить за новостями.

RASPBERRY PI И ARDUINO

Raspberry Pi и Arduino более подробно рассматриваются в разделе «Node для микроконтроллеров и микрокомпьютеров», с. 284.

Windows с Chakra Node

19 января 2016 года компания Microsoft направила запрос на включение (PR, Pull Request), обеспечивающий работу Node с JavaScript-ядром Microsoft ChakraCore. Компания создала прослойку V8, которая реализует большинство важнейших API V8 и предоставляет возможность прозрачного использования Node с ChakraCore.

Идея работы Node на чем-то, кроме V8, выглядит интересно и, на мой взгляд, довольно привлекательно. Хотя на первый взгляд V8 задает направление разработки Node (по крайней мере, текущих версий), с технической точки зрения нет никаких обязательных требований о том, чтобы технология Node строилась на базе V8. В конце концов, текущие обновления в большей степени направлены на совершенствование Node API и внедрение новшеств ECMAScript.

Компания Microsoft перевела ChakraCore на модель открытого исходного кода; это было необходимым первым шагом. ChakraCore обеспечивает поддержку ядра JavaScript для нового браузера Edge. И компания утверждает, что ее ядро превосходит V8.

И хотя тестирование и обсуждения еще не закончены, вы можете протестировать Node с ChakraCore уже сейчас; для этого вам понадобится машина с системой Windows, Python (2.6 или 2.7) и Visual Studio (например, версия Visual Studio 2015 Community version, доступная для бесплатной загрузки). Также можно загрузить готовый двоичный файл. Компания Microsoft предоставляет двоичные файлы для архитектуры ARM (Raspberry Pi), а также для более традиционных архитектур x86 и x64. Когда я установила пакет на своем компьютере с Windows, он создал такое же командное окно, которое я использовала для работы с Node без ChakraCore. Но самое замечательное, что его можно установить на одной машине с установкой Node и две системы могут использоваться параллельно.

Я попыталась запустить несколько примеров из книги, никаких проблем при этом не возникло. Я также опробовала пример, в котором отражено недавнее изменение в Node, позволяющее функции `child_process.spawn()` получать параметр `shell` (см. главу 8). Пример не работал с LTS-версией 4.x, но работает с новейшей текущей версией (6.0.0 на момент написания книги). И он работал с двоичным файлом сборки Node на базе ChakraCore — даже при том, что новая текущая версия была выпущена всего за несколько дней до этого. Таким образом, компания Microsoft включает в свою двоичную сборку Node/ChakraCore новейшую текущую версию Node API.

Еще более интересный результат был получен тогда, когда я попыталась выполнить следующий пример ES6:

```
'use strict'

// Приводится с разрешения доктора Акселя Раушмайера (Axel Rauschmayer)
// http://www.2ality.com/2014/12/es6-proxies.html

let target = {};
```

```
    let handler = {
        get(target, propKey, receiver) {
            console.log('get ' + propKey);
            return 123;
        }
    };

let proxy = new Proxy(target, handler);

console.log(proxy.foo);

proxy.bar = 'abc';
console.log(target.bar);
```

Пример работал с ChakraCore Node, но не работал с V8 Node, даже в самой последней версии. Также разработчики ChakraCore утверждают, что они реализуют более совершенную версию новых расширений ECMAScript.

Пример не работал со стабильной версией V5, но работал в более новой текущей версии V6.

В настоящее время ChakraCore работает только в системе Windows; компания Microsoft также предоставляет двоичный файл для Raspberry Pi и обещает скоро выпустить версию для Linux.

Node для микроконтроллеров и микрокомпьютеров

Технология Node также уверенно обосновалась на микроконтроллерах типа Arduino и микрокомпьютерах, таких как Raspberry Pi.

Говоря о Raspberry Pi, я использую термин «микрокомпьютер», но в действительности это полностью функциональный (хотя и маленький) компьютер. На нем можно установить операционную систему, например Windows 10 или Linux; подключить клавиатуру, мышь и монитор; запускать разные приложения: браузер, игры, офисные приложения и т. д. С другой стороны, Arduino используется для повторяющихся задач. Вместо того чтобы подключать клавиатуру или мышь напрямую к устройству, вы подключаете устройство к компьютеру и используете сопутствующее приложение для построения и передачи программы на устройство. Это *встроенный компьютер* — в отличие от обычных компьютеров (таких, как Raspberry Pi).

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ RASPBERRY PI И ARDUINO

Никто не заставляет вас выбирать между Raspberry Pi и Arduino; вы можете использовать их вместе. Pi становится «мозгом», а Arduino — «мускулами» системы.

Arduino и Raspberry Pi популярны в своих категориях, но это всего лишь часть доступных устройств, многие из которых Arduino-совместимы. В их число даже входит носимое устройство (LilyPad).

Если вы недавно занимаетесь разработкой для IoT и подключенных устройств, я рекомендую начать с Arduino Uno, а затем опробовать Raspberry Pi 2. Наборы продаются на таких сайтах, как AdaFruit, SparkFun, Cana Kit, Amazon, Maker Shed и других, а также у самих производителей плат. Наборы включают все необходимое для начала работы с платами и стоят недорого (менее $100). В них включены книги проектов и компоненты, необходимые для нескольких проектов.

В этом разделе я покажу, как создать простейшее приложение для устройств такого типа. Это приложение обращается к светодиодному индикатору и выдает световые импульсы. Я продемонстрирую приложение для Arduino Uno и Raspberry Pi 2.

НОВАЯ МОДЕЛЬ RASPBERRY 3

Когда я уже завершала работу над книгой, была выпущена новая модель Raspberry Pi 3. Пример, приведенный позднее для Raspberry Pi, должен легко преобразоваться для нового устройства. А самое замечательное, что Raspberry Pi 3 содержит встроенный модуль WiFi.

Но чтобы добиться хоть каких-то результатов в мире подключенных устройств, необходимо немного знать об электронике и о замечательном инструменте, который называется Fritzing.

Fritzing

Fritzing — программа с открытым кодом, предоставляющая разработчикам средства для построения прототипов и последующего физического построения спроектированной схемы в Fritzing Fab. Программа распространяется бесплатно (хотя я рекомендую внести пожертвование, если она покажется вам полезной). Все графические диаграммы, встречающиеся в проектах Arduino и Raspberry Pi, неизменно создаются в приложении Fritzing.

ГДЕ ЗАГРУЗИТЬ ПРИЛОЖЕНИЕ

Приложение Fritzing можно загрузить на сайте Fritzing (*http://fritzing.org/home/*). Приложение достаточно большое, что неудивительно с учетом количества графических компонентов, необходимых для приложения. Вы можете загрузить версию для Windows, OS X или Linux.

Когда вы открываете программу и создаете в ней новый *скетч* (как в Fritzing называется проектная схема), в него автоматически добавляется *макетная плата*. Макетная плата позволяет легко спроектировать прототип электронного проекта. Паять ничего не нужно; все, что от вас требуется, — подключить один конец провода к контакту на плате, а другой — к макетной плате. На пластиковой поверхности макетной платы расположены полоски токопроводящего металла: длинные вертикальные под шиной питания, если они расположены на плате (между линиями на рис. 12.2), и более короткие горизонтальные под каждым столбцом (рис. 12.3). Затем компоненты проекта помещаются на макетную плату.

АНАТОМИЯ МАКЕТНОЙ ПЛАТЫ

За более подробной информацией о макетной плате, ее истории и принципах работы я рекомендую обращаться к обучающему руководству SparkFun «How to Use a Breadboard» (*https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard*).

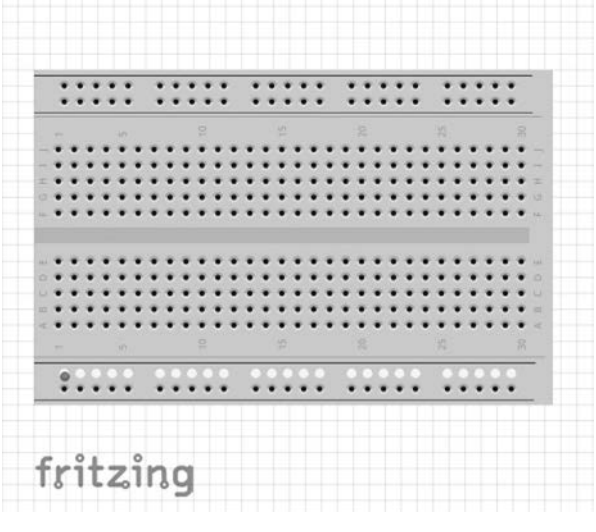


Рис. 12.2. Макетная плата с выделением контактов шины питания

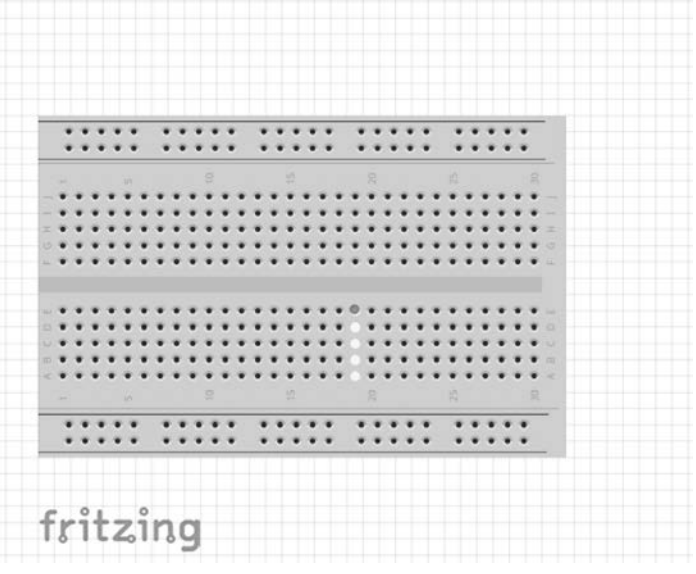


Рис. 12.3. Макетная плата с выделением контактной полоски

Используемые компоненты перетаскиваются из палитры, расположенной справа. На рис. 12.4 изображен скетч, который я создала для проекта с ми-

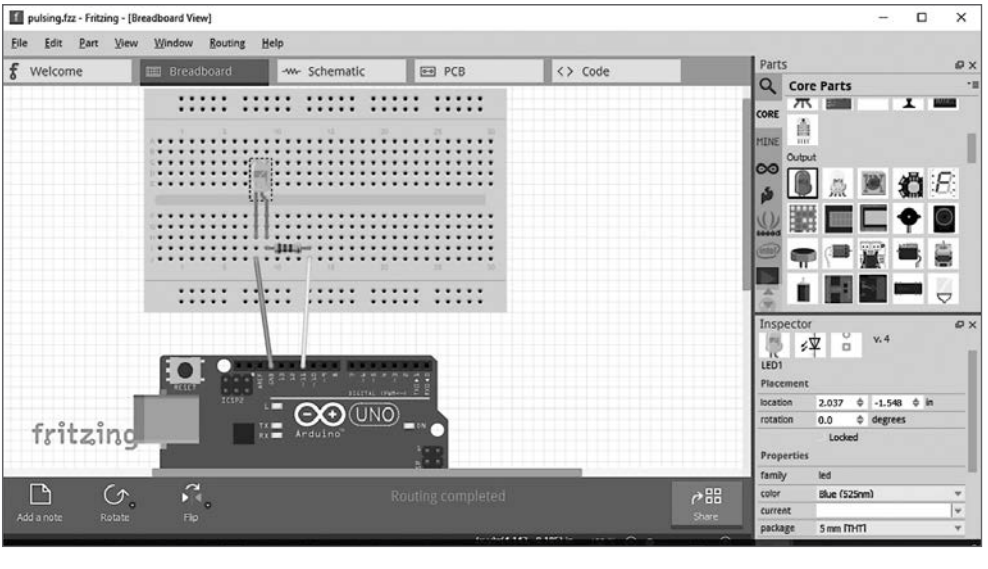


Рис. 12.4. Эскиз проекта управления светодиодом на базе Arduino в Fritzing

гающим светодиодом на базе Arduino из следующего раздела. Он состоит из компонента Arduino Uno, макетной платы половинного размера, двух соединительных проводов, резистора и светодиода из того же списка компонентов, цвет которого был изменен с красного на синий. В правом нижнем углу отображается описание компонента. В этом разделе можно вручную отредактировать информацию обо всем компонентах, включая макетную плату (изменить полный размер на половинный).

Чтобы создать проводное соединение, перетащите указатель мыши с контакта Arduino в то место, где провод должен заканчиваться на макетной плате. Программа автоматически создает провод. Добавьте нужные компоненты, перетаскивая их из меню компонентов в скетч. Чтобы схема была точной и полной, вы также можете просмотреть схематическое представление проекта (рис. 12.5).

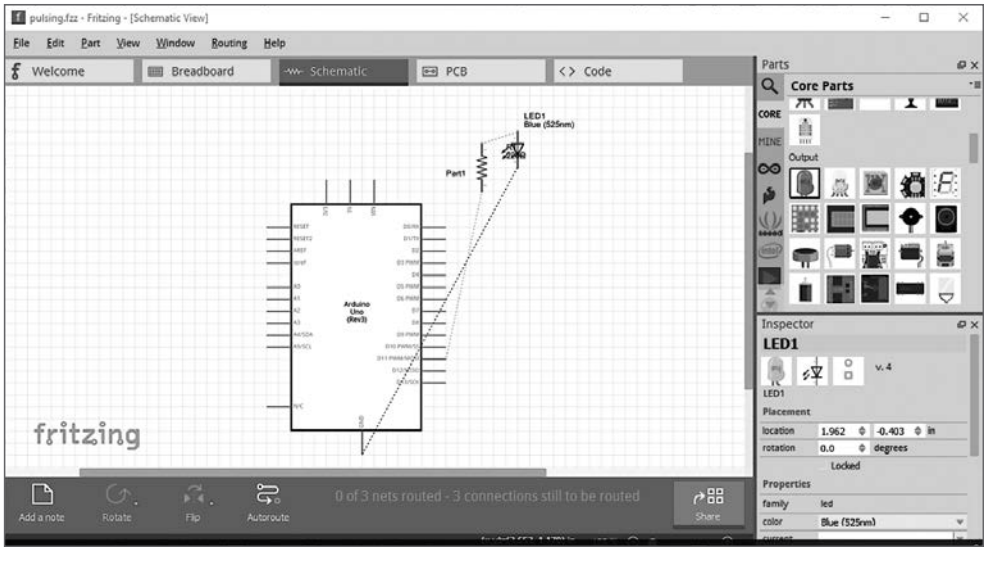


Рис. 12.5. Схематическая диаграмма проекта управления светодиодом на базе Arduino

У светодиода две ножки, одна длиннее другой. На диаграмме более длинная ножка изображена изогнутой. Более длинная ножка соответствует аноду (положительному электроду), а более короткая — катоду (отрицательному электроду). На скетче отрицательный электрод подключается к заземлению на Arduino, тогда как положительный электрод подключается к контакту 11.

Один провод соединяет контакт заземления (GND) с платой. Светодиод размещается на плате, при этом катод (короткая ножка) находится в одном столбце с проводом от заземления. Резистор тоже размещается на плате, один контакт подключается к одному столбцу с анодом светодиода. Наконец, добавляется провод, соединяющий другой контакт резистора с контактом 11 на Arduino; построение схемы на этом завершается.

Резистор играет важную роль в этой схеме. Резисторы, то есть сопротивления, делают то, что следует из их названия: они сопротивляются прохождению электрического тока. Если не добавить резистор в эту схему, через светодиод может пройти слишком сильный ток, а это приведет к повреждению контакта GPIO и платы.

Сопротивление измеряется в омах (Ω); резистор в этом эскизе имеет сопротивление 220 Ом. Сопротивление резистора можно определить по его цветовой маркировке. Цвет колец (и их количество) определяет тип резистора, включенного в схему.

ЗАКОН ОМА

Как определить, какой резистор следует использовать в конкретном случае? По формуле, называемой «законом Ома». Да, этот закон, который вы изучали в школе, можно применить на практике. За дополнительной информацией о законе Ома и выборе резисторов обращайтесь к статьям «Calculating correct resistor value to protect Arduino pin» (*http://hardwarefun.com/tutorials/calculating-correct-resistor-value-to-protect-arduino-pin*) и «Do I really need resistors when controlling LEDs with Arduino?» (*http://electronics.stackexchange.com/questions/32990/do-i-really-need-resistors-when-controlling-leds-with-arduino*).

Светодиод должен размещаться на плате в определенной ориентации: катод присоединяется к заземлению. У резистора направленности нет, он может располагаться на плате в любой ориентации. Кроме того, резистор не обязательно размещать после светодиода; как видно из рис. 12.2, с таким же успехом его можно разместить перед светодиодом. Так как это очень простая схема, в данном случае важно лишь, чтобы цепь содержала резистор. Эскиз Fritzing для проекта Raspberry Pi из раздела «Node и Raspberry Pi 2» на с. 298, в котором резистор размещается перед светодиодом, показан на рис. 12.6.

Наше знакомство с основными компонентами электрических схем было очень кратким, но для примеров в двух следующих разделах этого вполне достаточно.

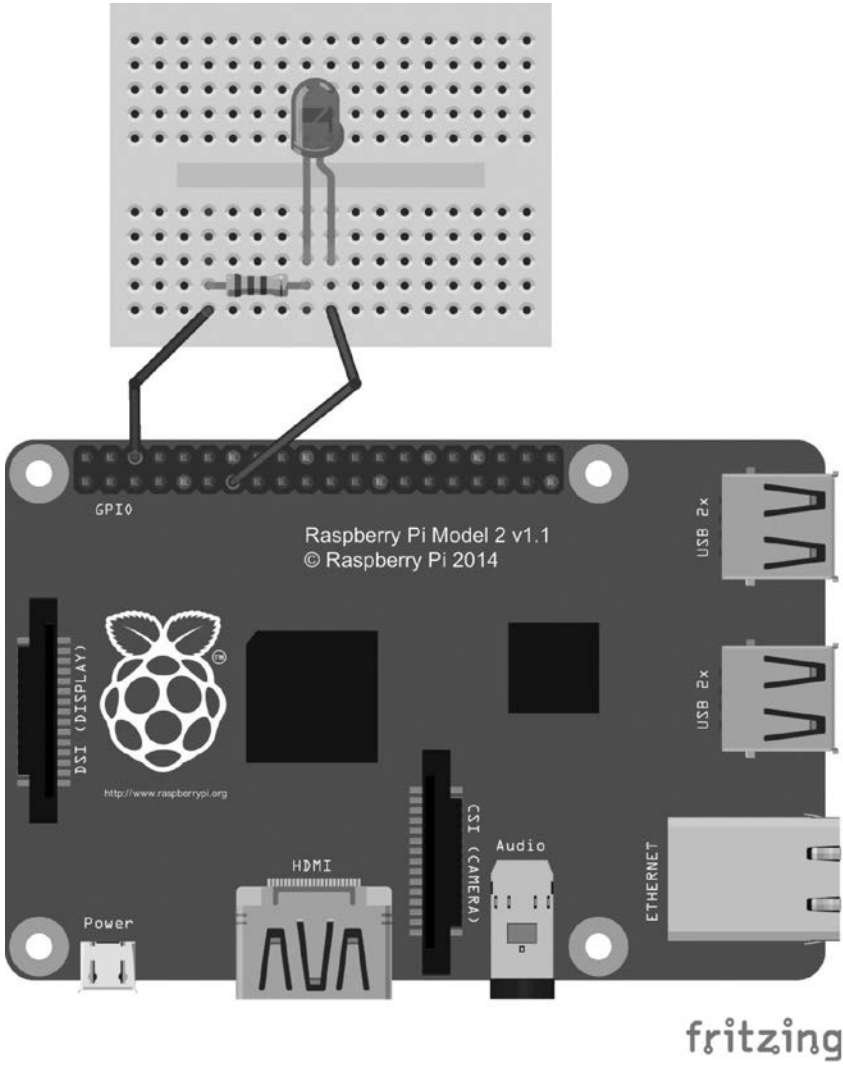


Рис. 12.6. Эскиз Fritzing для проекта управления светодиодом

Node и Arduino

Чтобы заниматься программированием Arduino Uno, необходимо установить на компьютере программу Arduino. В этом примере я использовала версию 1.7.8 на машине с Windows 10. Также существуют версии для OS X и Linux. Подробные, качественные инструкции по установке можно найти на сайте Arduino (*https://www.arduino.cc/en/Guide/HomePage*). После установки

подключите Arduino к PC кабелем USB и запомните последовательный порт, который использовался для подключения (в моем случае COM3).

Затем для использования Node необходимо отправить на Arduino реализацию протокола Firmata для обмена данными с микроконтроллером через программное обеспечение на компьютере. В приложении Arduino выберите команду File▸Examples▸Firmata▸StandardFirmata. На рис. 12.7 показана реализация Firmata, загруженная в приложении. В верхней части окна находится кнопка со стрелкой, указывающей вправо. Щелкните на ней, чтобы передать реализацию Firmata в Arduino.

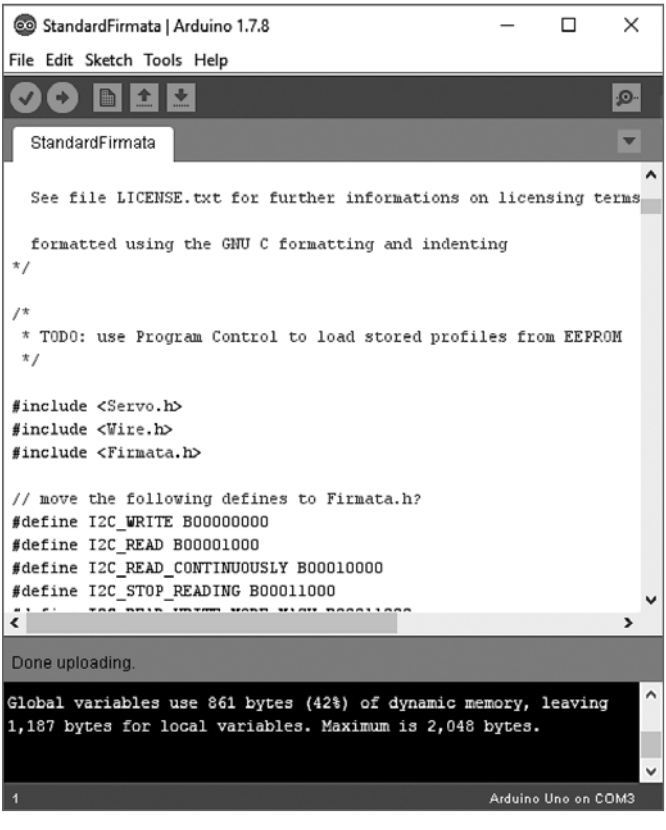


Рис. 12.7. Для работы Node необходимо загрузить в Arduino реализацию протокола Firmata

Node и JavaScript не являются стандартным (или просто самым распространенным) способом управления Arduino или Raspberry Pi (Python — популярный вариант). Впрочем, для программирования таких устройств существует

фреймворк Node, который называется Johnny-Five (*http://johnny-five.io/*). Установите его в Node следующей командой:

```
npm install johnny-five
```

На сайте Johnny-Five приведено не только описание обширного API, используемого для управления платой, но и ряд примеров, включая простейшее приложение управления светодиодом. В отличие от примера на сайте Johnny-Five, в котором светодиод подключается прямо к контактам платы Arduino Uno, мы будем управлять светодиодом, встроенным в плату. Для обращения к этому светодиоду используется контакт с номером 13.

Код мигания встроенным светодиодом (или внешним светодиодом, присоединенным к контакту 13) выглядит так:

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  var led = new five.Led(13);
  led.blink(500);
});
```

Приложение загружает модуль Johnny-Five и создает новый объект, представляющий Arduino. Когда объект платы будет готов, приложение создает новый объект светодиода `Led` с использованием номера контакта 13. Обратите внимание: этот номер не совпадает с номером GPIO; Johnny-Five использует физическую систему нумерации, отражающую местонахождение контактов на плате. После создания объекта, представляющего светодиод, вызывается его функция `blink()`. Результат показан на рис. 12.8 (стрелка указывает на светодиод).

Приложение открывает сеанс REPL. Чтобы завершить приложение, введите команду `.exit`. Светодиод продолжит мигать до тех пор, пока вы не завершите приложение; если светодиод горит в момент выхода, он так и останется включенным до выключения питания устройства.

Мигающий светодиод — это, конечно, хорошо, но раз уж мы занялись этой новой игрушкой, можно придумать что-нибудь поинтереснее. Объект светодиода поддерживает несколько функций с завлекательными именами типа `pulse()` и `fadeIn()`. Однако для этих функций необходим контакт широтно-импульсной модуляции (PWM, Pulse Width Modulation), также называемый *аналоговым выводом*. Контакт 13 таковым не является. С другой стороны, контакт 11 относится к этой категории, на что указывает тильда (~) перед номером на плате (~11).

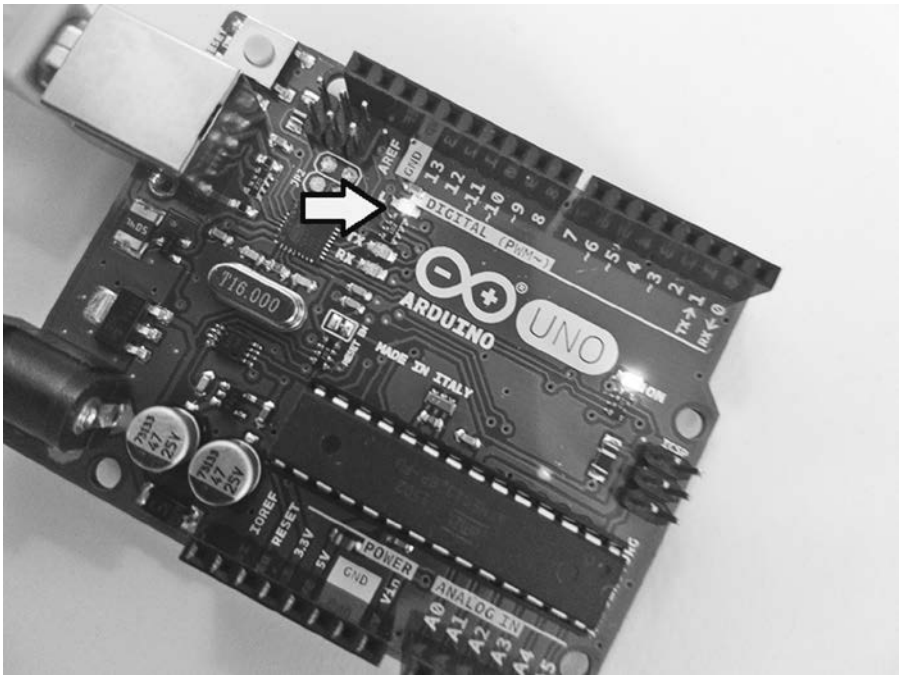


Рис. 12.8. Мигание встроенного светодиода на контакте 13 в Arduino Uno

БЕЗОПАСНОСТЬ, БЕЗОПАСНОСТЬ

IoT — это очень весело и забавно, пока вы не спалите свою плату. Или компьютер.

На самом деле вы можете безопасно и просто работать с большинством приложений IoT при условии соблюдения инструкций. Вот почему так важно всегда проверять диаграммы при размещении компонентов и всегда использовать правильные компоненты.

Также очень важно поддерживать чистоту и порядок на рабочем месте, использовать хороший мощный источник света, не ходить по нейлоновому ковру в носках перед тем, как прикасаться к электронным компонентам, отсоединять платы перед изменением компонентов и держать любопытных кошек и гиперактивных собак (и маленьких детей) за дверями комнаты во время работы. Я также не рекомендую ставить кофейную чашку рядом с платой.

Если у вас есть маленькие дети, уберите всю электронику после завершения работы — не много найдется деталей, которые не сможет проглотить ребенок (светодиоды особенно часто кажутся съедобными). Проекты — отличный источник знаний для детей постарше, но пусть ваш здравый смысл подскажет вам, в каком возрасте ребенок сможет работать над проектами независимо.

Я ношу очки. Если вы их не носите, купите рабочие очки для защиты глаз.

Также стоит принять меры для защиты электронных компонентов. На плате Arduino Uno имеются встроенные предохранители, которые не позволят этому забавному устройству сжечь ваш компьютер. Тем не менее сама плата не защищена от повреждений. Стоят эти платы недорого, но не настолько, чтобы регулярная замена или повреждение контактов прошли бесследно для вашего бюджета.

За дополнительной информацией обращайтесь к разделу электроники на сайте StackExchange (*http://electronics.stackexchange.com/questions/139461/safety-of-using-microcontrollers-such-as-arduino*).

Сначала отсоедините плату. Возьмите макетную плату, светодиод, резистор на 220 Ом и два соединительных провода. Все эти компоненты должны присутствовать в большинстве наборов.

Поверните плату так, чтобы надпись «Arduino» была правильно ориентирована. Наверху расположен ряд контактов. Подключите соединительный провод к контакту заземления с пометкой GND. Подключите второй провод к контакту с пометкой ~11. На макетной плате рядом с Arduino разместите остальные компоненты в соответствии с эскизом Fritzing на рис. 12.9. Когда все будет готово, верните Arduino на место.

Приложение из листинга 12.1 предоставляет REPL доступ к нескольким функциям вызовом `REPL.inject()`. Это означает, что вы можете управлять платой из интерактивной среды REPL. Для этого достаточно ввести имя функции, например `on()` для включения индикатора или `fadeOut()` для его выключения с затуханием.

Некоторым функциям, таким как `pulse()`, `fadeIn()` и `fadeOut()`, необходим контакт PWM. Приложение также использует анимацию для функции `pulse()`. Чтобы остановить анимацию, введите `stop()` на консоли REPL. Появляется сообщение «Animation stopped». Даже после остановки анимации светодиод нужно выключить. Кроме того, анимацию следует остановить перед выключением светодиода, в противном случае анимация продолжится.

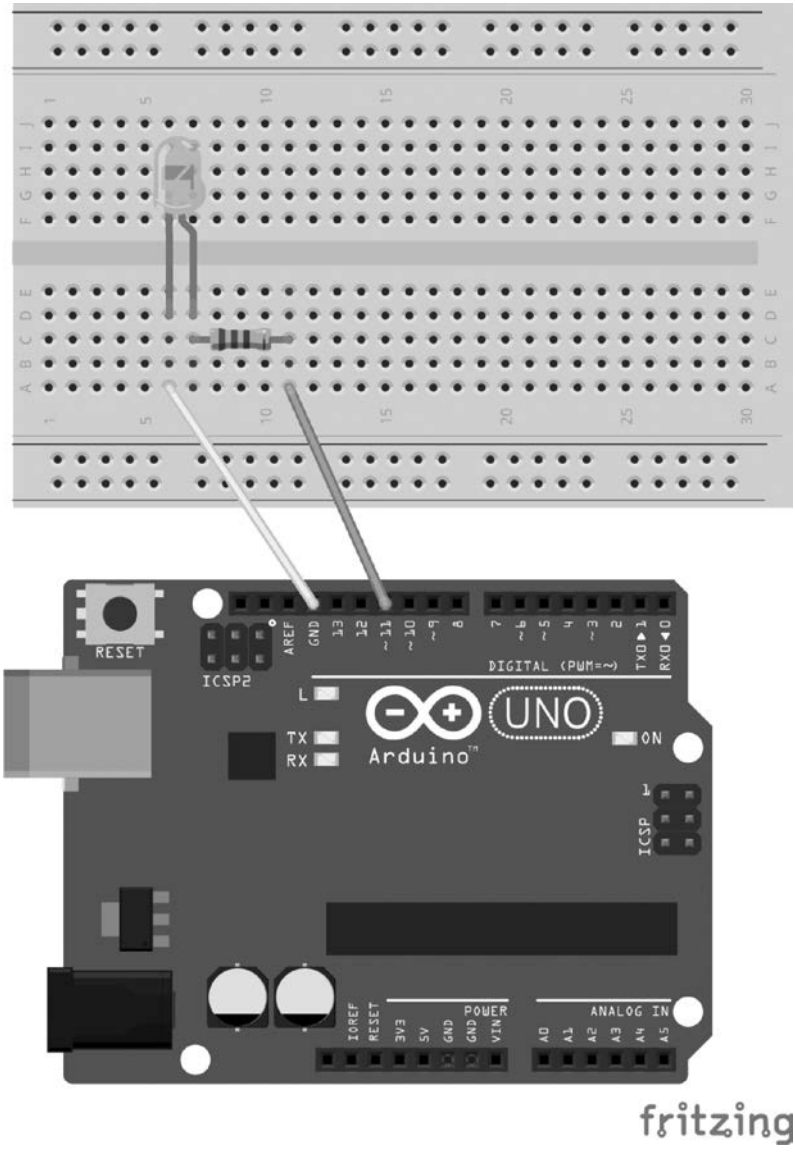


Рис. 12.9. Эскиз Fritzing для проекта на базе Arduino Uno

Листинг 12.1. Интерактивное приложение использует REPL для управления светодиодом

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  console.log("Ready event. Repl instance auto-initialized!");

  var led = new five.Led(11);

  this.repl.inject({
    // Разрешение ограниченного доступа
    // к экземпляру Led из среды REPL.
    on: function() {
      led.on();
    },
    off: function() {
      led.off();
    },
strobe: function() {
    led.strobe(1000);
},
pulse: function() {
    led.pulse({
           easing: "linear",
           duration: 3000,
           cuePoints: [0, 0.2, 0.4, 0.6, 0.8, 1],
           keyFrames: [0, 10, 0, 50, 0, 255],
           onstop: function() {
               console.log("Animation stopped");
           }
       });
},
stop: function() {
    led.stop();
},
fade: function() {
    led.fadeIn();
},
fadeOut: function() {
    led.fadeOut();
}
  });
});
```

Запустите приложение с использованием Node:

```
node fancyblinking
```

После получения сообщения «Ready event ...» можно вводить команды. Например, следующая команда включает вспышки света:

```
>> strobe()
```

Команда `stop()` останавливает эффект вспышки, а команда `off()` полностью выключает светодиод. Затем опробуйте более сложные эффекты `pulse()`, `fade()` и `fadeOut()`. На рис. 12.10 изображен проект в режиме мигающего света (на статической иллюстрации это продемонстрировать нелегко — поверьте на слово, все работает).

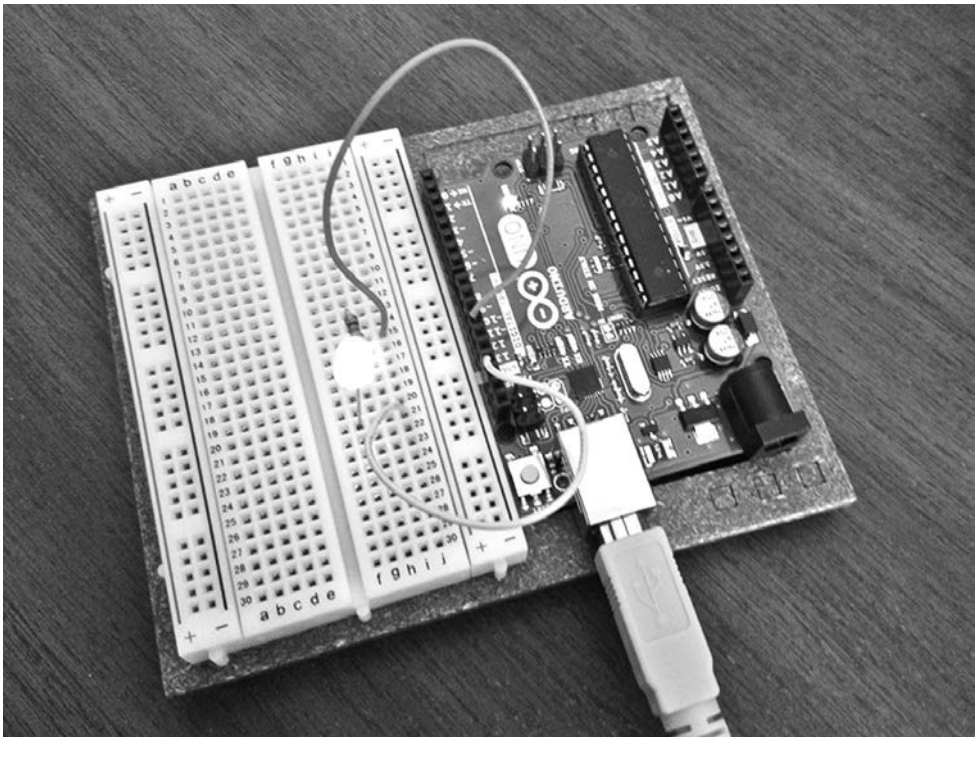


Рис. 12.10. Проект управления светодиодом на базе Arduino Uno

После того как вы освоите управление светодиодом, опробуйте несколько других приложений Node/Arduino:

- Регистрация температуры в реальном времени с использованием Arduino, Node и Plotly (*http://www.instructables.com/id/Real-Time-Temperature-Logging-With-Arduino-NodeJS-/*).

- В статье «The Arduino Experimenter's Guide to NodeJs» (*http://node-ardx.org/*) приведена подборка проектов с готовым кодом.
- Управление лодкой с использованием Arduino и Node (*https://www.sitepoint.com/controlling-a-motorbot-using-arduino-and-node-js/*).
- В качестве альтернативы для Johnny-Five опробуйте модуль Cylon Arduino (*https://cylonjs.com/*).
- Радиоуправляемая машина на базе Arduino Node.js (*http://www.webondevices.com/arduino-nodejs-rc-car-driven-with-html5-gamepad-api/*).
- Управление освещением Philips Hue из Arduino и датчик движения (*http://www.makeuseof.com/tag/control-philips-hue-lights-arduino-and-motion-sensor/*).

С Arduino можно развлекаться месяцами. Впрочем, Raspberry Pi добавляет в IoT-разработку совершенно новое измерение.

Node и Raspberry Pi 2

Raspberry Pi 2 — намного более сложное устройство, чем Arduino. К нему можно подключить монитор, клавиатуру и мышь и использовать как обычный компьютер. Microsoft предоставляет версию Windows 10 для этого устройства, но большинство людей используют Raspbian — реализацию Linux на базе Debian Jesse.

ПОДКЛЮЧЕНИЕ К RASPBERRY PI С ИСПОЛЬЗОВАНИЕМ SSH

Добавив WiFi-адаптер, вы сможете подключаться к Raspberry Pi через SSH. Такие адаптеры стоят недорого, а те, что я использовала, работали без какой-либо настройки. После получения доступа к Интернету на Pi поддержка SSH должна быть включена по умолчанию. Определите IP-адрес Pi и введите его в своей программе SSH. В новой модели Raspberry Pi 3 имеется встроенная поддержка WiFi.

Raspberry Pi 2 работает с карты MicroSD. Карта должна иметь размер не менее 8 Гбайт и относиться к классу 10. Инструкции по установке приведены на сайте Raspberry Pi, но я рекомендую отформатировать карту; скопировать программу NOOBs (New Out Of Box), позволяющую выбрать операционную систему для установки, а затем установить Raspbian. Вы также можете установить загруженный с сайта образ Raspbian напрямую.

На момент написания книги новейшая версия Raspbian была выпущена в феврале 2016 года. В этой версии присутствует встроенная поддержка Node, так как в ней также содержится Node-RED — приложение на базе Node, которое позволяет буквально построить схему методом перетаскивания и привести в действие Raspberry Pi прямо из программы. При этом используется не самая новая версия Node — v0.10.x. Если вы используете Johnny-Five для управления устройством из Node, эта версия должна работать, но все же Node лучше обновить. Я рекомендую использовать LTS-версию Node (4.4.x на момент написания книги) и выполнить инструкции Node-RED по обновлению Node и Node-RED (*http://nodered.org/docs/getting-started/installation.html*) вручную.

Поддержка Node для Node-RED может быть установлена под другим именем. В таком случае вы сможете установить новую версию Node без удаления существующих программ.

После обновления Node установите Johnny-Five. Также необходимо установить другой модуль `raspi-io` — плагин, позволяющий Johnny-Five работать с Raspberry Pi.

```
npm install johnny-five raspi-io
```

Не стесняйтесь исследовать новый компьютер, включая настольные приложения. После этого наступает время настройки физической цепи. Для начала выключите Raspberry Pi.

Для простейшего приложения с мигающим индикатором потребуется макетная плата. Также понадобится резистор — желательно на 220 Ом; такие резисторы включаются в большинство наборов Raspberry Pi. Цветовая маркировка резистора должна состоять из четырех колец: красное, красное, коричневое и золотое.

ЧТЕНИЕ ЦВЕТОВОЙ МАРКИРОВКИ

Digi-Key Electronics предоставляет удобный калькулятор и графическую диаграмму для определения сопротивления резистора в омах (*http://www.digikey.com/en/resources/conversion-calculators/conversion-calculator-resistor-color-code-4-band*). Если вы плохо различаете цвета, обратитесь за помощью к другу или члену семьи или воспользуйтесь мультиметром.

Скетч Fritzing был показан на рис. 12.6. Добавьте резистор и светодиод на макетную плату. Возьмите два провода из набора Raspberry Pi 2 и присоедините один к контакту GRND (третий слева в верхнем ряду), а другой к контакту 13 (седьмой слева в нижнем ряду) на плате Raspberry Pi. Подключите другие концы к макетной плате: провод GRND подключается параллельно первому контакту резистора, а провод GPIO — параллельно аноду (длинной ножке) светодиода.

НЕПРОЧНЫЕ КОНТАКТЫ RASPBERRY PI

Известно, что контакты Raspberry Pi не отличаются прочностью, поэтому многие разработчики используют широкий кабель brekout, подключаемый между контактами Raspberry Pi и макетной платой. Компоненты подключаются к кабелю, а не к Raspberry Pi напрямую.

Снова включите Raspberry Pi. Введите код приложения Node. Он почти не отличается от кода приложения для Arduino, но на этот раз используется плагин `raspi-io`. Кроме того, изменяется нумерация контактов: при работе с Arduino использовалось числовое значение, а для Raspberry Pi — строка. Различия выделены в коде жирным шрифтом:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-13");
  led.blink();
});
```

Запустите программу; светодиод мигает, как и в приложении для Arduino (рис. 12.11).

Интерактивное приложение также можно запустить с Raspberry Pi 2. На этой плате контакт PWM обозначается GPIO 18, тогда как в приложении для Johnny-Five это контакт 12. Это соответствует контакту 6 в верхнем ряду. Обязательно выключите питание Raspberry Pi, прежде чем переключать контакт 13 на контакт 12. Я не буду повторять весь код, а изменения представлены в следующем блоке:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-12");

  // Добавление анимаций и команд
  this.repl.inject({
     ...

  });
});
```

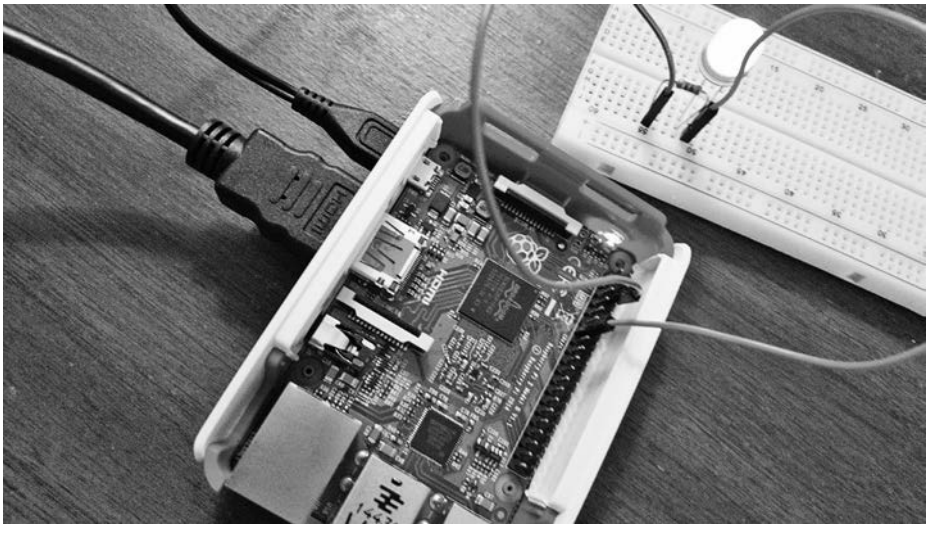


Рис. 12.11. Управление светодиодом с Raspberry Pi и Node

Этот светодиод больше, поэтому анимация будет лучше видна при вызове функции `pulse()`.

Несколько других интересных проектов для Raspberry Pi и Node:

- Простой контроллер светодиода на базе Node.js и WebSockets для Raspberry Pi (*http://www.instructables.com/id/Easy-NodeJS-WebSockets-LED-Controller-for-Raspberr/*).

- Системы домашнего мониторинга на базе Raspberry Pi и Node (*https://www.hackster.io/andreioros/home-monitoring-with-raspberry-pi-and-node-js-8ec795)*.
- Heimcontrol.js: домашняя автоматизация на базе Raspberry Pi и Node (*https://ni-c.github.io/heimcontrol.js/get-started.html*).
- Построение смарт-телевизора с использованием RaspberryPi, NodeJS и Socket.io (*https://www.codementor.io/nodejs/tutorial/build-google-tv-raspberrypi-nodejs-socket-io*).
- Построение системы открывания гаражных дверей на базе Node и MQTT с использованием Intel Edison (*https://blog.risingstack.com/getting-started-with-nodejs-and-mqtt/*).
- Руководство Amazon по созданию устройства Alexa на базе Raspberry Pi (*http://www.techworm.net/2016/03/amazons-guide-make-raspberry-pi-powered-alexa-device.html*).

Вы не представляете, насколько приятно видеть непосредственный материальный отклик на действия вашего приложения Node.

Ш. Пауэрс

Изучаем Node. Переходим на сторону сервера

2-е издание, дополненное и переработанное

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*И. Карпова*
Художник	*С. Заматевская*
Корректоры	*Н. Викторова, И. Тимофеева*
Верстка	*Л. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 22.12.16. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*