Марк Бейтс

CoffeeScript. Второе дыхание JavaScript

Mark Bates

Programming in CoffeeScript

Addison-Wesley

Марк Бейтс

CoffeeScript. Второе дыхание JavaScript

ДМК Пресс
издательство
Москва, 2012

УДК 004.738.5:004.45CoffeeScript
ББК 32.973.202-018.2
Б41

Марк Бейтс

Б41 CoffeeScript. Второе дыхание JavaScript: пер. с анг. А. Киселёв. – М.: ДМК Пресс, 2012. – 312 с.: ил.

ISBN 978-5-94074-842-7

Если вы уже умеете писать на JavaScript, на CoffeeScript вы сможете делать это лучше. А поскольку программный код на CoffeeScript «компилируется» в программный код на JavaScript, он легко впишется практически в любое веб-окружение. В книге автор рассказывает веб-разработчикам, почему язык CoffeeScript так удобен и как он позволяет избежать проблем, часто способствующих появлению ошибок в программном коде на JavaScript и усложняющих его сопровождение. Он познакомит вас со всеми особенностями и приемами, которые необходимо знать, чтобы писать качественный программный код на CoffeeScript, и покажет, как использовать преимущества еще более надежного набора инструментов, входящих в состав языка CoffeeScript.

Издание предназначено в первую очередь веб-разработчикам, использующим JavaScript, а также всем тем, кто хочет писать качественный и понятный код.

УДК 004.738.5:004.45CoffeeScript
ББК 32.973.202-018.2

ISBN 978-0-321-82010-5 (анг.)
ISBN 978-5-94074-842-7 (рус.)

Содержание

Благодарности[1]

Я говорил это в своей первой книге, и повторю здесь: труд писателя невероятно тяжел! Поверьте, никто не скажет, что это не так. Если же кто-то попробует опровергнуть мои слова, он либо лгун, либо Стивен Кинг (Stephen King). К счастью для меня, я нахожусь где-то посередине.

Работа над книгой, с одной стороны, это труд одного, независимого человека, а с другой – труд целой команды. Уложив детей спать, я устремляюсь в свой офис, вскрываю несколько банок Гиннесса (Guiness), завожу музыку и за работу, в уединении, поздно ночью. Закончив главу, я отсылаю ее своему редактору, а он рассылает ее множеству людей, которые берут мою писанину и улучшают ее. Улучшениями могут быть простые исправления грамматических и орфографических ошибок или что-то более сложное, например, помощь в улучшении организации книги или рекомендации по улучшению примеров программного кода, с целью сделать их более понятными. То есть, с одной стороны книга может писаться в одиночку, в темной комнате, но конечный продукт является результатом упорного труда множества людей.

В этом разделе книги я имею возможность сказать спасибо всем, кто помогал создавать, оформлять или как-то иначе обеспечивать высочайшее качество книги, которую вы сейчас держите в руках (или загрузили из Интернета). Без дальнейших обиняков я хочу выразить свои благодарности в стиле лауреата кинопремии Оскара: знайте, я уверен, что кого-то наверняка пропущу, о чем невероятно сожалею.

---

[1] Многие в моем издательстве считали, что мой раздел с выражением благодарности, а также другие части этой книги содержали слишком рискованные шутки, поэтому текст оригинала был исправлен до того, что вы видите перед собой. Я приношу свои извинения, если что-то из написанного мной оскорбило вас, это не было моей целью. Мне уже говорили, что мое чувство юмора понятно далеко не всем. Если вы ничего не имеете против неприличных шуток, следуйте за мной в Твиттере (учетная запись: @markbates).

Прежде всего я хочу поблагодарить мою красавицу жену Рейчел (Rachel). Рейчел – одна из самых участливых и сильных людей, когда-либо встречавшихся мне. Каждую ночь я ложусь с ней в постель, и каждое утро я радуюсь, просыпаясь рядом с ней. Мне доставляет удовольствие заглядывать в ее глаза и видеть там безграничную любовь. Она поддерживает меня, когда я пишу книги, поддерживала, когда я открывал свое дело, и делает все, чтобы моя жизнь была счастливее. Она дала мне двух красавцев сыновей, а я взамен даю ей скабрезные шутки и мои использованные сотовые телефоны. Я определенно отхватил самый лучший кусок в этой сделке, называемой браком, за что бесконечно благодарен ей.

Далее, я хочу поблагодарить моих сыновей, Дилана (Dylan) и Лео (Leo). Даже при том, что ни один из них не принимал прямого участия в создании книги, они придают ценность моей жизни, наполняют ее энергией и эмоциями, которые могут дать только дети. Я люблю вас, мальчики, очень-очень.

Прежде чем перейти от моей семьи к другим людям, я хотел бы сказать спасибо моим родителям (особенно тебе, Мама!) и всем остальным членам моей семьи за то, что всегда были рядом и одновременно поддерживали меня и возвращали к действительности. Я люблю вас всех.

Далее я хочу поблагодарить Дебру Вильямс Коли (Debra Williams Cauley). Дебра была моим редактором, наставником и психиатром, когда я работал над первой книгой «Distributed Programming with Ruby». Я могу только пожелать удачи другим авторам работать с таким же хорошим редактором, как Дебра. Она обладает по-настоящему ангельским терпением.

Надеюсь, если когда-нибудь мне доведется писать еще одну книгу, Дебра окажется рядом. Я не могу представить себе процесс создания книги без ее участия. Спасибо тебе, Дебра.

В создании технической литературы очень важную роль играют технические рецензенты. Работа технического рецензента заключается в том, чтобы читать главы и критиковать их с технической точки зрения, а также постоянно отвечать на вопрос: «Имеет ли смысл рассказывать об этом здесь?». Эти рецензенты выступают в роли ваших читателей. Они технически подкованы и знают свой предмет. Поэтому их отзывы имеют огромное значение. С этой книгой работало несколько технических рецензентов. Но особенно я хотел бы отметить двоих из них – Стюарта Гарнера (Stuart Garner) и Дена Пикетта (Dan Pickett). При работе над этой книгой, Стюарт и

Ден не ограничились лишь служебным долгом и не боялись прямо высказывать свое мнение, когда я делал или говорил какие-нибудь глупости. Они откликались на мои истеричные телефонные звонки и электронные письма в любое время дня и ночи и давали бесценные советы. Если бы я не хотел сам получить все эти суммы гонораров, проставленные в чеках, я не смог бы устоять перед соблазном включить в чеки и их фамилии. (Не волнуйтесь, их труд не остался неоплаченным. Каждый из них получил купон на один бесплатный час времени «Марка».) Спасибо вам, Ден и Стюарт, и всем остальным техническим рецензентам, за ваш кропотливый труд.

В эту книгу вложен труд многих людей. Кто-то занимался оформлением обложки, кто-то алфавитным указателем, кто-то создавал язык (CoffeeScript), и еще множество людей, так или иначе вовлеченных в процесс. Ниже приводится список некоторых из них (о ком я знаю) без какого-то определенного порядка: Джереми Ашкенас (Jeremey Ashkenas), Тревор Барнхам (Trevor Burnham), Ден Фишман (Dan Fishman), Крис Зан (Chris Zahn), Грегг Поллак (Gregg Pollack), Гари Адайр (Gary Adair), Сандра Шредер (Sandra Schroeder), Оби Фернандес (Obie Fernandez), Кристи Харт (Kristy Hart), Энди Бестер (Andy Beaster), Барбара Хача (Barbara Hacha), Тим Райт (Tim Wright), Дебби Вильямс (Debbie Williams), Брайен Франс (Brian France), Ванесса Эванс (Vanessa Evans), Ден Шерф (Dan Scherf), Гари Адайр (Gary Adair), Нони Ратклифф (Nonie Ratcliff) и Ким Бодигреймер (Kim Boedigheimer).

Я также хотел бы сказать спасибо всем, с кем встречался, пока работал над книгой, и кто выслушивал мою бесконечную болтовню об этом. Я знаю, многим это совсем не интересно, но, черт возьми, я обожаю звук моего голоса. Спасибо всем вам, что не врезали мне за мою болтливость, хотя я это, возможно, и заслуживаю.

Наконец, я хотел бы поблагодарить вас, мой читатель. Спасибо, что приобрели эту книгу и помогли поддержать всех, кто, как и я сам, по-настоящему хотят помочь своим собратьям разработчикам, поделившись своими знаниями с остальным миром. Именно для вас я вложил массу времени и сил в эту книгу. Я надеюсь, что когда вы перевернете последнюю страницу, вы будете лучше понимать язык CoffeeScript, и как он может повлиять на вашу работу. Удачи!

Об авторе

Марк Бейтс (Mark Bates) является основателем и главным архитектором консалтинговой компании Meta42 Labs со штаб-квартирой в Бостоне. Марк проводит дни, занимаясь разработкой новых приложений для своих клиентов и консультируя их. После работы он пишет книги, воспитывает детей, а иногда созывает свой ансамбль и «пытается сделать это».

Марк занимается разработкой веб-приложений, в том или ином виде, начиная с 1996 года. Свою карьеру он начинал, как разработчик пользовательского интерфейса, используя HTML и JavaScript, а затем переключился на разработку более сложного программного обеспечения на языках Java и Ruby. В настоящее время Марк тратит свое время, изменяя Ruby со своей новой возлюбленной – CoffeeScript.

Всегда желая поделиться своими знаниями, а точнее, просто услышать звук своего голоса, Марк неоднократно выступал на широко известных конференциях, таких как RubyConf, RailsConf, и jQueryConf. Кроме того, он преподавал на курсах изучения Ruby и Ruby on Rails. В 2009 году издательством Addison-Wesley была опубликована первая книга Марка (и самое замечательное, что не последняя!) «Distributed Programming with Ruby».

Марк живет в пригороде Бостона со своей супругой Рейчел и двумя сыновьями, Диланом и Лео. Марка можно найти в Веб, по адресам: http://www.markbates.com, http://twitter.com/markbates и http://github.com/markbates.

Предисловие

Моя профессиональная карьера разработчика началась в 1999 году, когда я получил первую зарплату как разработчик. (Я не считаю несколько лет, когда я просто получал удовольствие, играя с Веб.) В 1999 году Всемирная паутина была жутким местом. HTML-файлы были перегружены тегами `font` и `table`. Каскадные таблицы стилей CSS только-только начали выходить на сцену. Язык JavaScript [1] существовал всего несколько лет, а война браузеров была в самом разгаре. Безусловно, тогда можно было написать JavaScript-сценарий, выполняющий некоторые операции в одном браузере, но смог бы он работать в другом? Скорее всего, нет. Из-за этого в 2000 годах язык JavaScript получил дурную славу.

В середине 2000-х произошло два важных события, которые помогли JavaScript подняться в глазах разработчиков. Первым из них было появление технологии AJAX. [2] Технология AJAX позволяет разработчикам создавать более быстрые и более интерактивные вебстраницы, благодаря возможности отправлять запросы на сервер в фоновом режиме и устранению необходимости для конечного пользователя обновлять содержимое окна браузера.

Вторым событием стало появление популярных библиотек на JavaScript, таких как Prototype, [3] которые существенно упростили создание JavaScript-сценариев, совместимых со всеми типами браузеров. Появилась возможность использовать технологию AJAX, чтобы сделать приложения более интерактивными и удобными в использовании, и задействовать библиотеку, такую как Prototype, чтобы обеспечить совместимость с основными типами браузеров.

В 2010 году, а точнее в 2011, развитие Всемирной паутины пошло по пути создания «одностраничных» приложений. Такие приложения выполняются под управлением JavaScript-фреймворков, таких как Backbone.js. [4] Эти фреймворки позволяют применять шаблон проектирования MVC [5] с использованием JavaScript. Стало возможным писать на JavaScript целые приложения, а затем загружать их и выполнять в браузере конечного пользователя. Все вместе это

позволяет писать поразительно интерактивные и полнофункциональные клиентские приложения.

Однако, с точки зрения разработчика, ситуация выглядела не так радужно. Несмотря на то, что фреймворки и инструменты значительно упростили разработку подобных приложений, сам язык JavaScript оставался болезненным местом. Язык JavaScript является одновременно невероятно мощным, и в то же время чрезвычайно запутанным. Он полон парадоксов и ловушек, которые быстро могут сделать ваш программный код неконтролируемым и наполненным ошибками.

Так чего же хотят разработчики? Они хотят создавать эти замечательные новые приложения, но единственным языком, который понимают все браузеры, является JavaScript. Конечно, они могут писать эти приложения на Flash, [6] но для этого в браузеры необходимо устанавливать расширения, к тому же эти расширения отсутствуют для некоторых платформ, таких как устройства на iOS [7].

Впервые с языком CoffeeScript [8] я столкнулся в октябре 2010 года. CoffeeScript давал мне надежду приручить JavaScript и подчеркивал наиболее выгодные стороны замысловатого языка, каковым является JavaScript. Он имеет ясный синтаксис, отдавая предпочтение пробелам вместо знаков пунктуации, и защищает от ловушек, поджидающих JavaScript-разработчиков на каждом шагу, таких как неочевидные правила видимости и неправильное употребление операторов сравнения. Но самое замечательное, что в конечном итоге программный код на CoffeeScript компилируется в стандартный программный код на JavaScript, который может выполняться в любом браузере или в другой среде выполнения JavaScript.

Когда я впервые попробовал использовать CoffeeScript, язык был еще далек от совершенства, даже в версии 0.9.4. Я использовал его в проекте моего клиента, чтобы попробовать и увидеть, является ли правдой все, что я слышал о нем. К сожалению, две причины заставили меня отложить его в сторону. Во-первых, язык еще не был готов к широкому использованию. В нем было слишком много ошибок и в нем отсутствовали многие возможности.

Вторая причина, заставившая меня отказаться от CoffeeScript, заключалась в том, что приложение, на котором я проводил эксперименты, не было настоящим JavaScript-приложением. Мне требовалось реализовать лишь кое-какие проверки и организовать отправку запросов с использованием технологии AJAX, что, благодаря

помощи Ruby on Rails [9] достигалось совсем небольшим объемом программного кода на JavaScript.

Так что же заставило меня вернуться к CoffeeScript? Спустя примерно шесть месяцев после первого знакомства с CoffeeScript, было объявлено, [10] что Rails 3.1 будет распространяться вместе с CoffeeScript, в качестве механизма JavaScript по умолчанию. Как и большинство разработчиков, я был ошеломлен этой новостью. Я пытался использовать язык CoffeeScript и не считал его чем-то выдающимся. О чем они думали?

В отличие от большинства моих собратьев разработчиков я решил уделить время, чтобы по-новому взглянуть на CoffeeScript. Шесть месяцев – достаточно долгий срок в разработке любого проекта. Язык CoffeeScript проделал длинный, очень длинный путь. И я решил еще раз попробовать использовать его, на этот раз в приложении, содержащем достаточно большой объем программного кода на JavaScript. Спустя несколько дней повторного использования CoffeeScript, я не только изменил свои взгляды, но и полюбил этот язык.

Не могу сказать точно, что повлияло на мои убеждения, и не буду пытаться объяснить, почему я полюбил этот язык. Я хочу, чтобы вы сами сформировали свое мнение о нем. Надеюсь, что в ходе чтения этой книги вы не только станете приверженцами, но и активными сторонниками этого замечательного маленького языка по вашим собственным причинам. А я коротко расскажу вам о том, что ждет вас впереди. Ниже приводится небольшой фрагмент программного кода на CoffeeScript из действующего приложения, а вслед за ним – эквивалентный фрагмент на JavaScript. Наслаждайтесь!

Пример: (исходный файл: `sneak_peak.coffee`)

```
@updateAvatars = ->
    names = $('.avatar[data-name]').map -> $(this).data('name')
      Utils.findAvatar(name) for name in $.unique(names)
```

Пример: (исходный файл: `sneak_peak.js`)

```
(function() {
    this.updateAvatars = function() {
        var name, names, _i, _len, _ref, _results;
        names = $('.avatar[data-name]').map(function() {
            return $(this).data('name');
        });
        _ref = $.unique(names);
```

```
        _results = [];
        for (_i = 0, _len = _ref.length; _i < _len; _i++) {
            name = _ref[_i];
            _results.push(Utils.findAvatar(name));
        }
        return _results;
    };
}).call(this);
```

Что такое CoffeeScript?

CoffeeScript – это язык программирования, программный код на котором компилируется в программный код на языке JavaScript. Не слишком понятно, знаю, но это так и есть. Язык CoffeeScript близко напоминает такие языки программирования, как Ruby [11] и Python. [12] Он создавался с целью помочь разработчикам повысить производительность труда при разработке программного кода на JavaScript. За счет избавления от необходимости использовать знаки пунктуации, такие как скобки, точки с запятой, и прочие, и использования значимых пробелов взамен этих символов, вы сможете быстро сосредотачиваться на программном коде, а не на бесконечных проверках – расставлены ли все закрывающие фигурные скобки.

Представьте, что вы пишите такой код на JavaScript:

Пример: (исходный файл: `punctuation.js`)

```
(function() {
    if (something === something_else) {
        console.log('do something');
    } else {
        console.log('do something else');
    }
}).call(this);
```

Почему бы не записать его так:

Пример: (исходный файл: `punctuation.coffee`)

```
if something is something_else
    console.log 'do something'
else
    console.log 'do something else'
```

Кроме того, язык CoffeeScript предоставляет ряд сокращенных форм записи, позволяющих записывать сложные фрагменты программного кода на JavaScript гораздо короче. Например, следующий фрагмент позволяет обойти в цикле значения, находящиеся в массиве, не заботясь об их индексах:

Пример: (исходный файл: `array.coffee`)

```
for name in array
      console.log  name
```

Пример: (исходный файл: `array.js`)

```
(function() {
    var name, _i, _len;
    for (_i = 0, _len = array.length; _i < _len; _i++) {
        name = array[_i];
        console.log(name);
    }
}).call(this);
```

В довесок к синтаксическому сахару, язык CoffeeScript помогает также писать более надежный программный код на JavaScript, следя за видимостью переменных и классов, гарантируя использование соответствующих операторов сравнения, и беря на себя решение многих других рутинных задач, в чем вы убедитесь, читая эту книгу.

Языки CoffeeScript, Ruby и Python часто упоминаются вместе и на то есть свои причины. За основу модели CoffeeScript были взяты краткость и простота синтаксиса этих языков. Благодаря этому CoffeeScript создает «ощущение» более современного языка, чем JavaScript, за основу которого были взяты такие языки, как Java [13] и C++. [14] Как и JavaScript, язык CoffeeScript можно использовать в любом программном окружении. Не имеет никакого значения, на каком языке вы пишете свое приложение, Ruby, Python, PHP, [15] Java или .Net [16]. Программный код, скомпилированный в JavaScript, будет работать со всеми ими.

Поскольку программный код на CoffeeScript компилируется в программный код на JavaScript, сохраняется возможность использовать любые библиотеки на JavaScript. Вы можете использовать jQuery, [17] Zepto, [18] Backbone, [19] Jasmine [20] или любую другую библиотеку, и все они будут работать. Такое не слишком часто говорят о новых языках.

Звучит неплохо, но я слышу, как вы спрашиваете: а есть ли недостатки у CoffeeScript, в сравнении со старым добрым JavaScript? Отличный вопрос. Ответ на него: не так много. Во-первых, несмотря на то, что CoffeeScript предоставляет по-настоящему замечательный способ разработки программного кода на JavaScript, он не дает ничего сверх того, что возможно в JavaScript. Например, я по-прежнему не могу создать JavaScript-версию знаменитого метода *method_missing* в языке Ruby. [21] Самый большой недостаток заключается в том, что вам и членам вашей команды придется учить новый язык. К счастью это легко поправимо. Как вы сами увидите, CoffeeScript чрезвычайно прост в изучении.

Наконец, если по каким-либо причинам выбор CoffeeScript окажется ошибочным для вас или вашего проекта, вы сможете взять сгенерированный программный код на JavaScript и работать с ним. Поэтому в действительности вы ничего не потеряете, если попробуете использовать CoffeeScript в следующем или даже в текущем своем проекте (CoffeeScript и JavaScript очень хорошо уживаются друг с другом).

Кому адресована эта книга?

Эта книга адресована разработчикам на JavaScript со средним и высоким уровнем подготовки. Есть несколько причин, по которым я не могу рекомендовать эту книгу тем, кто не знаком с JavaScript, или тем, кто имеет лишь общие представления о нем.

Во-первых, эта книга не учит программированию на языке JavaScript. Эта книга рассказывает о CoffeeScript. Попутно вы, конечно, узнаете кое-что о JavaScript (и CoffeeScript обладает особым талантом заставлять узнавать больше о JavaScript), но мы не будем погружаться в основы JavaScript и постепенно изучать его.

Пример: что делает следующий фрагмент? (исходный файл: `example.js`)

```
(function() {
    var array, index, _i, _len;
    array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    for (_i = 0, _len = array.length; _i < _len; _i++) {
        index = array[_i];
        console.log(index);
    }
}).call(this);
```

Если вам не удалось понять, что делает этот фрагмент, я рекомендую прервать чтение книги на этом. Не переживайте, я действительно хочу, чтобы вы вернулись и продолжили чтение. Просто я полагаю, что вы получите от книги больше, если будете иметь более близкое знакомство с языком JavaScript. Я буду рассказывать о некоторых основных особенностях языка JavaScript, чтобы проиллюстрировать или помочь лучше понять, что делает тот или иной фрагмент на CoffeeScript. Несмотря на то, что для ясности изложения будут охватываться некоторые основы JavaScript, действительно важно, чтобы вы получили базовые знания о языке JavaScript прежде, чем продолжить чтение. Поэтому, пожалуйста, найдите хорошую книгу о JavaScript (их существует великое множество), прочитайте ее и снова присоединяйтесь ко мне, чтобы стать гуру CoffeeScript.

Тем, кто уже является рок-звездой в JavaScript, я предлагаю поднять уровень игры. Эта книга расскажет вам, как писать ясный, более краткий и надежный программный код на JavaScript, используя вкусности CoffeeScript.

Как читать эту книгу

Я попытался так организовать материал в этой книге, чтобы помочь вам построить фундамент в освоении CoffeeScript. Главы в первой части следует читать по порядку, потому что каждая следующая глава основана на понятиях, изучаемых в предыдущих главах, поэтому, пожалуйста, не перепрыгивайте через главы.

В процессе чтения каждой главы вы заметите несколько обстоятельств.

Во-первых, когда я буду представлять какую-нибудь внешнюю библиотеку, новую идею или понятие, я буду включать ссылку на веб-сайт, где вы сможете получить дополнительную информацию о предмете обсуждения. Я очень хотел бы рассказать вам о многом, например, о языке Ruby, однако в книге недостаточно места для этого. Поэтому, если я буду говорить о чем-то, и вы пожелаете познакомиться с этим поближе, прежде чем продолжить чтение, отправляйтесь по указанной мной ссылке, утолите свою жажду познания и возвращайтесь к книге.

Во-вторых, в каждой главе, я время от времени буду сначала показывать неправильное решение проблемы. После знакомства с неправильным путем, мы исследуем его, чтобы понять, что именно в нем неправильно, а затем найдем правильное решение проблемы.

Отличный пример такого подхода встретится вам в главе 1, «Введение», где рассказывается о различных способах компиляции программного кода на CoffeeScript в программный код на JavaScript.

Иногда в книге вам будут встречаться такие примечания:

> **Совет. Какой-нибудь полезный совет.** Так будут оформляться небольшие советы и рекомендации, которые, на мой взгляд, могут вам пригодиться.

Наконец, на протяжении всей книги я буду представлять примеры программного кода сразу по два-три блока. Сначала будет приводиться пример на языке CoffeeScript, а затем скомпилированная версия (на JavaScript) того же примера. Если в процессе выполнения пример выводит какую-нибудь информацию (на мой взгляд, что-нибудь важное), я буду включать также вывод примера. Ниже показано, как это выглядит:

Пример: (исходный файл: `example.coffee`)

```
array = [1..10]
for index in array
      console.log  index
```

Пример: (исходный файл: `example.js`)

```
(function() {
    var array, index, _i, _len;
    array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    for (_i = 0, _len = array.length; _i < _len; _i++) {
        index = array[_i];
        console.log(index);
    }
}).call(this);
```

Вывод: (исходный файл: `example.coffee`)

```
1
2
3
4
5
6
7
```

```
8
9
10
```

Иногда будут встречаться ошибки, которые мне хотелось бы показать. Например:

Пример: (исходный файл: `oops.coffee`)

```
array = [1..10]
oops! index in array
      console.log  index
```

Вывод: (исходный файл: `oops.coffee`)

```
Error: In content/preface/oops.coffee, Parse error on line 3: Unexpected
↪'UNARY'
    at Object.parseError (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/parser.js:470:11)
    at Object.parse (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/parser.js:546:22)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪coffee-script.js:40:22
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/coffee-script.js:68:34)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:110:18
    at [object Object].<anonymous> (fs.js:114:5)
    at [object Object].emit (events.js:64:17)
    at afterRead (fs.js:1081:12)
      at  Object.wrapper  [as  oncomplete]  (fs.js:252:17)
```

Структура книги

С целью помочь вам извлечь максимум пользы из этой книги, я разбил ее на две части.

Часть I: Основы CoffeeScript

Цель первой части – охватить весь язык программирования CoffeeScript сверху донизу. К концу этой части вы должны быть готовы

столкнуться с любыми проектами на CoffeeScript, которые встретятся на вашем пути, включая примеры во второй части книги.

Глава 1, «Введение», рассказывает о различных способах компиляции и запуска программного кода на языке CoffeeScript. Здесь также будет представлена мощная утилита командной строки и одновременно интерактивная срда `coffee`, распространяемая в составе CoffeeScript.

Глава 2, «Основы», начинает исследование отличий между CoffeeScript и JavaScript. Рассказ о синтаксисе, переменных, областях видимости и других элементах языка закладывает мощный фундамент для всех остальных глав в книге.

Глава 3, «Управляющие конструкции», сосредотачивает внимание на наиболее важной части любого языка программирования – управляющих конструкциях, таких как условные операторы `if` и `else`. Здесь также рассказывается о различиях между одними и теми же операторами в CoffeeScript и JavaScript.

Глава 4, «Функции и аргументы», подробно охватывает функции в языке CoffeeScript. Здесь рассказывается об особенностях определения функций, их вызове и дополнительных возможностях, таких как аргументы со значениями по умолчанию и групповые аргументы.

От массивов к объектам, глава 5, «Коллекции и итерации», покажет, как в CoffeeScript можно использовать и изменять коллекции объектов, а также выполнять итерации по ним.

Глава 6, «Классы», завершает первую часть книги описанием поддержки классов в CoffeeScript. Она рассказывает, как определять новые классы и расширять существующие, как переопределять методы суперклассов и многое другое.

Часть II: Практическое применение CoffeeScript

Во второй части демонстрируется использование CoffeeScript на практических примерах. После знакомства с некоторыми сторонами экосистемы, окружающей CoffeeScript, а также после создания полноценного приложения, к концу второй части вы должны будете обладать отточенными навыками владения языком CoffeeScript.

Глава 7, «Инструмент сборки Cake и файлы сборки Cakefile», описывает инструмент Cake, распространяемый в составе CoffeeScript. Этот маленький инструмент можно использовать для создания

сценариев сборки, тестирования и выполнения других операций. Здесь будет рассматриваться все, что может предложить этот инструмент.

Тестирование является важной составляющей разработки программного обеспечения, и в главе 8, «Тестирование с помощью Jasmine», дается короткий обзор одной из наиболее популярных библиотек тестирования для CoffeeScript/JavaScript – Jasmine. В этой главе будет продемонстрировано применение популярного шаблона разработки через тестирование, где при разработке класса калькулятора сначала будут написаны тесты.

Глава 9, «Введение в Node.js», представляет собой краткое введение в управляемый событиями серверный фреймворк Node.js. В этой главе будет показано использование CoffeeScript для создания простого HTTP-сервера, который автоматически компилирует файлы CoffeeScript в файлы JavaScript при запросе их со стороны веб-браузера.

Глава 10, «Пример: список задач, часть 1 (серверная)», описывает создание серверной части приложения списка задач. Основываясь на главе 9, прикладной интерфейс приложения будет конструироваться с применением веб-фреймворка Express.js и компонента Mongoose ORM для MongoDB.

Глава 11, «Пример: список задач, часть 2 (клиент на основе jQuery)», описывает создание клиента прикладного интерфейса списка задач, созданного в главе 10, с применением популярной библиотеки jQuery.

Глава 12, «Пример: список задач, часть 3 (клиент на основе Backbone.js)», описывает создание клиента прикладного интерфейса списка задач, но на этот раз с применением клиентского фреймворка Backbone.js.

Установка CoffeeScript

Я не большой поклонник инструкций по установке в книгах, потому что когда книга попадает на книжную полку, инструкции по установке оказываются устаревшими. Однако некоторые (здесь я подразумеваю издателей книг) полагают, что книги должны содержать раздел с описанием процедуры установки. Поэтому ниже приводятся мои инструкции.

Установить CoffeeScript очень просто. Проще всего выполнить установку – это перейти на сайт http://www.coffeescript.org и отыскать инструкции по установке там.

Я доверяю ответственным за сопровождение в таких проектах, как CoffeeScript и Node [22], – они прекрасные специалисты и не забывают своевременно обновлять инструкции по установке, и вебсайты этих проектов – отличное место, где можно найти эти инструкции.

На момент написания этих строк актуальной версией CoffeeScript была версия 1.2.0. Все примеры в этой книге должны работать в этой версии.

Как запускать примеры

Отыскать и загрузить файлы с исходными текстами примеров для этой книге можно по адресу: https://github.com/markbates/Programming-In-CoffeeScript. Как уже было показано выше, для каждого примера в книге указывается имя файла. Файлы примеров сгруппированы в каталогах по номерам глав.

Если явно не оговаривается иное, примеры должны запускаться в терминале, например:

```
> coffee example.coffee
```

Итак, теперь, когда вы знаете, как запускать примеры после установки CoffeeScript, почему бы нам не встретиться в главе 1 и не приступить к изучению? Встретимся там.

Примечания

1. http://ru.wikipedia.org/wiki/JavaScript.
2. http://ru.wikipedia.org/wiki/Ajax.
3. http://www.prototypejs.org/.
4. http://documentcloud.github.com/backbone/.
5. http://ru.wikipedia.org/wiki/Model-view-controller.
6. http://www.adobe.com/.
7. http://www.apple.com/ios/.
8. http://www.coffeescript.org.
9. http://www.rubyonrails.org.

10. http://www.rubyinside.com/rails-3-1-adopts-coffeescript-jquery-sass-andcontroversy-4669.html.
11. http://ru.wikipedia.org/wiki/Ruby.
12. http://ru.wikipedia.org/wiki/Python.
13. http://ru.wikipedia.org/wiki/Java.
14. http://ru.wikipedia.org/wiki/C%2B%2B.
15. http://ru.wikipedia.org/wiki/PHP.
16. http://ru.wikipedia.org/wiki/.NET_Framework.
17. http://www.jquery.com.
18. https://github.com/madrobby/zepto.
19. http://documentcloud.github.com/backbone.
20. http://pivotal.github.com/jasmine/.
21. http://ruby-doc.org/docs/ProgrammingRuby/html/ref_c_object.html#Object.method_missing.
22. http://nodejs.org.

Часть I. Основы CoffeeScript

В этой первой половине книги рассматривается все, что вы хотели бы узнать и все, что вы должны знать о CoffeeScript. К концу этой части книги вы должны быть готовы начать программировать на CoffeeScript, не испытывать неудобств при использовании сопутствующих инструментов и знать все достоинства и недостатки самого языка.

Изучение мы начнем с самого начала, с таких основ, как компиляция и запуск файлов CoffeeScript. Затем мы перейдем к изучению синтаксиса языка CoffeeScript. После знакомства с синтаксисом будут рассматриваться управляющие конструкции, функции, коллекции и, наконец, классы.

Каждая следующая глава будет основываться на предыдущих главах. Попутно вы познакомитесь со всеми хитростями языка CoffeeScript, позволяющими писать потрясающие JavaScript-приложения. Итак, нам многое предстоит узнать – приступим!

1. Введение

Теперь, прочитав предисловие и установив CoffeeScript, можно ли сказать, что вы действительно приступили к использованию этого языка? В этой главе мы рассмотрим несколько способов компиляции и запуска файлов с программным кодом на языке CoffeeScript.

Я расскажу об удачных и не очень способах компиляции и выполнения программ. В этой главе не будут рассматриваться внутренние механизмы CoffeeScript, однако она ценна уже тем, что в ней вы начнете работать с CoffeeScript. Знание всех достоинств и недостатков инструментов командной строки, распространяемых вместе с CoffeeScript, упростит вашу жизнь, не только как читателя этой книги, но и как разработчика, приступающего к созданию своих первых приложений на CoffeeScript.

Даже если вам уже приходилось сталкиваться с инструментами командной строки CoffeeScript, все равно есть вероятность, что вы узнаете что-то новое в этой главе, поэтому потратьте несколько минут, чтобы прочитать ее, прежде чем перейти к главе 2, «Основы».

Интерактивная среда CoffeeScript

В состав CoffeeScript входит по-настоящему мощная интерактивная среда REPL [1], известная также как интерактивная консоль, позволяющая немедленно приступить к экспериментам с CoffeeScript.

Начать работу с консолью REPL очень просто. Достаточно лишь ввести следующую команду в окне терминала:

```
> coffee
```

После этого должно появиться приглашение к вводу, которое выглядит примерно так:

```
coffee>
```

После появления на экране приглашения `coffee` можно начинать экспериментировать с CoffeeScript.

Консоль REPL можно также запустить командой:

```
> coffee -i
```

или, если вам нравится работать на клавиатуре:

```
> coffee --interactive
```

Начнем с простого примера. Введите в консоли следующую команду:

```
coffee> 2 + 2
```

Вы должны получить следующий ответ:

```
coffee> 4
```

Поздравляю, вы только что написали свой первый программный код на языке CoffeeScript!

Теперь попробуем что-нибудь поинтереснее, что-нибудь более CoffeeScript-овское. Не будем пока задумываться, что делает следующий код (об этом будет рассказываться чуть ниже), а просто запустим его.

Пример: (исходный файл: `repl1.coffee`)

```
a = [1..10]
b = (x * x for x in a)
console.log b
```

Вывод: (исходный файл: `repl1.coffee`)

```
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

Этот фрагмент кода на CoffeeScript выглядит намного интереснее, не так ли? Если говорить в двух словах, мы создали массив и заполнили его целыми числами от 1 до 10. Затем мы обошли все числа в массиве `a`, умножили каждое из них на само себя и создали второй массив, `b`, содержащий эти новые значения. Здорово, правда? Как уже говорилось выше, позднее я буду рад объяснить, как все

это работает, а пока просто порадуемся всем строкам на JavaScript, которые не пришлось писать. Однако, ради любопытства, ниже приводится эквивалентный код на JavaScript:

Пример: (исходный файл: `repl1.js`)

```
(function() {
    var a, b, x;
    a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    b = (function() {
        var _i, _len, _results;
        _results = [];
        for (_i = 0, _len = a.length; _i < _len; _i++) {
            x = a[_i];
            _results.push(x * x);
        }
        return _results;
    })();

    console.log(b);
}).call(this);
```

Как видите, консоль REPL дает отличную возможность опробовать различные идеи и поэкспериментировать с ними. Однако, не все так прекрасно в стране CoffeeScript REPL. Как уже было показано выше, пробелы в языке CoffeeScript имеют большое значение. В результате возникает проблема с вводом многострочного кода на CoffeeScript в консоль REPL. Эта проблема легко решается с помощью символа \.

Попробуем написать простую функцию `add`, принимающую два числа и возвращающую их сумму. Введите следующий код в консоли REPL, строку за строкой:

Пример: (исходный файл: `repl2.coffee`)

```
add = (x, y)->\
    x + y
console.log add(1, 2)
```

Вывод: (исходный файл: `repl2.coffee`)

```
3
```

Обратите внимание, что в конце первой строки добавлен символ \. Он сообщает интерактивной консоли REPL, что далее следует

еще одна строка, продолжающая это выражение. Важно не забывать добавлять символ \ перед каждой следующей строкой, добавляемой в выражение. Первая же строка, не заканчивающаяся символом \, будет считаться концом выражения и консоль REPL попытается выполнить его.

Также важно отметить, что после символа \ по-прежнему необходимо соблюдать отступы, чтобы компилятор CoffeeScript смог корректно интерпретировать эту строку выражения и поместить ее на соответствующее место.

Наконец, чтобы *завершить работу* с консолью REPL, достаточно просто нажать комбинацию клавиш **Ctrl-C**.

Консоль REPL – мощный инструмент, позволяющий быстро опробовать идеи, но, как мы видели, использовать его становится немного труднее, когда приходится иметь дело с более сложным программным кодом. Далее в этой главе, в разделе «Выполнение файлов CoffeeScript», будет показан способ выполнения файлов на языке CoffeeScript, который лучше подходит для запуска сложного программного кода.

Компиляция в браузере

При разработке веб-приложения рано или поздно наступит момент, когда у вас появится желание вставить некоторый программный код на CoffeeScript непосредственно в HTML-файл [2]. Это вполне возможно, и я покажу, как это делается. Однако хочу предостеречь вас от подобного шага. Во-первых, существуют весьма веские причины, по которым такие методики программирования, как ненавязчивый JavaScript (Unobtrusive JavaScript) [3], приобрели большую популярность в последнее время. Не смотря на всю привлекательность возможности выполнить код на CoffeeScript непосредственно в браузере, в действительности это не самое лучшее решение. Отделяя код на JavaScript от разметки HTML, можно сохранить чистоту кода и обеспечить более изящную деградацию функциональности в окружениях, не поддерживающих JavaScript.

Первое время следование методике ненавязчивого JavaScript может показаться немного утомительным и сложным делом, но потом, привыкнув к ней, вы обнаружите, что она упрощает разработку программного кода, более пригодного для многократного использования, и вообще выглядит более логичной. Используя такие инструменты, как библиотека jQuery, можно дождаться окончания

загрузки страницы и подключить весь необходимый программный код на JavaScript к соответствующим объектам в странице. Однако иногда бывает необходимо предварительно «заполнить насос», если можно так выразиться. Обычно это означает, что требуется вызвать метод `init`, возможно, передав для этого некоторые данные в формате JSON [4]. Я мог бы предложить написать этот небольшой блок кода на чистом JavaScript. Однако для желающих написать его на CoffeeScript существует способ позволить браузеру самому скомпилировать этот код.

Рассмотрим HTML-файл с небольшим фрагментом программного кода на CoffeeScript, встроенным в него.

Пример: (исходный файл: `hello_world.html`)

```
<html>
    <head>
        <title>Hello World</title>
        <script src='http://jashkenas.github.com/coffee-script/
➥extras/coffee-script.js' type='text/javascript'></script>
    </head>
    <body>
        <script type='text/coffeescript'>
            name = prompt "What is your name?"
            alert "Hello, #{name}"
        </script>
    </body>
</html>
```

Браузеры, по крайней мере на момент написания этих строк, не имели встроенной поддержки языка CoffeeScript, поэтому к странице необходимо подключить компилятор. К счастью, команда проекта CoffeeScript разработала такой компилятор. Чтобы подключить его к HTML-странице, необходимо добавить следующую строку в раздел `head` HTML-файла:

```
<script src='http://jashkenas.github.com/coffee-script/extras/coffee-script.js'
type='text/javascript'></script>
```

Разумеется, при желании можно загрузить содержимое файла `coffee-script.js` и хранить его локально в своем проекте.

Единственное, что необходимо сделать во встроенном коде на CoffeeScript, чтобы скомпилировать его, это определить соответствующее значение атрибута `type` в теге `script`, как показано ниже:

```
<script type='text/coffeescript'></script>
```

После загрузки страницы компилятор CoffeeScript, хранящийся в файле `coffee-script.js`, отыщет в HTML-документе все теги `script` с атрибутом `type`, имеющим значение `text/coffeescript`, скомпилирует содержащиеся в них сценарии в эквивалентный код на JavaScript и затем выполнит скомпилированный код.

Предостережение

Теперь, когда вы знаете, как скомпилировать программный код на CoffeeScript, встроенный в HTML-документ, я хочу сделать несколько замечаний. Первое, все, что будет говориться в этой книге об областях видимости, об анонимных функциях-обертках и так далее, в равной степени относится и к компиляции CoffeeScript таким способом. Об этом следует помнить, при разработке подобного программного кода.

Второе и, пожалуй, самое важное – это далеко не самый быстрый способ компиляции сценариев на CoffeeScript. Это означает, что после передачи таких страниц в эксплуатацию, все ваши пользователи будут загружать дополнительный файл, размером 162.26 Кбайт, чтобы скомпилировать код на CoffeeScript. А после загрузки страницы компилятору потребуется некоторое время, чтобы отыскать в странице все теги `text/coffeescript`, скомпилировать их содержимое и затем выполнить полученный код на JavaScript. Не думаю, что это будет радовать пользователей.

После этих предупреждений я надеюсь, что вы выберете правильный путь и будете компилировать программный код на CoffeeScript до передачи своих веб-страниц в эксплуатацию.

Компиляция в командной строке

Несмотря на удобство и простоту компиляции и выполнения в браузере программного кода на языке CoffeeScript, в действительности это не самое лучшее решение. Программный код на CoffeeScript следует компилировать до того, как он попадет в Веб. Также вполне возможно, что вам придется писать приложения для фреймворка Node.js или другие серверные приложения, которые выполняются не в браузере и потому не смогут быть скомпилированы там.

Так как же тогда скомпилировать свой код на CoffeeScript? Отличный вопрос. Можно отыскать множество сторонних библиотек, которые скомпилируют ваши файлы с кодом на CoffeeScript (в код на различных языках и на различных платформах), но важнее понять, как сделать это самостоятельно, чтобы можно было написать свой сценарий компиляции, если это потребуется.

Флаг --compile

Начнем с самого важного флага команды `coffee`, `-c`. Флаг `-c` принимает в качестве аргумента имя файла с программным кодом на языке CoffeeScript и компилирует его в файл с программным кодом на языке JavaScript, сохраняя его в том же каталоге. Именно так я компилировал примеры в этой книге.

Сделайте шаг вперед, создайте файл с именем `hello_world.coffee` и добавьте в него следующий текст:

```
greeting = "Hello, World!"
console.log  greeting
```

Теперь скомпилируйте этот файл:

```
> coffee -c hello_world.coffee
```

Эта команда должна скомпилировать программный код CoffeeScript в исходном файле в программный код на JavaScript и сохранить его в новом файле с именем `hello_world.js` в том же каталоге. Файл `hello_world.js` должен содержать следующий текст:

```
(function() {
    var greeting;
    greeting = "Hello, World!";
    console.log(greeting);
}).call(this);
```

Новый файл `hello_world.js` с программным кодом на JavaScript готов к эксплуатации! Теперь можно выпить чашечку чая.

Интерфейс командной строки CoffeeScript

Мы немного поиграли с консолью REPL и узнали, как компилировать программный код на CoffeeScript с помощью инструмента

командной строки `coffee`, однако команда `coffee` предлагает множество дополнительных и интересных возможностей, на которые стоит взглянуть. Чтобы увидеть полный список возможностей команды `coffee`, введите следующую команду в окне терминала:

```
> coffee --help
```

Вы должны увидеть вывод, как показано ниже:

```
Usage: coffee [options] path/to/script.coffee
  -c, --compile      compile to JavaScript and save as .js files
  -i, --interactive  run an interactive CoffeeScript REPL
  -o, --output       set the directory for compiled JavaScript
  -j, --join         concatenate the scripts before compiling
  -w, --watch        watch scripts for changes, and recompile
  -p, --print        print the compiled JavaScript to stdout
  -l, --lint         pipe the compiled JavaScript through JavaScript Lint
  -s, --stdio        listen for and compile scripts over stdio
  -e, --eval         compile a string from the command line
  -r, --require      require a library before executing your script
  -b, --bare         compile without the top-level function wrapper
  -t, --tokens       print the tokens that the lexer produces
  -n, --nodes        print the parse tree that Jison produces
  --nodejs           pass options through to the "node" binary
  -v, --version      display CoffeeScript version
   -h, --help        display this help message
```

(Перевод:

```
Порядок использования: coffee [ключи] путь/к/файлу/script.coffee
  -c, --compile      скомпилировать в JavaScript и сохранить как файл .js
  -i, --interactive  запустить интерактивную консоль CoffeeScript REPL
  -o, --output       указать каталог для скомпилированных JavaScript-файлов
  -j, --join         объединить сценарии перед компиляцией
  -w, --watch        следить за изменениями и немедленно перекомпилировать
  -p, --print        вывести скомпилированный код JavaScript в устройство
                     стандартного вывода
  -l, --lint         пропустить скомпилированный JavaScript через
                     JavaScript Lint
  -s, --stdio        принять сценарий из устройства стандартного ввода и
                     скомпилировать его
  -e, --eval         скомпилировать строку из командной строки
  -r, --require      загрузить библиотеку перед выполнением сценария
```

```
-b, --bare          скомпилировать без добавления функции-обертки
-t, --tokens        вывести лексемы, выделенные лексическим анализатором
-n, --nodes         вывести дерево синтаксического анализа, произведенное
                    инструментом Jison
--nodejs            передать параметры фреймворку "node"
-v, --version       вывести версию CoffeeScript
 -h, --help         вывести это справочное сообщение
```

)

Рассмотрим некоторые из них поближе.

Флаг --output

Сохранение скомпилированного файла с кодом на JavaScript в том же каталоге вполне подходит для экспериментов с CoffeeScript, однако в полнее возможно, что вам потребуется сохранять эти файлы в отдельных каталогах. Так как же скомпилировать файл `hello_world.coffee`, например, в каталог `public/javascripts`?

Ответ на этот вопрос очень прост: достаточно добавить в команду флаг `-o`:

```
> coffee -o public/javascripts -c hello_world.coffee
```

Если после выполнения этой команды заглянуть в каталог `public/javascripts`, в нем можно будет обнаружить вновь скомпилированный файл `hello_world.js`.

Единственное, что не позволяет флаг `-o`, – изменить имя файла. Скомпилированный файл с программным кодом на JavaScript получает то же имя, что и файл с программным кодом на CoffeeScript, за исключением того, что вместо расширения `.coffee` файл получает расширение `.js`. Из-за невозможности изменить имя файла с помощью команды `coffee`, может потребоваться переименовать оригинальный файл или написать сценарий, выполняющий компиляцию файлов CoffeeScript и переименование получаемых файлов.

Флаг --bare

Как будет показано далее в книге, при компиляции программного кода на CoffeeScript он обертывается анонимной JavaScript-функцией. Подробнее об этом будет рассказываться в главе 2, поэтому я не буду углубляться в детали сейчас. Таким образом, результатом компиляции является анонимная JavaScript-функция:

Пример: (исходный файл: `hello_world.js`)

```
(function() {
    var greeting;
    greeting = "Hello, World!";
    console.log(greeting);
}).call(this);
```

В зависимости от каких-либо причин, иногда может возникнуть необходимость исключить генерацию анонимной функции-обертки. В этом случае компилятору CoffeeScript можно передать флаг `-b`.

```
> coffee -b -c hello_world.coffee
```

Эта команда скомпилирует следующий код на JavaScript:
Пример: (исходный файл: `hello_world_bare.js`)

```
var greeting;
greeting = "Hello, World!";
console.log(greeting);
```

Теперь, когда вы знаете, как убрать анонимную функцию-обертку, я хотел бы предостеречь вас от подобного шага. Существует масса веских причин, почему компилятор действует именно таким образом. Подробнее об этой анонимной функции рассказывается в главе 2.

Флаг --print

Иногда, во время компиляции файлов с программным кодом на CoffeeScript, бывает необходимо вывести результат на экран. Для этого команда `coffee` предусматривает флаг `-p`:

```
> coffee -p hello_world.coffee
```

Команда выше выведет в окно терминала такой текст:

```
(function() {
    var greeting;
    greeting = "Hello, World!";
    console.log(greeting);
}).call(this);
```

Этот флаг может пригодиться для отладки, а также будет полезен на этапе изучения CoffeeScript. Сравнивая программный код на CoffeeScript со скомпилированным кодом на JavaScript (как это делается в данной книге), вы быстро начнете понимать, что скрывает в себе программный код на CoffeeScript. Этот флаг оказал мне огромную помощь, когда я только начинал изучать CoffeeScript. Кроме того, он помог мне поднять уровень владения языком JavaScript, позволяя изучать решения, выбираемые компилятором.

Флаг --watch

В процессе разработки проектов на CoffeeScript довольно утомительно постоянно возвращаться в командную строку и повторно компилировать файлы. Чтобы хоть немного упростить вам жизнь, компилятор CoffeeScript предусматривает флаг `-w`. С помощью этого флага можно сообщить команде `coffee`, чтобы она продолжала следить за файлами с программным кодом на CoffeeScript и перекомпилировала их в случае изменения. Например:

```
> coffee -w -c app/assets/coffeescript
```

Эта команда автоматически скомпилирует любой файл с расширением `.coffee` в каталоге `app/assets/coffeescript` или в любом вложенном подкаталоге, как только он изменится [5].

Начиная с версии CoffeeScript 1.2.0, флаг `-w` также реагирует на появление новых файлов в указанном каталоге. Но по своему опыту могу сказать, что это может приводить к множеству ошибок из-за некоторых проблем в лежащем в основе фреймворке Node.js. Надеюсь, что эти проблемы будут устранены, когда данная книга попадет к вам в руки. Однако существует множество сторонних инструментов, предназначенных для обработки событий в файловой системе, таких как добавление или удаление файлов. Когда я работал над книгой, я предпочитал пользоваться инструментом Guard. [6] Это утилита на языке Ruby, позволяющая следить за подобными событиями и выполнять собственный программный код, такой как компиляция файлов с программным кодом на CoffeeScript, при их появлении.

Совет. Помимо инструмента Guard обратите также внимание на утилиту Jitter [7], созданную Тревором Бурхамом (Trevor Burham). Она тоже позволяет следить за файлами с программным кодом на CoffeeScript и компилировать их. Утилита написана с использованием CoffeeScript, поэтому может оказаться весьма полезной.

Выполнение файлов CoffeeScript

Мы рассмотрели несколько способов компиляции файлов с программным кодом на CoffeeScript и обсудили некоторые флаги команды `coffee`, но как запускать файлы CoffeeScript? Неважно, что вы пишете на CoffeeScript, целый веб-сервер или простенький сценарий, выполняющий несложные арифметические операции. Файлы сценариев можно скомпилировать с помощью уже известных инструментов, связать их в HTML-файле и запустить в браузере. Такой подход вполне приемлем для простых сценариев, но он неудобен для чего-то более сложного, такого как веб-сервер. И к тому же не практичен.

Чтобы помочь в этой ситуации, команда `coffee` позволяет выполнять файлы с программным кодом на CoffeeScript:

```
> coffee hello_world.coffee
```

Большинство примеров, которые встретятся нам в этой книге, можно запустить таким способом (если явно не оговаривается иное).

Прочие флаги

Существует еще несколько флагов, таких как `-n` и `-t`. Эти флаги позволяют получить по-настоящему интересную информацию и помогут добавить понимания, как CoffeeScript компилирует исходный программный код, но они никак не пригодятся нам в этой книге, поэтому я не буду ничего рассказывать о них. Однако я рекомендую потратить некоторое время, чтобы опробовать дополнительные флаги и посмотреть на полученные результаты. Найти дополнительную информацию об этих флагах можно в аннотированном исходном файле [8] на языке CoffeeScript.

В заключение

В этой главе мы рассмотрели различные способы компиляции и выполнения программного кода на языке CoffeeScript. Мы увидели достоинства и недостатки разных способов компиляции кода на CoffeeScript и теперь, вооруженные этими знаниями, готовы приступить к экспериментам с примерами в оставшейся части книги. Наконец, мы исследовали возможности команды `coffee` и познакомились

с наиболее важными флагами и параметрами, которые можно передать ей.

Примечания

1. Описание цикла Read-Eval-Print: http://ru.wikipedia.org/wiki/REPL.
2. http://ru.wikipedia.org/wiki/HTML.
3. http://ru.wikipedia.org/wiki/Ненавязчивый_JavaScript.
4. http://ru.wikipedia.org/wiki/JSON.
5. Понятие «изменится» в разных операционных системах понимается по-разному, но обычно, чтобы «изменить» файл и активировать операции, предусматриваемые флагом `-w`, достаточно просто сохранить его.
6. https://github.com/guard/guard.
7. https://github.com/TrevorBurnham/jitter.
8. http://jashkenas.github.com/coffee-script/documentation/docs/command.html.

2. Основы

Теперь, когда закончилось знакомство с такими скучными сведениями, как компиляция и выполнение файлов с программным кодом на CoffeeScript, можно приступать к знакомству с самим языком. Начнем прямо сейчас.

В этой главе мы исследуем синтаксис языка CoffeeScript. Познакомимся с пунктуацией, правилами видимости, переменными и некоторыми другими особенностями.

Синтаксис

Многие особенности языка CoffeeScript обусловлены его синтаксисом, в частности, отсутствием знаков пунктуации.

Знаки пунктуации, такие как фигурные скобки и точки с запятой, отсутствуют в мире CoffeeScript, а круглые скобки потихоньку вымирают.

В доказательство рассмотрим небольшой фрагмент на JavaScript, который большинству из вас может показаться знакомым. Ниже приводится фрагмент, использующий функции из библиотеки jQuery для отправки AJAX-запроса и выполнения некоторых операций с результатами:

Пример: (исходный файл: `jquery_example.js`)

```
$(function() {
    $.get('example.php', function(data) {
        if (data.errors != null) {
            alert("There was an error!");
        } else {
            $("#content").text(data.message);
        }
    }, 'json')
})
```

Язык CoffeeScript позволяет выбросить из этого примера массу лишних знаков пунктуации. Ниже приводится тот же программный код, написанный на языке CoffeeScript:

Пример: (исходный файл: `jquery_as_coffee.coffee`)

```
$ ->
    $.get 'example.php', (data) ->
        if data.errors?
            alert "There was an error!"
        else
            $("#content").text data.message
      , 'json'
```

Далее в этой книге мы подробнее рассмотрим, что делают отдельные части этого примера, а пока исследуем то, что было взято из примера на JavaScript, когда переписывали его на CoffeeScript.

Значимые пробелы

Первое что мы сделали, удалили все фигурные скобки и точки с запятой.

Пример: (исходный файл: `jquery.js`)

```
$(function()
    $.get('example.php', function(data)
        if (data.errors != null)
            alert("There was an error!")
        else
            $("#content").text(data.message)
    , 'json')
)
```

Как теперь компилятор CoffeeScript узнает, каким образом интерпретировать программный код, чтобы он не потерял смысл? Ответ на этот вопрос прост и, возможно, вы уже используете это решение в своей повседневной работе: пробелы! В языке CoffeeScript, как и в языке Python, используется понятие *значимых пробелов*, определяющих порядок интерпретации выражений.

Мне уже приходилось слышать ворчание людей по поводу значимых пробелов, которые аргументировали свое недовольство тем, что им это не нравится. Я нахожу этот аргумент странным. Будете ли вы оформлять тот же самый фрагмент на языке JavaScript, как показано ниже?

Пример: (исходный файл: `jquery.js`)

```
$(function() {
$.get('example.php', function(data) {
if (data.errors != null) {
alert("There was an error!");
} else {
$("#content").text(data.message);
}
}, 'json')
})
```

Надеюсь, что нет! Уверяю вас, при таком стиле оформления вы вынудите своих коллег разработчиков потратить некоторое время на оформление отступов. Удобочитаемость существенно облегчает сопровождение программного кода. Она также поможет вам перенести существующий код на JavaScript на язык CoffeeScript.

Благодаря значимым пробелам, компилятор CoffeeScript поймет, что строка с большим отступом, расположенная ниже инструкции `if`, находится внутри блока `if`. Затем, обнаружив строку с таким же отступом, что и строка с инструкцией `if`, компилятор поймет, что вы закрыли блок инструкции `if`, и будет интерпретировать эту инструкцию, как одноуровневую с инструкцией `if`.

Ниже приводится пример типичной ошибки, с которой можно столкнуться при неправильном оформлении отступов в программном коде на CoffeeScript:

Пример: (исходный файл: `whitespace.coffee`)

```
for num in [1..3]
    if num is 1
      console.log num
      console.log num * 2
  if num is 2
        console.log num
        console.log num * 2
```

Вывод: (исходный файл: `whitespace.coffee`)

```
Error: In content/the_basics/whitespace.coffee, Parse error on line 4:
↪Unexpected 'INDENT'
    at Object.parseError (/usr/local/lib/node_modules/coffee-script/
↪lib/coffee-script/parser.js:470:11)
    at Object.parse (/usr/local/lib/node_modules/coffee-script/
↪lib/coffee-script/parser.js:546:22)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪coffee-script.js:40:22
```

```
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/coffee-script.js:68:34)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:110:18
     at [object Object].<anonymous> (fs.js:114:5)
     at [object Object].emit (events.js:64:17)
     at afterRead (fs.js:1081:12)
     at  Object.wrapper [as  oncomplete] (fs.js:252:17)
```

Ключевое слово function

Следующее, что мы сделали, – избавились от устаревшего ключевого слова `function`. Нам больше не придется писать ключевое слово `function` в своем коде на CoffeeScript.

Пример: (исходный файл: `jquery.js`)

```
$( ()->
    $.get('example.php', (data)->
        if (data.errors != null)
            alert("There was an error!")
        else
            $("#content").text(data.message)
    , 'json')
)
```

Вместо ключевого слова `function` можно использовать стрелку, `->`, справа от списка аргументов. Я знаю, эту особенность немного сложнее запомнить и привыкнуть к ней, но она приобретает больше смысла, если представить ее как «указатель» на программный код, обрабатывающий аргументы.

Круглые скобки

Затем мы удалили из примера *круглые скобки*, но не все.

Пример: (исходный файл: `jquery.js`)

```
$ ->
    $.get 'example.php', (data)->
        if data.errors != null
            alert "There was an error!"
        else
```

```
            $("#content").text data.message
      , 'json'
```

Почему нельзя удалить все круглые скобки? Практически всегда удалять круглые скобки необязательно. Правила использования скобок могут показаться немного запутанными, особенно когда речь заходит о функциях. Рассмотрим пример внимательнее и выясним, почему одни скобки были удалены, а другие нет.

Когда в программе выполняется вызов функции, такой как:

```
alert "There was an error!"
```

мы можем убрать круглые скобки. То есть, когда функции передаются аргументы, разрешается опустить скобки. Однако, если функция не принимает никаких аргументов, скобки становятся необходимы, чтобы сообщить интерпретатору JavaScript, что производится вызов функции, а не обращение к переменной. Как уже говорилось, правила могут показаться запутанными.

> **Совет.** При наличии сомнений, всегда используйте круглые скобки, если это поможет сделать ваш программный код более понятным и удобочитаемым.

Итак, если при вызове функции с аргументами скобки не нужны, тогда почему мы оставили их в следующей строке?

```
$("#content").text data.message
```

Почему эту строку нельзя записать, как показано ниже?

```
$ "#content" .text data.message
```

Дело в том, что в этом случае скомпилированный код на JavaScript выглядел бы так:

```
$("#content".text(data.message));
```

Как видите, компилятор CoffeeScript не смог определить, какому объекту принадлежит метод `text`, поэтому он предположил, что вызывается метод строки `"#content"`. Оставив круглые скобки, мы

четко определили, что здесь вызывается метод `text` объекта jQuery, возвращаемого вызовом функции `$("#content")`.

Прежде чем закрыть тему круглых скобок (не волнуйтесь, мы еще вернемся к ней, когда будем рассматривать функции), я хотел бы отметить, что круглые скобки по-прежнему могут использоваться для логической группировки.

Пример: (исходный файл: `grouping.coffee`)

```
if x is true and (y is true or z is true)
      console.log 'hello, world'
```

Пример: (исходный файл: `grouping.js`)

```
(function() {
    if (x === true && (y === true || z === true))
        console.log('hello, world');
}).call(this);
```

Переменные и области видимости

В этом разделе мы поговорим о переменных – о правилах видимости переменных и как они объявляются в языке CoffeeScript. В языке JavaScript эта тема является одной из самых сложных, и часто является источником ошибок и недопонимания. В этом разделе мы посмотрим, как CoffeeScript старается оставить в прошлом проблемы, связанные с видимостью переменных.

Видимость переменных в JavaScript

Используя *переменные в языке JavaScript*, многие программисты, как начинающие, так и опытные, не подозревают, что существует два способа объявления переменных. А многие из тех, кто знают об этом, не знают, чем они отличаются. Поэтому задержимся ненадолго и попробуем разобраться в этих двух способах, хотя бы на самом простом уровне.

Взгляните на следующий фрагмент:

Пример: (исходный файл: `js_variable_scope.js`)

```
a = 'A';
myFunc = function() {
    a = 'AAA';
    var b = 'B';
}
```

Если бы мы выполнили этот фрагмент в браузере или под управлением другого механизма JavaScript, мы получили бы следующий результат:

Вывод:

```
> console.log(a)
A
> myFunc();
> console.log(a)
AAA
> console.log(b)
ReferenceError: b is not defined
```

Возможно, вы уже знаете, почему возникла ошибка при попытке обратиться к переменной `b`. Однако, я уверен, вы не ожидали, что переменная `a` сохранит значение, полученное внутри функции `myFunc`, я прав? Итак, почему это произошло?

Ответ на этот вопрос прост и корнями уходит в различия между тем, как объявлялись эти две переменные.

Объявив переменную `a` без ключевого слова `var`, мы сообщили интерпретатору JavaScript, что эта переменная должна быть определена в глобальном пространстве имен. Поскольку переменная с именем `a` уже существовала в глобальном пространстве имен, мы подменили ее новой, одноименной переменной, объявив ее внутри функции `myFunc`. Упс.

Переменная `b` была объявлена внутри функции `myFunc` с использованием ключевого слова `var`, поэтому интерпретатор JavaScript создал ее в области видимости функции `myFunc`. А так как переменная `b` видима только внутри функции, при попытке обратиться к ней за пределами этой функции мы получили сообщение об ошибке, из-за того, что интерпретатор JavaScript не смог отыскать переменную с именем `b` в глобальном пространстве имен.

> **Совет.** Пример, представленный в разделе «Видимость переменных в JavaScript», наглядно демонстрирует, почему всегда следует использовать ключевое слово `var` для объявления переменных. Это считается наиболее удачной практикой и CoffeeScript, желая помочь, гарантирует, что все будет происходить именно так.

Видимость переменных в CoffeeScript

Теперь, немного разобравшись с видимостью переменных в языке JavaScript, снова вернемся к нашему примеру, но на этот раз записанному на языке CoffeeScript:

Пример: (исходный файл: `coffeescript_variable_scope.coffee`)

```
a = 'A'
myFunc = ->
    a = 'AAA'
    b = 'B'
```

Пример: (исходный файл: `coffeescript_variable_scope.js`)

```
(function() {
    var a, myFunc;
    a = 'A';
    myFunc = function() {
        var b;
        a = 'AAA';
        return b = 'B';
    };
}).call(this);
```

Не будем пока обращать внимания на анонимную функцию, обертывающую наш программный код (об этом мы поговорим чуть ниже), и посмотрим, как компилятор CoffeeScript объявил наши переменные. Обратите внимание, что все переменные, включая переменную, хранящую ссылку на функцию `myFunc`, объявлены с использованием ключевого слова `var`. Таким способом CoffeeScript гарантирует, что все переменные будут объявлены соответствующим образом.

В этом примере есть еще одна примечательная особенность – несмотря на то, что CoffeeScript добавляет все необходимые объявления переменных, он не в состоянии предотвратить затирание оригинальной переменной `a`. Это обусловлено тем, что когда программный код на CoffeeScript компилируется в программный код на JavaScript, компилятор обнаруживает ранее объявленную переменную `a` и полагает, что она преднамеренно используется внутри функции.

Анонимная функция-обертка

Как вы уже видели и должны были заметить, все наши примеры, скомпилированные в программный код на JavaScript, были обернуты *анонимной функцией*, вызывающей саму себя. Вы наверняка сейчас задаете себе вопрос – что делает эта функция? Я расскажу об этом.

Как было показано в примерах, демонстрирующих видимость переменных в языке JavaScript, мы легко можем обращаться к переменным, объявленным без ключевого слова `var`. Когда переменные объявляются таким способом, они помещаются в глобальное пространство имен, доступное отовсюду.

Даже когда переменная `a` объявлялась с использованием ключевого слова `var`, она все равно оказывалась в глобальном пространстве имен. Почему так? Причина проста – эта переменная определялась за пределами каких-либо функций. А когда переменная определяется за пределами области видимости функции, она оказывается в глобальном пространстве имен. Это означает, что если какая-нибудь из используемых библиотек тоже определяет переменную `a`, одна из них будет затерта другой, которая определяется последней. Причем, это справедливо для любого пространства, не только для глобального.

Как вы определяете переменные и функции вне глобального пространства имен, чтобы они были доступны только вашей программе? Вы делаете это, обертывая свой программный код анонимной функцией. Именно это и делает компилятор CoffeeScript.

Теперь у вас должны зародиться две мысли. Первая: «Умно, теперь я могу делать все, что захочу, не волнуясь о загрязнении глобального пространства имен.». Вторая: «Минутку, а как же тогда я обеспечу доступ к моим переменным и функциям, которые я хотел бы поместить в глобальное пространство имен, чтобы ими могли пользоваться другие?». Это очень важные мысли. Рассмотрим вторую из них, потому что она единственная, которую действительно требуется рассмотреть.

Если поместить всю программу или библиотеку в тело анонимной функции, как это делает компилятор CoffeeScript, никакие другие библиотеки и программный код в вашем приложении не смогут использовать ее. Возможно, это именно то, чего вы добиваетесь. Но если это не так, эту проблему можно решить несколькими способами.

Ниже демонстрируется один из способов обеспечить доступ к своей функции из внешнего мира:

Пример: Экспортирование через объект `window` (исходный файл: `expose_with_window.coffee`)

```
window.sayHi = ->
      console.log "Hello, World!"
```

Пример: Экспортирование через объект `window` (исходный файл: `expose_with_window.js`)

```
(function() {
    window.sayHi = function() {
        return console.log("Hello, World!");
    };
}).call(this);
```

В этом примере для экспортирования функции был использован объект `window`. В мире, где программный код выполняется в браузере, это отличный способ экспортирования функций. Однако, благодаря успеху фреймворка Node.js и других серверных технологий на основе JavaScript, все более популярным становится использование окружений, отличных от браузеров, выполняющих программный код на JavaScript. Если попробовать выполнить этот пример с помощью команды `coffee`, мы получим следующий результат:

Пример: Экспортирование через объект `window` (исходный файл: `expose_with_window.coffee.output`)

```
ReferenceError: window is not defined
    at Object.<anonymous> (.../the_basics/expose_with_window.coffee:3:3)
    at Object.<anonymous> (.../the_basics/expose_with_window.coffee:7:4)
    at Module._compile (module.js:432:26)
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/coffee-script.js:68:25)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
↪command.js:110:18
    at [object Object].<anonymous> (fs.js:114:5)
    at [object Object].emit (events.js:64:17)
    at afterRead (fs.js:1081:12)
      at Object.wrapper [as oncomplete] (fs.js:252:17)
```

Когда выполняется команда `coffee`, в окружении выполнения отсутствует объект `window`, посредством которого производится попытка экспортировать функцию. Можно ли решить эту проблему? Ответ очень прост. Когда компилятор CoffeeScript создает анонимную функцию-обертку, он передает ей аргумент `this`. Мы легко можем присоединить свою функцию к аргументу `this` и экспортировать ее таким способом. Взгляните:

Пример: Экспортирование через ссылку this (исходный файл: expose_with_this.coffee)

```
this.sayHi = ->
      console.log "Hello, World!"
```

Пример: Экспортирование через ссылку this (исходный файл: expose_with_this.js)

```
(function() {
    this.sayHi = function() {
        return console.log("Hello, World!");
    };
}).call(this);
```

Если этот программный код выполнить в браузере, наша функция будет доступна другому программному коду на JavaScript, потому что ссылка this в браузере ссылается на объект window. При выполнении этого же фрагмента с помощью команды coffee или в приложении Node.js, ссылка this будет ссылаться на глобальный объект global. Используя ссылку this для экспортирования функций и переменных, можно сделать свой программный код доступным извне.

Теперь, для полноты картины, рассмотрим еще один, более простой и, на мой взгляд, более понятный способ экспортирования своего программного кода:

Пример: Экспортирование с помощью префикса @ (исходный файл: expose_with_at.coffee)

```
@sayHi = ->
      console.log "Hello, World!"
```

Пример: Экспортирование с помощью префикса @ (исходный файл: expose_with_at.js)

```
(function() {
    this.sayHi = function() {
        return console.log("Hello, World!");
    };
}).call(this);
```

Как видно из скомпилированного кода на JavaScript, компилятор CoffeeScript компилирует символ @ в ссылку this.. В программном

коде на CoffeeScript любую ссылку `this.` можно заменить символом `@` и быть уверенным, что он будет работать как прежде. Несмотря на то, что использование символа `@` не является обязательным, в CoffeeScript он является более предпочтительным, поэтому на протяжении всей книги я буду использовать исключительно этот способ.

Интерполяция

В этом разделе мы поговорим об *интерполяции строк*, встроенных документах и комментариях в языке CoffeeScript. Механизм интерполяции строк позволяет легко динамически конструировать строки, без необходимости использовать раздражающий синтаксис конкатенации строк, в котором легко ошибиться. Встроенные документы дают возможность встраивать отформатированный многострочный текст. А комментарии, ну тут и так все понятно.

Интерполяция строк

Одна из моих любимых мозолей, набитых в JavaScript, – динамическое конструирование строк. Рассмотрим пример создания на языке JavaScript текстового поля ввода в виде разметки HTML, где динамически определяются некоторые атрибуты:

Пример: (исходный файл: `javascript_concatenation.js`)

```
var field, someId, someName, someValue;
someName = 'user[firstName]';
someId = 'firstName';
someValue = 'Bob Example';
field = "<input type='text' name='" + someName + "' id='" + someId + "'
↪value='" + (escape(someValue)) + "'>";
console.log(field);
```

Вывод: (исходный файл: `javascript_concatenation.js`)

```
<input type='text' name='user[firstName]' id='firstName'
↪value='Bob%20Example'>
```

Видите, как некрасиво выглядит этот код, в котором, к тому же, легко допустить ошибку? Не забыл ли я вставить парные закрывающие апострофы ', окружающие значения атрибутов в теге? Все ли кавычки " на месте? Я уверен, что это так, но убедиться в этом при беглом осмотре очень непросто.

Следуя за некоторыми современными языками программирования, такими как Ruby, CoffeeScript дает возможность использовать две разновидности строк, интерполируемые и литералы. Рассмотрим их.

Интерполируемые строки

Чтобы избавиться от раздражающей операции конкатенации, представленной в примере с полем ввода HTML, язык CoffeeScript дает возможность использовать интерполируемые строки и избавиться от этой проблемы.

Что такое интерполяция строк? Интерполяция строк – это способ добавления произвольного кода на CoffeeScript в строки и его выполнения во время работы программы. В данном примере необходимо вставить в HTML-строку значения нескольких переменных, и CoffeeScript дает такую возможность. Ниже показано, как можно записать тот же самый пример, используя механизм интерполяции строк в языке CoffeeScript:

Пример: (исходный файл: `html_string_interpolation.coffee`)

```
someName = 'user[firstName]'
someId = 'firstName'
someValue = 'Bob Example'
field = "<input type='text' name='#{someName}' id='#{someId}'
➥value='#{escape someValue}'>"
console.log  field
```

Пример: (исходный файл: `html_string_interpolation.js`)

```
(function() {
    var field, someId, someName, someValue;
    someName = 'user[firstName]';
    someId = 'firstName';
    someValue = 'Bob Example';
    field = "<input type='text' name='" + someName + "' id='" + someId + "'
➥value='" + (escape(someValue)) + "'>";
    console.log(field);
}).call(this);
```

Вывод: (исходный файл: `html_string_interpolation.coffee`)

```
<input type='text' name='user[firstName]' id='firstName'
➥value='Bob%20Example'>
```

Такой код выглядит лучше, не так ли? Его проще писать, читать и сопровождать.

Как известно, в языке JavaScript отсутствует механизм интерполяции строк. Все строки интерпретируются одинаково. В языке CoffeeScript, напротив, не все строки интерпретируются одинаково. Строки в кавычках, как в только что рассмотренном примере, интерпретируются компилятором CoffeeScript на этапе компиляции и при необходимости преобразуются в последовательность операций конкатенации строк на языке JavaScript. Строки в апострофах в языке CoffeeScript называются строковыми литералами и будут рассматриваться чуть ниже.

Когда возникает необходимость вставить некоторый код на CoffeeScript в строку, заключенную в кавычки, используется синтаксическая конструкция `#{}`. Все выражения, находящиеся внутри фигурных скобок, будут выделены компилятором и затем объединены в операцию конкатенации для получения конструируемой строки. В фигурные скобки можно заключить любой допустимый программный код на языке CoffeeScript:

Пример: (исходный файл: `string_interpolation_extra.coffee`)

```
text = "Add numbers: #{1 + 1}"
console.log text
text = "Call a function: #{escape "Hello, World!"}"
console.log text
day = 'Sunday'
console.log "It's a beautiful #{if day is 'Sunday' then day else "Day"}"
```

Пример: (исходный файл: `string_interpolation_extra.js`)

```
(function() {
    var day, text;
    text = "Add numbers: " + (1 + 1);
    console.log(text);
    text = "Call a function: " + (escape("Hello, World!"));
    console.log(text);
    day = 'Sunday';
    console.log("It's a beautiful " + (day === 'Sunday' ? day : "Day"));
}).call(this);
```

Вывод: (исходный файл: `string_interpolation_extra.coffee`)

```
Add numbers: 2
Call a function: Hello%2C%20World%21
It's a beautiful Sunday
```

Строковые литералы

Строковые литералы, это именно то, что подразумевает их название – литералы строк. То есть все, что входит в состав строкового литерала, возвращается в виде обычной строки – именно так ведут себя строки в языке JavaScript.

Для создания строкового литерала в языке CoffeeScript необходимо использовать апострофы '. Вернемся к предыдущему примеру конструирования текстового поля ввода в виде разметки HTML. На этот раз вместо кавычек подставим апострофы и посмотрим, что из этого получится:

Пример: (исходны файл: `html_string_literal.coffee`)

```
someName = 'user[firstName]'
someId = 'firstName'
someValue = 'Bob Example'
field = '<input type=\'text\' name=\'#{someName}\'
↪id=\'#{someId}\' value=\'#{escape(someValue)}\'>'
console.log field
```

Пример: (исходный файл: `html_string_literal.js`)

```
(function() {
    var field, someId, someName, someValue;
    someName = 'user[firstName]';
    someId = 'firstName';
    someValue = 'Bob Example';
    field = '<input type=\'text\' name=\'#{someName}\'
↪id=\'#{someId}\' value=\'#{escape(someValue)}\'>';
    console.log(field);
}).call(this);
```

Вывод: (исходный файл: `html_string_literal.coffee`)

```
<input type='text' name='#{someName}' id='#{someId}'
↪value='#{escape(someValue)}'>
```

Глядя на вывод можно сказать, что полученный результат далек от желаемого. Это обусловлено тем, что строковые литералы не

поддерживают интерполяцию. Вместо динамически изменяющегося содержимого, мы наблюдаем в полученной строке программный код, который должен был добавить это динамическое содержимое. Конечно, иногда бывает желательно иметь именно такое поведение, однако такое случается довольно редко.

Несмотря на то, что строковые литералы не позволяют вставлять в них динамическое содержимое, тем не менее, они по-прежнему требуют дополнительной обработки, как и строки в языке JavaScript. Строковые литералы в CoffeeScript допускают возможность использования экранированных символов. Например:

Пример: (исходный файл: `literal_string_with_escapes.coffee`)

```
text = "Header\n\tIndented Text"
console.log  text
```

Пример: (исходный файл: `literal_string_with_escapes.js`)

```
(function() {
    var text;
    text = "Header\n\tIndented Text";
    console.log(text);
}).call(this);
```

Вывод: (исходный файл: `literal_string_with_escapes.coffee`)

```
Header
        Indented  Text
```

Как видите, символы перевода строки `\n` и табуляции `\t` были корректно распознаны и обработаны компилятором. Для экранирования символов обратного слеша в языке CoffeeScript, как и в языке JavaScript, допускается использовать двойные символы обратных слешей, например:

Пример: (исходный файл: `literal_string_with_backslash.coffee`)

```
text = "Insert \\some\\ slashes!"
console.log  text
```

Пример: (исходный файл: `literal_string_with_backslash.js`)

```
(function() {
    var text;
```

```
    text = "Insert \\some\\ slashes!";
    console.log(text);
}).call(this);
```

Вывод: (исходный файл: `literal_string_with_backslash.coffee`)

```
Insert \some\ slashes!
```

В таких языках программирования, как Ruby, использование строковых литералов позволяет увеличить производительность программ. Это обусловлено тем, что во время выполнения программы отпадает необходимость анализировать строку и выполнять какие-либо дополнительные операции с ней. Однако, поскольку программный код на языке CoffeeScript компилируется в программный код на языке JavaScript, вся дополнительная работа по интерполяции строк выполняется на этапе компиляции.

> **Совет.** Падение производительности, обусловленное использованием интерполируемых строк вместо строковых литералов, наблюдается только на этапе компиляции. я не вижу недостатков, препятствующих постоянному использованию интерполируемых строк в кавычках. Даже если вся мощь интерполируемых строк не нужна прямо сейчас, ее можно будет задействовать позднее, если все строки в программе сделать интерполируемыми.

Встроенные документы

Встроенные документы [1] позволяют легко встраивать многострочный текст в программный код на CoffeeScript, сохраняя его форматирование. При работе со встроенными документами применяются те же правила, что и при работе с обычными строковыми литерами. Чтобы на языке CoffeeScript создать интерполируемый встроенный документ, следует использовать тройные кавычки с каждого конца документа. Чтобы создать литерал встроенного документа, следует использовать тройные апострофы.

Рассмотрим простой пример. Возьмем за основу предыдущее текстовое поле ввода и добавим вокруг него немного разметки HTML:

Пример: (исходный файл: `heredoc.coffee`)

```
someName = 'user[firstName]'
someId = 'firstName'
```

```
someValue = 'Bob Example'
field = """
        <ul>
            <li>
                <input type='text' name='#{someName}' id='#{someId}'
↪value='#{escape(someValue)}'>
            </li>
        </ul>
        """
console.log  field
```

Пример: (исходный файл: `heredoc.js`)

```
(function() {
    var field, someId, someName, someValue;
    someName = 'user[firstName]';
    someId = 'firstName';
    someValue = 'Bob Example';
    field = "<ul>\n <li>\n <input type='text' name='" + someName + "' id='"
↪+ someId + "' value='" + (escape(someValue)) + "'>\n </li>\n</ul>";
    console.log(field);
}).call(this);
```

Вывод: (исходный файл: `heredoc.coffee`)

```
<ul>
    <li>
        <input type='text' name='user[firstName]' id='firstName'
↪value='Bob%20Example'>
    </li>
</ul>
```

Как видите, вывод сохранил форматирование, как в оригинале. Здесь также видно, что при выводе учитывается величина отступа самого встроенного документа, что позволяет сохранить форматирование программного кода.

Комментарии

Любой хороший язык программирования должен давать возможность вставлять *комментарии* более чем одним способом, и CoffeeScript не является исключением. Комментарии в языке CoffeeScript можно оформлять двумя способами, и оба способа по-разному влияют на скомпилированный код JavaScript.

Встроенные комментарии

Первый тип комментариев – *встроенные комментарии*. Встроенные комментарии очень просты. Чтобы создать встроенный комментарий, достаточно просто вставить в его начало символ #. Все, что следует за символом # до конца строки будет игнорироваться компилятором CoffeeScript.

Пример: (исходный файл: `inline_comment.coffee`)

```
# Вычислить платежную ведомость компании
calcPayroll()
payBils() # Заплатить по счетам компании
```

Пример: (исходный файл: `inline_comment.js`)

```
(function() {
    calcPayroll();
    payBils();
}).call(this);
```

Как видите, наши комментарии не попали в скомпилированный код на JavaScript. В настоящее время все еще ведутся споры, хорошо это или плохо. Было бы неплохо, если бы комментарии попадали в программный код на JavaScript. Однако, из-за того, что программный код на CoffeeScript не всегда напрямую отображается в код на JavaScript, иногда компилятору будет непросто определить, куда вставлять комментарий. К тому же при таком подходе скомпилированный код на JavaScript получается более легковесным, потому что не перегружен комментариями, а за нами сохраняется возможность комментировать свой код, не опасаясь, что это приведет к разбуханию кода на JavaScript.

Блочные комментарии

Другой тип комментариев, поддерживаемых языком CoffeeScript, – это *блочные комментарии*. Блочные комментарии прекрасно подходят для добавления длинных, многострочных пояснений. В виде таких комментариев можно добавлять информацию о лицензировании и версии продукта, описание особенностей использования API и так далее. В отличие от встроенных комментариев, блочные комментарии включаются в скомпилированный код на JavaScript.

Пример: (исходный файл: `block_comment.coffee`)

```
###
Моя замечательная библиотека v1.0
Авторские права принадлежат: Мне!
Распространяется на условиях лицензии MIT
###
```

Пример: (исходный файл: `block_comment.js`)

```
/*
Моя замечательная библиотека v1.0
Авторские права принадлежат: Мне!
Распространяется на условиях лицензии MIT
*/
(function() {
}).call(this);
```

Как видно из этого примера, блочные комментарии близко напоминают встроенные документы. Чтобы создать блочный комментарий, достаточно добавить по три символа # с обоих концов пояснительного текста.

Расширенный синтаксис регулярных выражений

Синтаксис определения и выполнения *регулярных выражений* в языках CoffeeScript и JavaScript абсолютно идентичен. [2] Однако в языке CoffeeScript имеются некоторые дополнительные возможности, упрощающие создание по-настоящему длинных и сложных регулярных выражений.

Всем нам иногда приходится сталкиваться с довольно громоздкими регулярными выражениями. Было бы неплохо иметь возможность разбивать подобные выражения на несколько строк и добавлять комментарии к каждой его части. В языке CoffeeScript такая возможность имеется.

Чтобы определить многострочное регулярное выражение, его следует обернуть тройными символами слеша (по три с каждого конца), подобно тому, как оформляются встроенные документы и блочные комментарии.

Взгляните, как это выглядит в исходном программном коде на языке CoffeeScript:

Пример: (исходный файл: `extended_regex.coffee`)

```
REGEX = /// ^
    (/ (?! [\s=] )   # пропустить ведущие пробелы или знаки равно
    [^ [ / \n \\ ]*  # все остальное
    (?:
        (?: \\[\s\S] # класс любых экранированных
          | \[       # символов
               [^ \] \n \\ ]*
               (?: \\[\s\S] [^ \] \n \\ ]* )*
             ]
        ) [^ [ / \n \\ ]*
    )*
    /) ([imgy]{0,4}) (?!\w)
///
```

Пример: (исходный файл: `extended_regex.js`)

```
(function() {
    var REGEX;
    REGEX = /^(\/(?![\s=])[^[\/\n\\]*(?:(?:\\[\s\S]|\[[^\]\n\\]*(?:\\[\s\S]
↪[^\]\n\\]*)*])[^[\/\n\\]*)*\/)([imgy]{0,4})(?!\w)/;
}).call(this);
```

Как видите, из скомпилированного кода на JavaScript удаляются все дополнительные пробелы и комментарии.

В заключение

Теперь, когда вы познакомились с синтаксисом языка CoffeeScript, можно приступать к изучению более интересных подробностей о языке. Если что-то, из того, о чем рассказывалось выше, осталось для вас непонятным, потратьте несколько минут, вернитесь к началу главы и перечитайте ее еще раз. Очень важно, чтобы вы понимали все, что здесь обсуждалось, потому что эти сведения образуют фундамент для всего, о чем будет рассказываться далее в книге. Когда вы разберетесь со всем, что было непонятно, переходите к следующей главе!

Примечания

1. http://en.wikipedia.org/wiki/Heredoc.
2. http://ru.wikipedia.org/wiki/Регулярные_выражения.

3. Управляющие конструкции

Практически все языки программирования предусматривают понятия операторов [1] и условных инструкций [2]. Языки JavaScript и CoffeeScript не являются исключениями. Операторы и условные инструкции используются совместно, для образования важнейших элементов в любом языке программирования. Операторы позволяют выполнять такие операции, как сложение или вычитание двух чисел, сравнение двух объектов, сдвиг байтов в объектах и так далее. Условные инструкции, в свою очередь, позволяют управлять потоком выполнения приложения, исходя из определенных условий. Например, *если* пользователь не зарегистрировался, *тогда* отправить его на страницу регистрации, *иначе* показать ему секретную страницу. Это и есть условная инструкция.

В этой главе мы познакомимся с операторами и условными инструкциями, и как они определяются в языке CoffeeScript.

Операторы и псевдонимы

Языки программирования JavaScript и CoffeeScript прекрасно согласуются друг с другом, в том смысле, что одни и те же операторы в них по большей части ведут себя одинаково. Однако в языке CoffeeScript имеется ряд особенностей, помогающих не наступить на некоторые мины, скрытые в тайниках JavaScript. С целью убедиться, что мы полностью понимаем особенности всех операторов, совершим короткий тур по операторам JavaScript, а попутно я расскажу, чем они отличаются от аналогичных операторов в языке CoffeeScript. Прежде чем продолжить, хочу предупредить: далее я буду полагать, что вы знакомы со всеми операторами в языке JavaScript. Если это не так, тогда сейчас самое время освежить свои знания. тем, кому необходимо справочное руководство, могу порекомендовать http://en.wikibooks.org/wiki/JavaScript/Operators. В этом кратком, но хорошо написанном справочнике вы найдете обзор операторов, доступных в языке JavaScript.

Арифметические операторы

Ниже приводится список всех операторов в языке JavaScript:

- + сложение;
- - вычитание;
- * умножение;
- / деление (возвращает вещественное значение);
- % деление по модулю (возвращает целочисленный остаток);
- + унарный оператор преобразования строки в число;
- - унарный минус (изменяет знак числа);
- ++ инкремент (имеет префиксную и постфиксную формы записи);
- -- декремент (имеет префиксную и постфиксную формы записи).

Теперь посмотрим, как эти операторы транслируются в их аналоги, в мире CoffeeScript:

Пример: (исходный файл: `arithmetic.coffee`)

```
console.log "+ Addition: #{1 + 1}"
console.log "- Subtraction: #{10 - 1}"
console.log "* Multiplication: #{5 * 5}"
console.log "/ Division: #{100 / 10}"
console.log "% Modulus: #{10 % 3}"
console.log "+ Unary conversion of string to number: #{+'100'}"
console.log "- Unary negation: #{-50}"
i = 1
x = ++i
console.log "++ Increment: #{x}"
i = 1
x = --i
console.log "-- Decrement: #{x}"
```

Пример: (исходный файл: `arithmetic.js`)

```
(function() {
    var i, x;
    console.log("+ Addition: " + (1 + 1));
    console.log("- Subtraction: " + (10 - 1));
    console.log("* Multiplication: " + (5 * 5));
    console.log("/ Division: " + (100 / 10));
    console.log("% Modulus: " + (10 % 3));
    console.log("+ Unary conversion of string to number: " + (+'100'));
    console.log("- Unary negation: " + (-50));
```

```
    i = 1;
    x = ++i;
    console.log("++ Increment: " + x);
    i = 1;
    x = --i;
    console.log("-- Decrement: " + x);
}).call(this);
```

Вывод: (исходный файл: `arithmetic.coffee`)

```
+ Addition: 2
- Subtraction: 9
* Multiplication: 25
/ Division: 10
% Modulus: 1
+ Unary conversion of string to number: 100
- Unary negation: -50
++ Increment: 2
--  Decrement:  0
```

Как видно из примера, все арифметические операторы CoffeeScript напрямую отображаются в аналогичные операторы JavaScript, следовательно, мы не будем терять сон, пытаясь вспомнить их особенности.

Присваивание

Теперь перейдем к *операторам присваивания* в языке JavaScript, которые перечислены ниже:

- `=` присваивание;
- `+=` сложение и присваивание;
- `-=` вычитание и присваивание;
- `*=` умножение и присваивание;
- `/=` деление и присваивание;
- `%=` деление по модулю и присваивание;
- `?=` проверка существования или присваивание;
- `||=` логическое ИЛИ или присваивание;
- `&&=` присваивание, если оба операнда истинны.

Посмотрим, как они выражаются в языке CoffeeScript.

Пример: (исходный файл: `assignment.coffee`)

```
console.log "= Assign:"
x = 10
```

```
console.log x
console.log "+= Add and assign:"
x += 25
console.log x
console.log "-= Subtract and assign:"
x -= 25
console.log x
console.log "*= Multiply and assign:"
x *= 10
console.log x
console.log "/= Divide and assign:"
x /= 10
console.log x
console.log "%= Modulus and assign:"
x %= 3
console.log x
console.log "?= Exists or assign:"
y ?= 3
console.log y
y ?= 100
console.log y
console.log "||= Or or assign:"
z = null
z ||= 10
console.log z
z ||= 100
console.log z
console.log "&&= Assign if both are true:"
a = 1
b = 2
console.log a &&= b
console.log  a
```

Пример: (исходный файл: `assignment.js`)

```
(function() {
    var a, b, x, z;
    console.log("= Assign:");
    x = 10;
    console.log(x);
    console.log("+= Add and assign:");
    x += 25;
    console.log(x);
    console.log("-= Subtract and assign:");
```

```
    x -= 25;
    console.log(x);
    console.log("*= Multiply and assign:");
    x *= 10;
    console.log(x);
    console.log("/= Divide and assign:");
    x /= 10;
    console.log(x);
    console.log("%= Modulus and assign:");
    x %= 3;
    console.log(x);
    console.log("?= Exists or assign:");
    if (typeof y === "undefined" || y === null) y = 3;
    console.log(y);
    if (typeof y === "undefined" || y === null) y = 100;
    console.log(y);
    console.log("||= Or or assign:");
    z = null;
    z || (z = 10);
    console.log(z);
    z || (z = 100);
    console.log(z);
    console.log("&&= Assign if both are true:");
    a = 1;
    b = 2;
    console.log(a && (a = b));
    console.log(a);
}).call(this);
```

Вывод: (исходный файл: `assignment.coffee`)

```
= Assign:
10
+= Add and assign:
35
-= Subtract and assign:
10
*= Multiply and assign:
100
/= Divide and assign:
10
%= Modulus and assign:
1
?= Exists or assign:
```

```
33
||= Or or assign:
10
10
&&= Assign if both are true:
22
```

И снова наблюдается прямое соответствие. Разве это не здорово?

Сравнение

Теперь перейдем к *операторам сравнения* и посмотрим, как они соответствуют в языках CoffeeScript и JavaScript.

- == равно;
- != не равно;
- > больше;
- >= больше или равно;
- < меньше;
- <= меньше или равно;
- === идентично (операнды равны и имеют один и тот же тип);
- !== не идентично.

Посмотрим на поведение этих операторов в языке CoffeeScript:

Пример: (исходный файл: `comparison.coffee`)

```
console.log "== Equal: #{1 == 1}"
console.log "!= Not equal: #{1 != 2}"
console.log "> Greater than: #{2 > 1}"
console.log ">= Greater than or equal to: #{1 >= 1}"
console.log "< Less than: #{1 < 2}"
console.log "<= Less than or equal to: #{1 < 2}"
console.log "=== Identical: #{'a' === 'a'}"
console.log "!== Not identical: #{1 !== 2}"
```

Вывод: (исходный файл: `comparison.coffee`)

```
Error: In content/control_structures/comparison.coffee, Parse error on line
➥ 13: Unexpected '='
    at Object.parseError (/usr/local/lib/node_modules/
➥ coffee-script/lib/coffee-script/parser.js:470:11)
    at Object.parse (/usr/local/lib/node_modules/coffee-script/lib/
➥ coffee-script/parser.js:546:22)
    at /usr/local/lib/node_modules/coffee-script/lib/coffee-script/
➥ coffee-script.js:40:22
```

```
    at Object.run /usr/local/lib/node_modules/coffee-script/lib/
➥ coffee-script/coffee-script.js:68:34)
    at /usr/local/lib/node_modules/coffee-script/lib/
➥ coffee-script/command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/
➥ coffee-script/command.js:110:18
    at [object Object].<anonymous> (fs.js:114:5)
    at [object Object].emit (events.js:64:17)
    at afterRead (fs.js:1081:12)
      at Object.wrapper [as oncomplete] (fs.js:252:17)
```

Что-то пошло не так. Посмотрим, что произошло, и почему наш пример породил массу ошибок. Дело в том, что язык CoffeeScript не позволяет использовать операторы `===` и `!==`. Надеюсь, что когда эта книга выйдет из печати, сообщения об ошибках будут более точные и информативные. Я уже слышу, как некоторые из вас вскрикивают в ужасе, потому что эти операторы сравнения используются повсеместно, но не волнуйтесь, эти операторы есть в CoffeeScript, просто они оформляются немного иначе. Позвольте мне объяснить.

Переделаем наш пример – выбросим из него примеры использования операторов `===` и `!==`:

Пример: (исходный файл: `comparison2.coffee`)

```
console.log "== Equal: #{1 == 1}"
console.log "!= Not equal: #{1 != 2}"
console.log "> Greater than: #{2 > 1}"
console.log ">= Greater than or equal to: #{1 >= 1}"
console.log "< Less than: #{1 < 2}"
console.log "<= Less than or equal to: #{1 < 2}"
```

Пример: (исходный файл: `comparison2.js`)

```
(function() {
    console.log("== Equal: " + (1 === 1));
    console.log("!= Not equal: " + (1 !== 2));
    console.log("> Greater than: " + (2 > 1));
    console.log(">= Greater than or equal to: " + (1 >= 1));
    console.log("< Less than: " + (1 < 2));
    console.log("<= Less than or equal to: " + (1 < 2));
}).call(this);
```

Вывод: (исходный файл: `comparison2.coffee`)

```
== Equal: true
!= Not equal: true
> Greater than: true
>= Greater than or equal to: true
< Less than: true
<= Less than or equal to: true
```

Отлично! Наш пример не породил ни одной ошибки, но вы должны были заметить одну очень интересную особенность. Заметили, как операторы == и != были скомпилированы в коде на JavaScript? Они были скомпилированы в операторы === и !==, соответственно. Почему компилятор CoffeeScript поступил так? Взгляните, что происходит в JavaScript при использовании операторов == и !=:

Пример: (исходный файл: javascript_comparison.js)

```
x = 1;
y = '1';
console.log(x == y); // истина
```

В примере на языке JavaScript число 1 равно строке “1”, несмотря на то, что это объекты разных типов. Причина в том, что при использовании оператора сравнения ==, интерпретатор JavaScript автоматически приводит операнды к одному типу и затем выполняет сравнение. То же относится и к оператору !=. Такое поведение является источником огромного количества ошибок в программном коде на JavaScript. Чтобы реализовать истинное сравнение объектов, необходимо использовать оператор ===.

Если в этот же пример подставить оператор ===, операция сравнения вернет ложное значение вместо истинного:

Пример: (исходный файл: javascript_comparison2.js)

```
x = 1;
y = '1';
console.log(x === y); // ложь
```

Чтобы избавить нас от подобных ошибок, компилятор CoffeeScript автоматически преобразует все операторы == и != в === и !==. Разве не здорово? Язык CoffeeScript помогает сократить вероятность появления ошибок до минимума. Скажите ему спасибо. Однако, существует еще один способ использования операторов === и !==, который мы рассмотрим ниже, когда будем обсуждать псевдонимы.

Строки

Наконец, существует несколько операторов для работы со *строками*.

- + конкатенация;
- += конкатенация и присваивание.

Ниже демонстрируется использование этих операторов в языке CoffeeScript:

Пример: (исходный файл: `string_operators.coffee`)

```
console.log "+ Concatenation: #{'a' + 'b'}"
x = 'Hello'
x += " World"
console.log "+= Concatenate and assign: #{x}"
```

Пример: (исходный файл: `string_operators.js`)

```
(function() {
    var x;
    console.log("+ Concatenation: " + ('a' + 'b'));
    x = 'Hello';
    x += " World";
    console.log("+= Concatenate and assign: " + x);
}).call(this);
```

Вывод: (исходный файл: `string_operators.coffee`)

```
+ Concatenation: ab
+= Concatenate and assign: Hello World
```

К счастью, эти операторы действуют точно так же, как в языке JavaScript.

Оператор проверки существования

Когда я впервые столкнулся с языком CoffeeScript, я тут же влюбился в *оператор проверки существования*. Этот оператор позволяет проверить... существование переменной с помощью единственного символа ?.

Взгляните, как он действует:

Пример: (исходный файл: `existential1.coffee`)

```
console.log x?
```

Пример: (исходный файл: existential1.js)

```
(function() {
    console.log(typeof x !== "undefined" && x !== null);
}).call(this);
```

Вывод: (исходный файл: existential1.coffee)

```
false
```

Как видно в этом примере, компилятор CoffeeScript генерирует программный код на языке JavaScript, который проверяет, была ли определена переменная x; если переменная определена, ее значение сравнивается со значением null. Этот оператор позволяет писать весьма мощные условные инструкции.

Пример: (исходный файл: existential_if.coffee)

```
if html?
      console.log  html
```

Пример: (исходный файл: existential_if.js)

```
(function() {
    if (typeof html !== "undefined" && html !== null) console.log(html);
}).call(this);
```

Но это еще не все. С помощью этого оператора можно убедиться в существовании некоторого объекта и если он существует, вызвать его метод. Моим любимым примером такого объекта является объект console. Для тех, кто не знаком с объектом console, замечу, что он существует в большинстве браузеров и служит для вывода сообщений во встроенную консоль ошибок JavaScript. Обычно этот объект используется разработчиками для вывода сообщений в определенных точках программы с целью отладки или получения дополнительной информации. Я использую его практически во всех примерах в этой книге для демонстрации вывода.

Проблема с объектом console заключается в том, что он доступен не везде (Internet Explorer, я с укоризной смотрю на тебя!). Если попытаться вызвать метод или обратиться к свойству переменной, которая не существует, браузер возбудит исключение и выполнение программы прервется. Оператор проверки существования позволяет обойти проблему вызова методов не существующих объектов, таких как объект console в некоторых браузерах.

Сначала рассмотрим пример, в котором не используется оператор проверки существования:

Пример: (исходный файл: existential2.coffee)

```
console.log "Hello, World"
console.log someObject.someFunction()
console.log "Goodbye, World"
```

Пример: (исходный файл: existential2.js)

```
(function() {
    console.log("Hello, World");
    console.log(someObject.someFunction());
    console.log("Goodbye, World");
}).call(this);
```

Вывод: (исходный файл: existential2.coffee)

```
Hello, World
ReferenceError: someObject is not defined
    at Object.<anonymous> (.../control_structures/existential2.coffee:5:15)
    at Object.<anonymous> (.../control_structures/existential2.coffee:9:4)
    at Module._compile (module.js:432:26)
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/coffee-script.js:68:25)
    at /usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/command.js:110:18
    at [object Object].<anonymous> (fs.js:114:5)
    at [object Object].emit (events.js:64:17)
    at afterRead (fs.js:1081:12)
      at Object.wrapper [as oncomplete] (fs.js:252:17)
```

Неприятная ошибка, правда? Наш пример породил ошибку, потому что интерпретатор JavaScript не смог отыскать объект с именем someObject, когда мы попытались вызвать его метод. Теперь добавим оператор проверки существования после имени someObject и посмотрим, что получится.

Пример: (исходный файл: existential3.coffee)

```
console.log "Hello, World"
console.log someObject?.someFunction()
console.log "Goodbye, World"
```

Пример: (исходный файл: existential3.js)

```
(function() {
    console.log("Hello, World");
    console.log(typeof someObject !== "undefined" && someObject !== null ?
➥someObject.someFunction() : void 0);
    console.log("Goodbye, World");
}).call(this);
```

Вывод: (исходный файл: existential3.coffee)

```
Hello, World
undefined
Goodbye, World
```

Так определенно лучше. Даже при том, что мы получили сообщение `undefined` при попытке обратиться к переменной `someObject`, программа сохранила свою работоспособность. В действующей программе можно было бы вывести сообщение в журнал или вывести на экран какое-либо предупреждение, а пока будем считать, что нам удалось сделать наш программный код немного надежнее.

Псевдонимы

Чтобы сделать программный код немного дружественнее, в языке CoffeeScript имеется несколько *псевдонимов* для наиболее типичных операторов. Некоторые из этих псевдонимов являются отличными расширениями языка, но есть такие, которые могут вызывать путаницу. Список псевдонимов в языке CoffeeScript и их аналоги в языке JavaScript приводится в табл. 3.1.

Таблица 3.1. Псевдонимы в языке CoffeeScript и их аналоги в JavaScript

CoffeeScript	JavaScript
`is`	`===`
`isnt`	`!==`
`not`	`!`
`and`	`&&`
`or`	`||`
`true, yes, on`	`true`
`false, no, off`	`false`
`@, this`	`this`
`of`	`in`
~~`in`~~	не определен

Рассмотрим все эти псевдонимы, кроме двух последних. Два последних псевдонима будут рассматриваться в главе 5, «Коллекции и итерации».

Псевдонимы is и isnt

Ниже приводится пример использования псевдонимов операторов `is` и `isnt`:

Пример: (исходный файл: `is_aliases.coffee`)

```
name = "mark"
console.log name is "mark"
console.log  name  isnt  "bob"
```

Пример: (исходный файл: `is_aliases.js`)

```
(function() {
    var name;
    name = "mark";
    console.log(name === "mark");
    console.log(name !== "bob");
}).call(this);
```

Вывод: (исходный файл: `is_aliases.coffee`)

```
true
true
```

Как видите, псевдоним `is` отображается в оператор `===`, а псевдоним `isnt` отображается в оператор `!==`. Как уже говорилось выше в этой главе, компилятор CoffeeScript не только приветствует, но и вынуждает использовать операторы сравнения `===` и `!==`. Кроме того, компилятор CoffeeScript предпочел бы, чтобы вы использовали псевдонимы `is` и `isnt`. Их считают «фишкой CoffeeScript». Сейчас ведутся споры, касающиеся удаления операторов `==` и `!=` из языка CoffeeScript и принудительного использования операторов `is` и `isnt`. Когда я пишу эти строки, этого еще не произошло, но вы должны знать о такой возможности. Данное предупреждение касается всех псевдонимов и соответствующих им операторов.

Псевдоним not

Псевдоним `not` вызывает у меня смешанные чувства. Мне нравится, как он украшает программный код, но мне не нравится, что он

действует не всегда так, как мне хотелось бы. По сути псевдоним `not` является точным аналогом оператора `!` в языке JavaScript: он «переключает» логическое значение переменной. То есть, истинное значение он превращает в ложное и наоборот.

Посмотрим, как ведет себя псевдоним `not`:

Пример: (исходный файл: `not_alias.coffee`)

```
userExists = false
if not userExists
      console.log "the user doesn't exist!"
```

Пример: (исходный файл: `not_alias.js`)

```
(function() {
    var userExists;
    userExists = false;
    if (!userExists) console.log("the user doesn't exist!");
}).call(this);
```

Вывод: (исходный файл: `not_alias.coffee`)

```
the user doesn't exist!
```

Как видите, между псевдонимом `not` и переменной, логическое значение которой требуется «переключить», необходимо вставить пробел.

И что же здесь неправильного? Дело в том, что на языке CoffeeScript можно написать следующий код:

Пример: (исходный файл: `not_alias_wrong.coffee`)

```
name = "mark"
console.log name isnt "bob"
console.log name is not "bob"
```

Пример: (исходный файл: `not_alias_wrong.js`)

```
(function() {
    var name;
    name = "mark";
    console.log(name !== "bob");
    console.log(name === !"bob");
}).call(this);
```

Вывод: (исходный файл: not_alias_wrong.coffee)

```
true
false
```

С точки зрения грамматики эти две строки выглядят вполне корректно, но в действительности это совершенно разные выражения. Строка, где используется псевдоним isnt, проверяет равенство значений двух объектов. Строка, где используется псевдоним is not, проверяет равенство значения первого объекта с инвертированным логическим значением второго объекта. Такая особенность позволяет легко допустить ошибку, особенно если вы только начинаете знакомиться с языком CoffeeScript.

Псевдонимы and и or

Мне нравятся псевдонимы and и or. Они не только улучшают читаемость программного кода, но и ведут себя именно так, как ожидается! Например:

Пример: (исходный файл: and_or.coffee)

```
if true and true
    console.log "true and true really is true"
if false or true
    console.log "something was true"
```

Пример: (исходный файл: and_or.js)

```
(function() {
    if (true && true) console.log("true and true really is true");
    if (false || true) console.log("something was true");
}).call(this);
```

Вывод: (исходный файл: and_or.coffee)

```
true and true really is true
something was true
```

Как я уже сказал, их поведение в точности соответствует ожидаемому.

Псевдонимы логических значений

Язык CoffeeScript поддерживает не только логические значения true и false, но также взял на вооружение некоторые псевдонимы

из языка YAML [3], позволяющие сделать программный код более удобочитаемым.

Взгляните:

Пример: (исходный файл: `boolean_operators.coffee`)

```
myAnswer = true
console.log myAnswer is yes
console.log myAnswer is true
light = true
console.log light is on
console.log light is true
myAnswer = false
console.log myAnswer is no
console.log myAnswer is false
light = false
console.log light is off
console.log  light  is  false
```

Пример: (исходный файл: `boolean_operators.js`)

```
(function() {
    var light, myAnswer;
    myAnswer = true;
    console.log(myAnswer === true);
    console.log(myAnswer === true);
    light = true;
    console.log(light === true);
    console.log(light === true);
    myAnswer = false;
    console.log(myAnswer === false);
    console.log(myAnswer === false);
    light = false;
    console.log(light === false);
    console.log(light === false);
}).call(this);
```

Вывод: (исходный файл: `boolean_operators.coffee`)

```
true
true
true
true
true
```

```
true
true
true
```

Как видите, псевдонимы yes, no, on и off могут сделать программный код более наглядным и удобочитаемым.

Псевдоним @

Последний псевдоним, который мы рассмотрим, прежде чем двинуться дальше, – это псевдоним @. Мы еще будем возвращаться к этому псевдониму время от времени на протяжении всей книги, когда будем знакомиться с различными областями языка CoffeeScript. А пока рассмотрим наиболее типичный случай его применения: как псевдонима ключевого слова this в языке JavaScript.

Ниже приводится очень простой пример использования псевдонима @:

Пример: (исходный файл: at_alias.coffee)

```
object = {
    name: 'mark'
    sayHi: ->
        console.log "Hello: #{@name}"
}
object.sayHi()
console.log  @name
```

Пример: (исходный файл: at_alias.js)

```
(function() {
    var object;
    object = {
        name: 'mark',
        sayHi: function() {
            return console.log("Hello: " + this.name);
        }
    };
    object.sayHi();
    console.log(this.name);
}).call(this);
```

Вывод: (исходный файл: at_alias.coffee)

```
Hello: mark
undefined
```

Если вы плохо понимаете назначение ссылки `this` в языке JavaScript или вообще не знаете, что это такое, я настоятельно рекомендую отложить сейчас эту книгу, взять книгу о JavaScript и прочитать ее.

Если сравнить программный код на языке CoffeeScript с результатами его компиляции в программный код на языке JavaScript, можно заметить, что все символы `@` были заменены на `this.`. Я считаю, что псевдоним `@` упрощает восприятие программного кода. Благодаря ему я легко могу отличить «локальные» переменные, иногда называемые переменными экземпляра, внутри функций от переменных и функций, объявленных за пределами текущей функции.

Условные инструкции if/unless

В своей карьере разработчика я еще не встречал языка программирования, который не поддерживал бы понятия условных инструкций. Я верю, что такой язык существует, но я сомневаюсь, что кто-то сможет пользоваться им, кроме его создателя.

Условные инструкции позволяют наделить наши программы разумом. Они дают возможность писать приложения, реагирующие на изменение ситуации. Пользователь зарегистрировался в системе? Если нет, ему следует предложить пройти регистрацию, в противном случае показать ему его личную страницу. Текущий пользователь заплатил по счету? Если заплатил, дать ему доступ к его счету, если нет – предложить ему заплатить. Все это примеры условных инструкций. На основе ответов на эти вопросы программа выбирает разные пути дальнейшего выполнения.

В языке CoffeeScript, как и в большинстве других языков программирования, тоже имеются условные инструкции. Обычно эти условные инструкции используются совместно с операторами и псевдонимами, рассмотренными выше, чтобы помочь программе в принятии умных решений.

Инструкция if

В этой книге уже приводились примеры использования инструкции `if`. Она имеет весьма простую организацию.

Пример: (исходный файл: `if.coffee`)

```
if true
      console.log "the statement was true"
```

Пример: (исходный файл: `if.js`)

```
(function() {
    if (true) console.log("the statement was true");
}).call(this);
```

Вывод: (исходный файл: `if.coffee`)

```
the statement was true
```

Несмотря на искусственность примера, он наглядно демонстрирует структуру инструкции `if`. Вслед за ключевым словом `if` следует выражение, проверяющее необходимое условие. Если это выражение возвращает значение `true`, выполняется блок программного кода, следующий за ним. Если выражение возвращает значение `false`, этот блок кода пропускается.

Ниже приводится менее искусственный пример:

Пример: (исходный файл: `if2.coffee`)

```
today = "Sunday"
if today is "Sunday"
    console.log "Today is Sunday"
```

Пример: (исходный файл: `if2.js`)

```
(function() {
    var today;
    today = "Sunday";
    if (today === "Sunday") console.log("Today is Sunday");
}).call(this);
```

Вывод: (исходный файл: `if2.coffee`)

```
Today is Sunday
```

Инструкция if/else

Иногда бывает необходимо выполнить некоторые операции, когда условное выражение возвращает истинное значение, и другие операции, когда оно возвращает ложное значение. В подобных случаях можно использовать уже знакомую нам условную инструкцию `if` и ключевое слово `else`, определяющее блок программного кода,

выполняемого, когда условное выражение возвращает ложное значение.

Например:

Пример: (исходный файл: `if_else.coffee`)

```
today = "Monday"
if today is "Sunday"
    console.log "Today is Sunday"
else
    console.log "Today is not Sunday"
```

Пример: (исходный файл: `if_else.js`)

```
(function() {
    var today;
    today = "Monday";
    if (today === "Sunday") {
        console.log("Today is Sunday");
    } else {
        console.log("Today is not Sunday");
    }
}).call(this);
```

Вывод: (исходный файл: `if_else.coffee`)

```
Today is not Sunday
```

Как видите, условное выражение в нашем примере вернуло ложное значение, поэтому был выполнен блок кода, следующий за ключевым словом `else`.

В языке CoffeeScript этот фрагмент можно записать в одну строку, как показано ниже:

Пример: (исходный файл: `if_else_1_line.coffee`)

```
today = "Monday"
console.log if today is "Sunday" then "Today is Sunday" else "Today is not
↪Sunday"
```

Пример: (исходный файл: `if_else_1_line.js`)

```
(function() {
    var today;
```

```
    today = "Monday";
    console.log(today === "Sunday" ? "Today is Sunday" : "Today is not ↪Sunday");
}).call(this);
```

Вывод: (исходный файл: `if_else_1_line.coffee`)

```
Today is not Sunday
```

> **Совет.** Я считаю, что однострочный вариант инструкции `if then else` сложнее для восприятия, поэтому я крайне редко пользуюсь такой возможностью.

Для записи того же примера в одну строку в языке JavaScript можно использовать оператор, называемый тернарным (трехместным) [4], в который собственно и компилируется программный код на CoffeeScript. Однако CoffeeScript не поддерживает тернарный оператор, поэтому любые попытки использовать его дают в результате весьма странный код на JavaScript. Взгляните сами:

Пример: (исходный файл: `ternary.coffee`)

```
today = "Monday"
console.log today is "Sunday" ? "Today is Sunday" : "Today is not Sunday"
```

Пример: (исходный файл: `ternary.js`)

```
(function() {
    var today, _ref;
    today = "Monday";
    console.log((_ref = today === "Sunday") != null ? _ref : {
        "Today is Sunday": "Today is not Sunday"
    });
}).call(this);
```

Вывод: (исходный файл: `ternary.coffee`)

```
false
```

Я не буду объяснять, что здесь произошло. Сделайте себе пометку, чтобы вернуться к этому примеру по окончании чтения этой книги и попытаться разрешить загадку, почему компилятор CoffeeScript сгенерировал такой код на JavaScript.

Инструкция if/else if

Представим на мгновение, что мы пишем очень простое приложение для ведения списка дел. Нам нужно чтобы в субботу приложение выводило список дел на этот день, в воскресенье предлагало расслабиться и насладиться выходным днем и, наконец, в любой другой день советовало бы сконцентрироваться на работе. Как можно реализовать такую функциональность, используя знания, полученные до сих пор? Реализация могла бы выглядеть примерно так:

Пример: (исходный файл: `if_else_if_1.coffee`)

```
today = "Monday"
if today is "Saturday"
    console.log "Here are your todos for the day..."
if today is "Sunday"
    console.log "Go watch football and relax!"
if today isnt "Saturday" and today isnt "Sunday"
    console.log "Get to work you lazy bum!"
```

Пример: (исходный файл: `if_else_if_1.js`)

```
(function() {
    var today;
    today = "Monday";
    if (today === "Saturday") console.log("Here are your todos for the
↪day...");
    if (today === "Sunday") console.log("Go watch football and relax!");
    if (today !== "Saturday" && today !== "Sunday") {
        console.log("Get to work you lazy bum!");
    }
}).call(this);
```

Вывод: (исходный файл: `if_else_if_1.coffee`)

```
Get to work you lazy bum!
```

Хотя такая реализация вполне работоспособна, она далеко не самая эффективная. Во-первых, в ней выполняются все проверки, в каждой условной инструкции `if`. Даже если первое условное выражение вернет `true`, остальные инструкции `if` все равно вычислят значения своих условных выражений, хотя нам заранее известно, что в этом случае выражения вернут `false`. Кроме того, последняя

инструкция if повторно выполняет проверку, аналогичную той, что выполнялась в предыдущей инструкции. Наконец, в такой реализации легко допустить ошибку. Если потребуется перейти от использования полных названий дней недели, таких как Sunday (воскресенье), к кратким, таким как Sun (вск), нам придется отыскать и изменить все вхождения названия Sunday.

С помощью инструкции else if легко можно устранить лишнюю проверку и повысить эффективность реализации. Например:

Пример: (исходный файл: if_else_if_2.coffee)

```
today = "Monday"
if today is "Saturday"
    console.log "Here are your todos for the day..."
else if today is "Sunday"
    console.log "Go watch football and relax!"
else
     console.log "Get to work you lazy bum!"
```

Пример: (исходный файл: if_else_if_2.js)

```
(function() {
    var today;
    today = "Monday";
    if (today === "Saturday") {
        console.log("Here are your todos for the day...");
    } else if (today === "Sunday") {
        console.log("Go watch football and relax!");
    } else {
        console.log("Get to work you lazy bum!");
    }
}).call(this);
```

Вывод: (исходный файл: if_else_if_2.coffee)

```
Get to work you lazy bum!
```

Такой код выглядит намного лучше, разве не так? Кроме того, он существенно эффективнее. Например, если предположить, что сегодня суббота (Saturday), выполнится только первый блок кода, а остальные инструкции else if и else будут пропущены, так как выполнять их нет необходимости.

Далее в этой главе мы перепишем этот пример, применив инструкцию switch, что позволит сделать реализацию еще более прозрачной.

Инструкция unless

В языке Ruby имеется инструкция `unless`, и парни из команды разработки CoffeeScript, решив, что неплохо было бы иметь такую инструкцию, умыкнули ее.

Как действует инструкция `unless`? Если говорить в двух словах, она позволяет вставлять ключевое слово `else` перед инструкцией `if`. Я представляю, как вы чешете себе затылок. Поначалу такое объяснение кажется непонятным, но в действительности все просто.

Инструкция `unless` проверяет, не вернуло ли условное выражение ложное значение. Если выражение вернуло ложное значение, выполняется блок кода, следующий за инструкцией. Например:

Пример: (исходный файл: `unless.coffee`)

```
today = "Monday"
unless today is "Sunday"
    console.log "No football today!"
```

Пример: (исходный файл: `unless.js`)

```
(function() {
    var today;
    today = "Monday";
    if (today !== "Sunday") console.log("No football today!");
}).call(this);
```

Вывод: (исходный файл: `unless.coffee`)

```
No football today!
```

Этот же пример можно было бы реализовать любым из следующих способов:

Пример: (исходный файл: `unless_as_if.coffee`)

```
today = "Monday"
unless today is "Sunday"
    console.log "No football today!"
if not (today is "Sunday")
    console.log "No football today!"
if today isnt "Sunday"
    console.log "No football today!"
```

Пример: (исходный файл: unless_as_if.js)

```
(function() {
    var today;
    today = "Monday";
    if (today !== "Sunday") console.log("No football today!");
    if (!(today === "Sunday")) console.log("No football today!");
    if (today !== "Sunday") console.log("No football today!");
}).call(this);
```

Вывод: (исходный файл: `unless_as_if.coffee`)

```
No football today!
No football today!
No football today!
```

Из трех примеров мне больше нравится последний, использующий псевдоним `isnt`. На мой взгляд он выглядит проще и легче читается. Однако, выбирать вам. Все три являются вполне допустимыми способами реализации одного и того же алгоритма.

Встроенные условные инструкции

Помимо ключевого слова `unless`, команда разработчиков CoffeeScript свистнула из языка Ruby идею *встроенных условных инструкций*. Встроенная условная инструкция позволяет совместить в одной строке условную инструкцию и блок кода.

Это проще продемонстрировать на примере:

Пример: (исходный файл: `inline.coffee`)

```
today = "Sunday"
console.log "Today is Sunday" if today is "Sunday"
```

Пример: (исходный файл: `inline.js`)

```
(function() {
    var today;
    today = "Sunday";
    if (today === "Sunday") console.log("Today is Sunday");
}).call(this);
```

Вывод: (исходный файл: `inline.coffee`)

```
Today is Sunday
```

При использовании в паре с оператором проверки существования, описанным выше в этой главе, встроенная условная инструкция помогает сделать программный код еще более прозрачным.

Инструкции switch/when

Выше, когда мы рассматривали инструкции `else if`, я упомянул, что с помощью инструкции `switch` [5] программный код можно сделать еще прозрачнее. Инструкция `switch` позволяет конструировать таблицы условных инструкций, выполняющих сопоставление с объектом. При обнаружении совпадения с одной из них, выполняется соответствующий блок кода. В таблицу можно также добавить вариант `else`, соответствующий случаю, когда не обнаруживается совпадения ни с одним из условий.

Вернемся к нашему примеру с инструкциями `else if` и перепишем его с применением инструкции `switch`.

Пример: (исходный файл: `switch1.coffee`)

```
today = "Monday"
switch today
    when "Saturday"
         console.log "Here are your todos for the day..."
    when "Sunday"
         console.log "Go watch football and relax!"
    else
         console.log "Get to work you lazy bum!"
```

Пример: (исходный файл: `switch1.js`)

```
(function() {
    var today;
    today = "Monday";
    switch (today) {
        case "Saturday":
            console.log("Here are your todos for the day...");
            break;
        case "Sunday":
            console.log("Go watch football and relax!");
            break;
        default:
```

```
            console.log("Get to work you lazy bum!");
    }
}).call(this);
```

Вывод: (исходный файл: switch1.coffee)

```
Get to work you lazy bum!
```

В этом примере, значение переменной today сопоставляется с вариантами when. Поскольку текущее значение переменной today, переданное нами, не соответствует ни одному из вариантов, управление передается варианту else, объявленному в конце инструкции switch. Следует отметить, что блок else в конце не является обязательным. Если бы мы не определили блок else, наш пример ничего бы не вывел.

Наиболее наблюдательные могли заметить в скомпилированном коде на JavaScript ключевое слово break в конце каждой инструкции case. В языке JavaScript имеется возможность продолжать сопоставление с другими вариантами case, находящимися ниже. Эта возможность практически никогда не используется и является источником большого количества ошибок. Наиболее типичная ошибка заключается в том, что при обнаружении совпадения выполняется соответствующий блок программного кода, а затем продолжается сопоставление с остальными вариантами и завершается выполнением блока для варианта default в конце инструкции. Ключевое слово break сообщает инструкции switch, что необходимо прервать выполнение и прекратить сопоставление с другими вариантами. Это еще один пример того, как язык CoffeeScript пытается прикрыть вашу спину и сократить вероятность появления ошибок.

Инструкция switch позволяет также указывать в вариантах when списки значений, разделенных запятыми, с которыми требуется сопоставить проверяемое значение. Это дает возможность организовать выполнение одного и того же блока кода для разных проверяемых значений. Допустим, нам потребовалось проверить, не является ли сегодняшний день недели выходным. В выходной день программа должна сообщить, что можно отдохнуть, а в остальные дни – что мы должны работать. Реализовать это можно так:

Пример: (исходный файл: switch2.coffee)

```
today = "Sunday"
switch today
```

```
    when "Saturday", "Sunday"
         console.log "Enjoy your #{today}!"
    else
         console.log "Off to work you go. :("
```

Пример: (исходный файл: `switch2.js`)

```
(function() {
    var today;
    today = "Sunday";
    switch (today) {
        case "Saturday":
        case "Sunday":
            console.log("Enjoy your " + today + "!");
            break;
        default:
            console.log("Off to work you go. :(");
    }
}).call(this);
```

Вывод: (исходный файл: `switch2.coffee`)

```
Enjoy your Sunday!
```

На этом исследование замечательной инструкции `switch` подошло к концу. Среди разработчиков постоянно ведутся споры по поводу когда, где и как использовать инструкцию `switch`. Она не относится к числу инструкций, которые я использую каждый день, но у нее определенно есть своя ниша. Я не буду углубляться в этот вопрос и оставляю за вами право самим решать, когда использовать ее в своих приложениях.

В заключение

В этой главе мы охватили множество базовых сведений. Мы рассмотрели различные операторы, имеющиеся в языке CoffeeScript, и как они отображаются в программный код на языке JavaScript. Мы увидели несколько ситуаций, где CoffeeScript пытается помогать писать более качественный программный код на JavaScript. Псевдонимы показали, как писать «привлекательный» программный код, более напоминающий текст на английском языке, нежели программу. Вы увидели, как конструировать условные инструкции, чтобы помочь

своим программам принимать более интеллектуальные решения и выполнять те или иные блоки кода в зависимости от конкретных условий. Наконец, мы узнали, как инструкция `switch` может помочь сделать сложный программный код более прозрачным.

В первоначальном варианте этой книги данная глава являлась частью главы 2, «Основы», но я понял, что она содержит достаточно большой объем информации, чтобы ее можно было выделить в отдельную главу. Эту главу можно было бы назвать «Основы – часть 2». Я сообщаю об этом, потому что получив знания, содержащиеся в этой главе и в главе 2, мы заложили фундамент для дальнейшего изучения языка CoffeeScript. Теперь мы готовы перейти к изучению по-настоящему интересных возможностей.

Примечания

1. http://ru.wikipedia.org/wiki/Оператор_(программирование).
2. http://ru.wikipedia.org/wiki/_Оператор_ветвления.
3. http://www.yaml.org/spec/1.2/spec.html.
4. http://ru.wikipedia.org/wiki/Тернарная_условная_операция.
5. http://en.wikipedia.org/wiki/Switch_statement.

4. Функции и аргументы

В этой главе мы рассмотрим одну из важнейших особенностей любого языка программирования, функции. *Функции* позволяют выделить дискретные блоки программного кода, пригодные для многократного использования. Без функций программа легко может превратиться в длинную, трудночитаемую и неудобную в сопровождении мешанину из программного кода.

У меня было желание дать вам пример, демонстрирующий, как мог бы выглядеть программный код на JavaScript, если бы у нас не было возможности использовать функции, но у меня ничего не получилось. Даже простейший пример преобразования всех символов строки в нижний регистр требует использования функций JavaScript.

Поскольку я не могу показать пример, в котором не используются функции, я продемонстрирую пример на языке CoffeeScript, использующий одну или две вспомогательных функции, чтобы вы могли осознать, насколько важную роль играют функции в обеспечении удобства сопровождения программного кода.

Пример: (исходный файл: `no_functions_example.coffee`)

```
tax_rate = 0.0625
val = 100
console.log "What is the total of $#{val} worth of shopping?"
tax = val * tax_rate
total = val + tax
console.log "The total is #{total}"
val = 200
console.log "What is the total of $#{val} worth of shopping?"
tax = val * tax_rate
total = val + tax
console.log  "The  total  is  #{total}"
```

Пример: (исходный файл: `no_functions_example.js`)

```
(function() {
    var tax, tax_rate, total, val;
```

```
    tax_rate = 0.0625;
    val = 100;
    console.log("What is the total of $" + val + " worth of shopping?");
    tax = val * tax_rate;
    total = val + tax;
    console.log("The total is " + total);
    val = 200;
    console.log("What is the total of $" + val + " worth of shopping?");
    tax = val * tax_rate;
    total = val + tax;
    console.log("The total is " + total);
}).call(this);
```

Вывод: (исходный файл: `no_functions_example.coffee`)

```
What is the total of $100 worth of shopping?
The total is 106.25
What is the total of $200 worth of shopping?
The  total  is  212.5
```

В этом примере вычисляется общая стоимость товара, продажа которого облагается определенным налогом. Помимо простоты примера, в нем можно заметить, что блок кода, вычисляющий общую сумму, повторяется несколько раз.

Выделим этот блок кода, оформим его в виде функции и попробуем сделать пример более прозрачным.

Пример: (исходный файл: `with_functions_example.coffee`)

```
default_tax_rate = 0.0625
calculateTotal = (sub_total, rate = default_tax_rate) ->
    tax = sub_total * rate
    sub_total + tax
val = 100
console.log "What is the total of $#{val} worth of shopping?"
console.log "The total is #{calculateTotal(val)}"
val = 200
console.log "What is the total of $#{val} worth of shopping?"
console.log  "The  total  is  #{calculateTotal(val)}"
```

Пример: (исходный файл: `with_functions_example.js`)

```
(function() {
    var calculateTotal, default_tax_rate, val;
```

```
    default_tax_rate = 0.0625;
    calculateTotal = function(sub_total, rate) {
        var tax;
        if (rate == null) rate = default_tax_rate;
        tax = sub_total * rate;
        return sub_total + tax;
    };
    val = 100;
    console.log("What is the total of $" + val + " worth of shopping?");
    console.log("The total is " + (calculateTotal(val)));
    val = 200;
    console.log("What is the total of $" + val + " worth of shopping?");
    console.log("The total is " + (calculateTotal(val)));
}).call(this);
```

Вывод: (исходный файл: `with_functions_example.coffee`)

```
What is the total of $100 worth of shopping?
The total is 106.25
What is the total of $200 worth of shopping?
The total is 212.5
```

Возможно, кое-что из сделанного здесь вам непонятно, но не волнуйтесь, именно для этого и написана данная глава. Однако, даже не зная специфики определения и использования функций в языке CoffeeScript, уже можно видеть, насколько прозрачнее стал программный код по сравнению с первым примером. В реорганизованной версии появилась даже возможность использовать различные коэффициенты налогов, если это потребуется. Это позволяет сохранить программный код «сухим» (принцип программирования DRY, Don't Repeat Yourself – не повторяйся)[1]. Исключение из программного кода повторяющихся фрагментов делает его более простым в сопровождении и снижает вероятность появления ошибок.

Основы функций

Начнем с самого важного – как определить функцию на языке CoffeeScript. В простейшем случае *определение* и *вызов* функции выглядят так:

Пример: (исходный файл: `simple_function.coffee`)

[1] Здесь игра слов: DRY переводится, как «сухой». – *Прим. перев.*

```
myFunction = ()->
    console.log "do some work here"
myFunction()
```

Пример: (исходный файл: `simple_function.js`)

```
(function() {
    var myFunction;
    myFunction = function() {
        return console.log("do some work here");
    };
    myFunction();
}).call(this);
```

В этом примере мы создали функцию с именем `myFunction` и определили блок кода, реализующий ее. Тело функции – программный код с отступом, следующий за стрелкой `->`, подчиняется правилу значимых пробелов, о котором рассказывалось в главе 2, «Основы».

Данная функция не принимает аргументов. Об этом свидетельствуют пустые круглые скобки, предшествующие стрелке `->`. В языке CoffeeScript требуется указывать круглые скобки, при вызове функции без аргументов: `myFunction()`.

Поскольку функция не принимает аргументов, их можно опустить в определении, например:

Пример: (исходный файл: `simple_function_no_parens.coffee`)

```
myFunction = ->
    console.log "do some work here"
myFunction()
```

Пример: (исходный файл: `simple_function_no_parens.js`)

```
(function() {
    var myFunction;
    myFunction = function() {
        return console.log("do some work here");
    };
    myFunction();
}).call(this);
```

Записать эту простую функцию можно более чем одним способом. Тело функции состоит из единственной строки, поэтому все определение функции можно свернуть в одну строку, например:

Пример: (исходный файл: `simple_function_one_line.coffee`)

```
myFunction = -> console.log "do some work here"
myFunction()
```

Пример: (исходный файл: `simple_function_one_line.js`)

```
(function() {
    var myFunction;
    myFunction = function() {
        return console.log("do some work here");
    };
    myFunction();
}).call(this);
```

Все три примера выше дают в результате один и тот же программный код на JavaScript и действуют совершенно одинаково.

Совет. Несмотря на возможность создавать однострочные определения функций, я предпочитаю этого не делать. Лично я не считаю, что такой прием делает программный код более прозрачным или удобочитаемым. Кроме того, поместив тело функции в отдельную строку, ее проще будет расширить в будущем. Обратите также внимание, что последняя строка в каждой из функций на языке JavaScript содержит ключевое слово `return`. Компилятор CoffeeScript автоматически добавляет его. Независимо от того, что делает последняя строка в функции, ее значение станет возвращаемым значением. Эта особенность напоминает такие языки, как Ruby. Поскольку компилятор CoffeeScript автоматически добавляет ключевое слово `return` в скомпилированный программный код на JavaScript, использовать его в программном коде на CoffeeScript необязательно.

Совет. Я уверен, что добавление ключевого слова `return` иногда может добавить ясности в программный код. Используйте его, если вам покажется, что оно сделает ваш код более простым для чтения и понимания.

Совет. Если необходимо, чтобы функция не возвращала результат выполнения последней строки в теле функции, следует добавить свою последнюю строку с инструкцией `return`. Иногда с этой ролью прекрасно справляется инструкция `return null` или `return undefined`.

Аргументы

Как и в языке JavaScript, функции в языке CoffeeScript могут принимать *аргументы*. С помощью аргументов функциям можно передавать объекты, чтобы они могли производить вычисления, манипулировать данными или выполнять другие необходимые операции.

Определение функции на языке CoffeeScript, принимающей аргументы, не сильно отличается от определения аналогичной функции на языке JavaScript. Внутри круглых скобок через запятую перечисляются имена аргументов, принимаемых функцией.

Пример: (исходный файл: `function_with_args.coffee`)

```
calculateTotal = (sub_total, rate) ->
    tax = sub_total * rate
    sub_total + tax
console.log calculateTotal(100, 0.0625)
```

Пример: (исходный файл: function_with_args.js)

```
(function() {
    var calculateTotal;
    calculateTotal = function(sub_total, rate) {
        var tax;
        tax = sub_total * rate;
        return sub_total + tax;
    };
    console.log(calculateTotal(100, 0.0625));
}).call(this);
```

Вывод: (исходный файл: `function_with_args.coffee`)

```
106.25
```

Здесь мы определили функцию, принимающую два аргумента и выполняющую некоторые вычисления для получения общей суммы. При вызове этой функции, мы передали ей два значения, которые она должна использовать в вычислениях.

В главе 2 мы коротко рассмотрели правила, касающиеся использования круглых скобок в языке CoffeeScript. Мне хотелось бы повторить одно из этих правил. Поскольку функция принимает аргументы, мы можем опустить скобки в *вызове функции*. То есть, предыдущий пример можно записать так:

Пример: (исходный файл: function_with_args_no_parens.coffee)

```
calculateTotal = (sub_total, rate) ->
    tax = sub_total * rate
    sub_total + tax
console.log calculateTotal 100, 0.0625
```

Пример: (исходный файл: function_with_args_no_parens.js)

```
(function() {
    var calculateTotal;
    calculateTotal = function(sub_total, rate) {
        var tax;
        tax = sub_total * rate;
        return sub_total + tax;
    };
    console.log(calculateTotal(100, 0.0625));
}).call(this);
```

Вывод: (исходный файл: function_with_args_no_parens.coffee)

```
106.25
```

Как видите, компилятор CoffeeScript скомпилировал корректный код на JavaScript, добавив скобки там, где они необходимы.

> **Совет.** Тема использования круглых скобок в вызовах функций горячо обсуждается в мире CoffeeScript. Лично я предпочитаю использовать их. Я считаю, что это делает программный код более понятным и снижает вероятность появления ошибок из-за неправильной расстановки скобок компилятором. При появлении сомнений, всегда используйте круглые скобки. Вы не пожалеете об этом.

Аргументы со значениями по умолчанию

В некоторых языках программирования, таких как Ruby, имеется возможность присваивать *аргументам значения по умолчанию*. То есть, если по каким-то причинам некоторые аргументы не были переданы функции, они получат свои значения по умолчанию.

Вернемся к примеру реализации калькулятора. Перепишем его так, чтобы аргумент, определяющий размер налога, имел значение по умолчанию, и его можно было бы не передавать функции:

Пример: (исходный файл: default_args.coffee)

```
calculateTotal = (sub_total, rate = 0.05) ->
    tax = sub_total * rate
    sub_total + tax
console.log calculateTotal 100, 0.0625
console.log  calculateTotal  100
```

Пример: (исходный файл: default_args.js)

```
(function() {
    var calculateTotal;
    calculateTotal = function(sub_total, rate) {
        var tax;
        if (rate == null) rate = 0.05;
        tax = sub_total * rate;
        return sub_total + tax;
    };
    console.log(calculateTotal(100, 0.0625));
    console.log(calculateTotal(100));
}).call(this);
```

Вывод: (исходный файл: default_args.coffee)

```
106.25
105
```

В определении функции мы сообщили компилятору CoffeeScript, что аргумент tax_rate имеет значение по умолчанию, равное 0.05. В первом вызове функции calculateTotal мы передаем ей аргумент tax_rate со значением 0.0625; во втором вызове мы опустили аргумент tax_rate, что было замечено реализацией и функция использовала соответствующее значение по умолчанию 0.05.

Можно пойти еще дальше и в качестве значений по умолчанию использовать ссылки на другие аргументы. Взгляните на следующий пример:

Пример: (исходный файл: default_args_referring.coffee)

```
href = (text, url = text) ->
    html = "<a href='#{url}'>#{text}</a>"
    return html
console.log href("Click Here", "http://www.example.com")
console.log  href("http://www.example.com")
```

Пример: (исходный файл: default_args_referring.js)

```
(function() {
    var href;
    href = function(text, url) {
        var html;
        if (url == null) url = text;
        html = "<a href='" + url + "'>" + text + "</a>";
        return html;
    };
    console.log(href("Click Here", "http://www.example.com"));
    console.log(href("http://www.example.com"));
}).call(this);
```

Вывод: (исходный файл: default_args_referring.coffee)

```
<a href='http://www.example.com'>Click Here</a>
<a  href='http://www.example.com'>http://www.example.com</a>
```

Когда эта функция вызывается без аргумента url, он принимает значение аргумента text.

В качестве значений аргументов по умолчанию можно также использовать функции. Поскольку функция, определяющая значение по умолчанию, будет вызываться, только в случае отсутствия аргумента, такой подход нельзя отнести к разряду проблем, связанных с производительностью.

Пример: (исходный файл: default_args_with_function.coffee)

```
defaultRate = -> 0.05
calculateTotal = (sub_total, rate = defaultRate()) ->
    tax = sub_total * rate
    sub_total + tax
console.log calculateTotal 100, 0.0625
console.log  calculateTotal  100
```

Пример: (исходный файл: default_args_with_function.js)

```
(function() {
    var calculateTotal, defaultRate;
    defaultRate = function() {
        return 0.05;
    };
    calculateTotal = function(sub_total, rate) {
```

```
        var tax;
        if (rate == null) rate = defaultRate();
        tax = sub_total * rate;
        return sub_total + tax;
    };
    console.log(calculateTotal(100, 0.0625));
    console.log(calculateTotal(100));
}).call(this);
```

Вывод: (исходный файл: `default_args_with_function.coffee`)

```
106.25
105
```

> **Совет.** При использовании аргументов со значениями по умолчанию важно помнить, что они должны быть последними в списке аргументов. Вполне допустимо, когда функция имеет несколько аргументов со значениями по умолчанию, при условии, что все они замыкают список аргументов.

Групповые аргументы...

Иногда, при разработке функций, заранее неизвестно, сколько аргументов потребуется. Иногда может потребоваться один аргумент, иногда – два, а иногда – сотня. Чтобы упростить решение этой проблемы, в языке CoffeeScript предусмотрена возможность использовать в функциях *групповые аргументы*. Групповые аргументы обозначаются добавлением троеточия (`...`) после имени аргумента.

> **Совет.** Отличный способ запомнить порядок использования групповых аргументов – интерпретировать троеточие, как фразу «и другие...». Это не только упростит запоминание, но и придаст программному коду особый шарм, если вы будете использовать аргумент с именем `etc...` (и другие...).

Когда следует использовать групповые аргументы? Их можно использовать во всех случаях, когда функция может принимать переменное число аргументов. Прежде чем взглянуть на подробный пример, рассмотрим простую функцию, принимающую групповой аргумент:

Пример: (исходный файл: `splats.coffee`)

```
splatter = (etc...) ->
    console.log "Length: #{etc.length}, Values: #{etc.join(', ')}"
splatter()
splatter("a", "b", "c")
```

Пример: (исходный файл: `splats.js`)

```
(function() {
    var splatter,
        __slice = Array.prototype.slice;
    splatter = function() {
        var etc;
        etc = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
        return console.log("Length: " + etc.length + ", Values: " +
↪(etc.join(', ')));
    };
    splatter();
    splatter("a", "b", "c");
}).call(this);
```

Вывод: (исходный файл: `splats.coffee`)

```
Length: 0, Values:
Length: 3, Values: a, b, c
```

Как видите, сколько бы аргументов ни передавалось функции, они автоматически помещаются в массив, и если функции не передается ни одного аргумента, она получает пустой массив.

> **Совет.** Групповые аргументы являются отличным примером возможностей, достижимых в языке JavaScript, но требующих написать немалый объем шаблонного программного кода для их реализации. Взгляните, как в скомпилированном коде на JavaScript реализован групповой аргумент и вы согласитесь, что писать шаблонный код – не самое увлекательное занятие.

В отличие от других языков программирования, поддерживающих подобные конструкции, CoffeeScript не принуждает помещать групповой аргумент в конец списка аргументов. Групповые аргументы допускается помещать в любое место в списке аргументов.

Единственное ограничении – функция может иметь только один групповой аргумент.

Чтобы доказать, что групповой аргумент может находиться в любом месте в списке аргументов, реализуем метод, который будет принимать произвольное количество аргументов и возвращать строку. При конструировании строки, символы в первом и последнем аргументах мы преобразуем в верхний регистр, а символы в остальных аргументах – в нижний. Затем мы объединим аргументы в строку, перечислив их через символ слеша.

Пример: (исходный файл: `splats_arg_join.coffee`)

```
joinArgs = (first, middles..., last) ->
    parts = []
    if first?
        parts.push first.toUpperCase()
    for middle in middles
        parts.push middle.toLowerCase()
    if last?
        parts.push last.toUpperCase()
    parts.join('/')
console.log joinArgs("a")
console.log joinArgs("a", "b")
console.log  joinArgs("a",  "B",  "C",  "d")
```

Пример: (исходный файл: `splats_arg_join.js`)

```
(function() {
    var joinArgs,
        __slice = Array.prototype.slice;
    joinArgs = function() {
        var first, last, middle, middles, parts, _i, _j, _len;
        first = arguments[0], middles = 3 <= arguments.length ?
↪__slice.call(arguments, 1, _i = arguments.length - 1) : (_i = 1, []), last
↪= arguments[_i++];
        parts = [];
        if (first != null) parts.push(first.toUpperCase());
        for (_j = 0, _len = middles.length; _j < _len; _j++) {
            middle = middles[_j];
            parts.push(middle.toLowerCase());
        }
        if (last != null) parts.push(last.toUpperCase());
        return parts.join('/');
    };
```

```
    console.log(joinArgs("a"));
    console.log(joinArgs("a", "b"));
    console.log(joinArgs("a", "B", "C", "d"));
}).call(this);
```

Вывод: (исходный файл: `splats_arg_join.coffee`)

```
A
A/B
A/b/c/D
```

Признаю, что это довольно сложный пример, но он прекрасно иллюстрирует, как действуют групповые аргументы. Когда вызывается функция `joinArgs`, первый аргумент функции присваивается переменной `first`, последний аргумент – переменной `last`, а все остальные аргументы между первым и последним помещаются в массив `middles`.

> **Совет.** Точно так же можно было бы написать функцию, принимающую единственный групповой аргумент и извлекающую первый и последний элемент из массива `middles`, а это означает, что совершенно необязательно было писать весь этот код. Вот это праздник!

Наконец, функции с групповым аргументом может потребоваться передать значения элементов массива в виде отдельных аргументов. Такое тоже возможно.

Взгляните на следующий пример:

Пример: (исходный файл: `splats_array.coffee`)

```
splatter = (etc...) ->
    console.log "Length: #{etc.length}, Values: #{etc.join(', ')}"
a = ["a", "b", "c"]
splatter(a)
splatter(a...)
```

Пример: (исходный файл: `splats_array.js`)

```
(function() {
    var a, splatter,
        __slice = Array.prototype.slice;
    splatter = function() {
        var etc;
```

```
        etc = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
        return console.log("Length: " + etc.length + ", Values: " +
↪(etc.join(', ')));
    };
    a = ["a", "b", "c"];
    splatter(a);
    splatter.apply(null, a);
}).call(this);
```

Вывод: (исходный файл: `splats_array.coffee`)

```
Length: 1, Values: a,b,c
Length: 3, Values: a, b, c
```

В этом примере мы сначала попробовали передать функции `splatter` массив, но она видит его, как единственный аргумент, потому что он таковым и является. Однако, если добавить троеточие `...` после имени массива в вызове функции, компилятор CoffeeScript разобьет его на отдельные аргументы и передаст их функции.

В заключение

Теперь вы знаете все, что следует знать о функциях в языке CoffeeScript! Сначала мы узнали, как определить простую функцию. В действительности мы познакомились с несколькими способами определения функций в языке CoffeeScript. Затем мы посмотрели, как определять аргументы функций и как вызывать функции, включая дополнительные пояснения о том, когда, где и как следует или не следует использовать круглые скобки при вызове функций. Мы также рассмотрели аргументы со значениями по умолчанию – одну из моих самых любимых особенностей языка CoffeeScript.

Наконец, мы исследовали групповые аргументы и как они позволяют писать функции, принимающие переменное число аргументов.

Совершив короткий тур по функциям и аргументам, можно двигаться к следующей остановке – главе 5, «Коллекции и итерации». Итак, сходите, прихватите чего-нибудь холодненького и встретимся там. Готовы?

Примечания

1. http://en.wikipedia.org/wiki/DRY.

5. Коллекции и итерации

Коллекции являются важнейшим элементом практически любого объектно-ориентированного языка программирования. Они позволяют хранить множество значений в виде списков, таких как массивы, или в виде пар ключ/значение, подобно объектам в языке JavaScript. В виде объектов можно представлять такие элементы, как книги, и присваивать им свойства, такие как название, автор и дата публикации. В виде массивов можно хранить списки объектов, например, представляющих книги.

Рука об руку с коллекциями идут итераторы. Итераторы позволяют выполнять обход коллекций, таких как список книг, и выводить на экран информацию о каждой из них, изменять сведения о каждой книге или выполнять другие необходимые операции.

В первой половине этой главы мы познакомимся с массивами и объектами в языке CoffeeScript. Мы увидим, как они отображаются в аналогичные конструкции в языке JavaScript. Поскольку в этой книге речь идет о языке CoffeeScript, мы также познакомимся с некоторыми замечательными особенностями, доступными при работе с массивами и объектами.

Во второй половине главы мы сосредоточимся на итераторах. Мы возьмем все знания о коллекциях, полученные в первой части, и посмотрим, как выполнять их обход и манипулировать ими.

Массивы

Не углубляясь в технические подробности организации массивов в памяти, будем считать, что *массив* – это простая структура данных, предназначенная для хранения данных в виде последовательного списка. Новые элементы добавляются в конец этого списка, если явно не оговаривается иное. Массивы в языке CoffeeScript ничем не отличаются от массивов в языке JavaScript. Нумерация элементов в массивах CoffeeScript начинается с нуля и организованы они точно так же, как и их аналоги в JavaScript.

Пример: (исходный файл: array1.coffee)

```
myArray = ["a", "b", "c"]
console.log  myArray
```

Пример: (исходный файл: array1.js)

```
(function() {
    var myArray;
    myArray = ["a", "b", "c"];
    console.log(myArray);
}).call(this);
```

Вывод: (исходный файл: array1.coffee)

```
[ 'a', 'b', 'c' ]
```

За исключением завершающей точки с запятой и ключевого слова var, массивы в языках CoffeeScript и JavaScript реализованы практически идентично. Однако их идентичность не означает, что в рукаве у CoffeeScript не припрятаны какие-нибудь хитрости, приготовленные для работы с массивами.

В языке CoffeeScript можно объявлять массивы, не вставляя запятые между элементами, для чего достаточно поместить каждый элемент в отдельной строке:

Пример: (исходный файл: array2.coffee)

```
myArray = [
            "a"
            "b"
            "c"
          ]
console.log  myArray
```

Пример: (исходный файл: array2.js)

```
(function() {
    var myArray;
    myArray = ["a", "b", "c"];
    console.log(myArray);
}).call(this);
```

Вывод: (исходный файл: array2.coffee)

```
[ 'a', 'b', 'c' ]
```

Такой способ записи определенно длиннее, но иногда разбиение определения массива на несколько строк делает программный код более удобочитаемым. Также допускается комбинировать использование запятых и новых строк, чтобы сделать программный код более простым для восприятия:

Пример: (исходный файл: array3.coffee)

```
myArray = [
            "a", "b", "c"
            "d", "e", "f"
            "g", "h", "i"
          ]
console.log  myArray
```

Пример: (исходный файл: array3.js)

```
(function() {
    var myArray;
    myArray = ["a", "b", "c", "d", "e", "f", "g", "h", "i"];
    console.log(myArray);
}).call(this);
```

Вывод: (исходный файл: array3.coffee)

```
[ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i' ]
```

Проверка на вхождение

К массивам в языке CoffeeScript могут применяться те же функции, что и к массивам в языке JavaScript. Однако в языке JavaScript есть несколько сложностей, возникающих при работе с массивами, которые пытается ликвидировать язык CoffeeScript. Некоторые из них мы рассмотрим далее в этой главе, когда будем говорить о диапазонах. А прямо сейчас я хотел бы поговорить об одной возможности в языке CoffeeScript, значительно упрощающей работу с массивами – *определение вхождения в массив определенного значения*.

Пример: (исходный файл: in_array.coffee)

```
myArray = ["a", "b", "c"]
if "b" in myArray
    console.log "I found 'b'."
unless "d" in myArray
    console.log "'d' was nowhere to be found."
```

Пример: (исходный файл: `in_array.js`)

```
(function() {
    var myArray,
        __indexOf = Array.prototype.indexOf || function(item) { for (var i =
↪0, l = this.length; i < l; i++) { if (i in this && this[i] === item) return
↪i; } return -1; };
    myArray = ["a", "b", "c"];
    if (__indexOf.call(myArray, "b") >= 0) console.log("I found 'b'.");
    if (__indexOf.call(myArray, "d") < 0) {
        console.log("'d' was nowhere to be found.");
    }
}).call(this);
```

Вывод: (исходный файл: `in_array.coffee`)

```
I found 'b'.
'd' was nowhere to be found.
```

Как видите, у нас имеется возможность проверить наличие определенного элемента в массиве с помощью ключевого слова `in`. В скомпилированном коде на JavaScript для этого создается функция, выполняющая обход массива и пытающаяся определить индекс искомого значения. Если это значение обнаруживается, она возвращает его индекс. Затем наш пример проверяет значение возвращаемого индекса, чтобы оно было больше либо равно нулю.

Присваивание с перестановкой

Рано или поздно возникает необходимость реализовать *обмен значениями* двух или более переменных. В одних языках это может оказаться сложнее, чем в других, но, к счастью, в CoffeeScript эта операция выполняется просто.

Взгляните, как выполнить обмен значениями двух переменных:

Пример: (исходный файл: `swap_assignment.coffee`)

```
x = "X"
y = "Y"
```

```
console.log "x is #{x}"
console.log "y is #{y}"
[x, y] = [y, x]
console.log "x is #{x}"
console.log "y is #{y}"
```

Пример: (исходный файл: `swap_assignment.js`)

```
(function() {
    var x, y, _ref;
    x = "X";
    y = "Y";
    console.log("x is " + x);
    console.log("y is " + y);
    _ref = [y, x], x = _ref[0], y = _ref[1];
    console.log("x is " + x);
    console.log("y is " + y);
}).call(this);
```

Вывод: (исходный файл: `swap_assignment.coffee`)

```
x is X
y is Y
x is Y
y is X
```

Для этой цели в языке CoffeeScript создаются два массива и затем выполняется присваивание этих массивов друг другу. В первом массиве, слева от оператора =, перечисляются переменные, которым должны быть присвоены новые значения. Во втором массиве, справа от оператора =, перечисляются новые значения, которые должны быть присвоены нашим переменным. Все остальное делает CoffeeScript.

Как видно из вывода значений переменных до и после перестановки, они действительно обменялись значениями.

Множественное, или реструктурирующее присваивание

Иногда приходится сталкиваться с функциями, возвращающими массив значений, которые желательно было бы присвоить отдельным переменным, чтобы упростить доступ к ним. Рассмотрим

пример из мира Ruby. В мире Ruby существует очень популярная библиотека Rack [1]. Функции из этой библиотеки реализуют упрощенный интерфейс между веб-серверами и прикладными фреймворками. Требования, предъявляемые этим интерфейсом, очень просты. Фреймворк должен вернуть массив. Первый элемент массива – код ответа протокола HTTP. Второй элемент – хеш[1] с HTTP-заголовками. Третий и последний элемент – тело ответа. Это лишь упрощенное описание того, как действует библиотека Rack, но для нас этого вполне достаточно.

Теперь попробуем написать функцию, возвращающую массив, соответствующий требованиям библиотеки Rack. А затем присвоим элементы этого массива некоторым переменным.

Пример: (исходный файл: `multiple_assignment.coffee`)

```
rack = ->
    [200, {"Content-Type": "text/html"}, "Hello Rack!"]
console.log rack()
[status, headers, body] = rack()
console.log "status is #{status}"
console.log "headers is #{JSON.stringify(headers)}"
console.log "body is #{body}"
```

Пример: (исходный файл: `multiple_assignment.js`)

```
(function() {
    var body, headers, rack, status, _ref;
    rack = function() {
        return [
            200, {
                "Content-Type": "text/html"
            }, "Hello Rack!"
        ];
    };
    console.log(rack());
    _ref = rack(), status = _ref[0], headers = _ref[1], body = _ref[2];
    console.log("status is " + status);
    console.log("headers is " + (JSON.stringify(headers)));
    console.log("body is " + body);
}).call(this);
```

[1] Ассоциативный массив. – *Прим. перев.*

Вывод: (исходный файл: multiple_assignment.coffee)

```
[ 200, { 'Content-Type': 'text/html' }, 'Hello Rack!' ]
status is 200
headers is {"Content-Type":"text/html"}
body is Hello Rack!
```

Здесь мы использовали тот же прием, что и при реализации присваивания с перестановкой. Слева от оператора = мы поместили массив с переменными, которым требуется присвоить значения. А справа – вызов функции, возвращающей массив значений, которые следует присвоить нашим переменным.

> **Совет.** Обратите внимание, что при присваивании с перестановкой и множественном присваивании не требуется предварительно объявлять переменные, участвующие в этой операции. Компилятор CoffeeScript сам позаботится об этом.

Для присваивания множества значений можно даже использовать синтаксис групповых аргументов, описанный в главе 4, «Функции и аргументы»:

Пример: (исходный файл: splat_assignment.coffee)

```
myArray = ["A", "B", "C", "D"]
[start, middle..., end] = myArray
console.log "start is #{start}"
console.log "middle is #{middle}"
console.log "end is #{end}"
```

Пример: (исходный файл: splat_assignment.js)

```
(function() {
    var end, middle, myArray, start, _i,
        __slice = Array.prototype.slice;
    myArray = ["A", "B", "C", "D"];
    start = myArray[0], middle = 3 <= myArray.length ? __slice.call(myArray,
↪1, _i = myArray.length - 1) : (_i = 1, []), end = myArray[_i++];
    console.log("start is " + start);
    console.log("middle is " + middle);
    console.log("end is " + end);
}).call(this);
```

Вывод: (исходный файл: splat_assignment.coffee)

```
start is A
middle is B,C
end  is  D
```

А что произойдет, если переменных окажется больше, чем значений? Давайте посмотрим:

Пример: (исходный файл: too_much_assignment.coffee)

```
myArray = ["A", "B"]
[a, b, c] = myArray
console.log "a is #{a}"
console.log "b is #{b}"
console.log  "c  is  #{c}"
```

Пример: (исходный файл: too_much_assignment.js)

```
(function() {
    var a, b, c, myArray;
    myArray = ["A", "B"];
    a = myArray[0], b = myArray[1], c = myArray[2];
    console.log("a is " + a);
    console.log("b is " + b);
    console.log("c is " + c);
}).call(this);
```

Вывод: (исходный файл: too_much_assignment.coffee)

```
a is A
b is B
c  is  undefined
```

Как видите, последняя переменная получила значение undefined, потому что для нее в массиве не оказалось значения.

Если переменных окажется меньше, чем значений в массиве, как в следующем примере, эти значения просто останутся не присвоенными.

Пример: (исходный файл: too_little_assignment.coffee)

```
myArray = ["A", "B", "C"]
[a, b] = myArray
```

```
console.log "a is #{a}"
console.log "b is #{b}"
```

Пример: (исходный файл: `too_little_assignment.js`)

```
(function() {
    var a, b, myArray;
    myArray = ["A", "B", "C"];
    a = myArray[0], b = myArray[1];
    console.log("a is " + a);
    console.log("b is " + b);
}).call(this);
```

Вывод: (исходный файл: `too_little_assignment.coffee`)

```
a is A
b is B
```

Диапазоны

Диапазоны в языке CoffeeScript позволяют легко заполнять массивы последовательностями чисел в диапазоне от начального до конечного значения. Для определения диапазонов используется следующий синтаксис:

Пример: (исходный файл: `range1.coffee`)

```
myRange = [1..10]
console.log myRange
```

Пример: (исходный файл: `range1.js`)

```
(function() {
    var myRange;
    myRange = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    console.log(myRange);
}).call(this);
```

Вывод: (исходный файл: `range1.coffee`)

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

Отделяя двумя точками (`..`) начальное число от последнего, мы получаем массив, заполненный последовательностью чисел от

первого до последнего включительно. Если потребуется создать диапазон, не включающий последнее число, вместо двух точек следует использовать три (...):

Пример: (исходный файл: range2.coffee)

```
myRange = [1...10]
console.log  myRange
```

Пример: (исходный файл: range2.js)

```
(function() {
    var myRange;
    myRange = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    console.log(myRange);
}).call(this);
```

Вывод: (исходный файл: range2.coffee)

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Как видите, заменив две точки (..) на три (...), мы исключили число 10 из массива.

С помощью диапазонов можно также создавать массивы, заполненные числовыми значениями *в порядке убывания*:

Пример: (исходный файл: range3.coffee)

```
myRange = [10..1]
console.log  myRange
```

Пример: (исходный файл: range3.js)

```
(function() {
    var myRange;
    myRange = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1];
    console.log(myRange);
}).call(this);
```

Вывод: (исходный файл: range3.coffee)

```
[ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
```

К обратным диапазонам применяется то же правило, при замене двух точек (..) на три (...).

Как видите, компилятор CoffeeScript производит программный код на JavaScript, содержащий полный массив, заполненный всеми числами из диапазона. А что если нам потребуется определить массив, содержащий сотню или тысячу чисел? Неужели компилятор CoffeeScript сгенерирует огромный фрагмент кода на JavaScript со списком из тысячи чисел между квадратными скобками? Конечно нет! Если компилятор CoffeeScript обнаружит, что количество элементов в диапазоне превышает некоторое пороговое значение, он произведет код, генерирующий элементы массива в цикле:

Пример: (исходный файл: `range4.coffee`)

```
myRange = [1..50]
console.log  myRange.join(", ")
```

Пример: (исходный файл: `range4.js`)

```
(function() {
    var myRange, _i, _results;
    myRange = (function() {
        _results = [];
        for (_i = 1; _i <= 50; _i++){ _results.push(_i); }
        return _results;
    }).apply(this);
    console.log(myRange.join(", "));
}).call(this);
```

Вывод: (исходный файл: `range4.coffee`)

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50
```

> **Совет.** Если кого-то интересует пороговое значение количества элементов в диапазоне, скажу, что оно равно 22. Я не знаю, почему 22, но это так. Если вы не доверяете мне – попробуйте сами. Я попробовал.

Срезы массивов

Используя синтаксис диапазонов, легко можно реализовать получение различных *срезов массивов*.

Пример: (исходный файл: `slice_array1.coffee`)

```
myArray = [1..10]
firstThree = myArray[0..2]
console.log  firstThree
```

Пример: (исходный файл: `slice_array1.js`)

```
(function() {
    var firstThree, myArray;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    firstThree = myArray.slice(0, 3);
    console.log(firstThree);
}).call(this);
```

Вывод: (исходный файл: `slice_array1.coffee`)

```
[ 1, 2, 3 ]
```

Тот же пример можно было бы записать, использовав диапазон `0...3` вместо `0..2`:

Пример: (исходный файл: `slice_array2.coffee`)

```
myArray = [1..10]
firstThree = myArray[0...3]
console.log  firstThree
```

Пример: (исходный файл: `slice_array2.js`)

```
(function() {
    var firstThree, myArray;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    firstThree = myArray.slice(0, 3);
    console.log(firstThree);
}).call(this);
```

Вывод: (исходный файл: `slice_array2.coffee`)

```
[ 1, 2, 3 ]
```

Мы не ограничены только началом массива. Мы можем получить любую часть массива, какую пожелаем:

Пример: (исходный файл: `slice_array3.coffee`)

```
myArray = [1..10]
middle = myArray[4..7]
console.log  middle
```

Пример: (исходный файл: `slice_array3.js`)

```
(function() {
    var middle, myArray;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    middle = myArray.slice(4, 8);
    console.log(middle);
}).call(this);
```

Вывод: (исходный файл: `slice_array3.coffee`)

```
[ 5, 6, 7, 8 ]
```

Как видите, у нас получилось извлечь значения из середины массива. Здорово!

Замена значений в массиве

Мы еще не закончили исследование диапазонов и их синтаксиса. Синтаксис диапазонов можно также использовать для *замены группы значений* в массиве.

Пример: (исходный файл: `replace_array.coffee`)

```
myArray = [1..10]
console.log myArray
myArray[4..7] = ['a', 'b', 'c', 'd']
console.log  myArray
```

Пример: (исходный файл: `replace_array.js`)

```
(function() {
    var myArray, _ref;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    console.log(myArray);
    [].splice.apply(myArray, [4, 4].concat(_ref = ['a','b','c','d'])), _ref;
    console.log(myArray);
}).call(this);
```

Вывод: (исходный файл: `replace_array.coffee`)

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 1, 2, 3, 4, 'a', 'b', 'c', 'd', 9, 10 ]
```

Держу пари, такого вы еще не видели! Мощная возможность.

Вставка значений

Иногда возникает необходимость *вставить значения* из одного массива в середину другого массива, в определенную позицию. Для этого можно использовать прием, напоминающий операцию замены значений в группе элементов. Единственное отличие состоит в том, что в качестве последнего значения в диапазоне используется число -1.

Пример: (исходный файл: `injecting_values.coffee`)

```
myArray = [1..10]
console.log myArray
myArray[4..-1] = ['a', 'b', 'c', 'd']
console.log  myArray
```

Пример: (исходный файл: `injecting_values.js`)

```
(function() {
    var myArray, _ref;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    console.log(myArray);
    [].splice.apply(myArray, [4, -1 - 4 + 1].concat(_ref = ['a', 'b', 'c',
↪'d'])), _ref;
    console.log(myArray);
}).call(this);
```

Вывод: (исходный файл: `injecting_values.coffee`)

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 1, 2, 3, 4, 'a', 'b', 'c', 'd', 5, 6, 7, 8, 9, 10 ]
```

Как видите, на место пятого элемента в первом массиве были вставлены значения из второго массива. А оригинальные значения в первом массиве оказались сдвинуты вправо, чтобы освободить место для вставляемых значений.

Объекты/хеши

Объекты в языке JavaScript реализованы очень просто. По сути, они являются хранилищами информации в виде пар ключ/значение. Значениями в этих парах могут быть другие объекты, функции, числа, строки и так далее.

> **Совет.** То, что в JavaScript и CoffeeScript называется объектами, обычно называют хеш-таблицами или просто хешами. Называть их «объектами», как мне кажется, не совсем правильно, потому что в JavaScript существуют другие типы объектов. Я предпочитаю считать их обычными коллекциями пар ключ/значение.

Создание простейшего объекта в языке CoffeeScript выглядит так:

Пример: (исходный файл: `basic_object.coffee`)

```
obj = {}
console.log  obj
```

Пример: (исходный файл: `basic_object.js`)

```
(function() {
    var obj;
    obj = {};
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `basic_object.coffee`)

```
{}
```

Следует признать, что в этом нет ничего особенного. Мы можем приправить этот объект, добавив в него несколько пар ключ/значение:

Пример: (исходный файл: `basic_object2.coffee`)

```
obj =
    firstName: "Mark"
    lastName: "Bates"
console.log  obj
```

Пример: (исходный файл: `basic_object2.js`)

```
(function() {
    var obj;
```

```
    obj = {
        firstName: "Mark",
        lastName: "Bates"
    };
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `basic_object2.coffee`)

```
{ firstName: 'Mark', lastName: 'Bates' }
```

Обратите внимание, что когда пары ключ/значение располагаются в отдельных строках, их не требуется отделять запятой друг от друга. Кроме того, при использовании многострочного синтаксиса можно опустить и фигурные скобки. Тот же самый объект можно было бы определить в одной строке, как показано ниже:

Пример: (исходный файл: `basic_object2_single.coffee`)

```
obj = { firstName: "Mark", lastName: "Bates" }
console.log  obj
```

Пример: (исходный файл: `basic_object2_single.js`)

```
(function() {
    var obj;
    obj = {
        firstName: "Mark",
        lastName: "Bates"
    };
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `basic_object2_single.coffee`)

```
{ firstName: 'Mark', lastName: 'Bates' }
```

При таком подходе получается сэкономить несколько строк кода, но становится обязательным использовать фигурные скобки и запятые, и попутно снижается удобочитаемость.

Если в объект потребуется добавить функцию, сделать это будет очень просто:

Пример: (исходный файл: `basic_object3.coffee`)

```
obj =
    firstName: "Mark"
    lastName: "Bates"
    fullName: ->
        "#{@firstName} #{@lastName}"
console.log  obj
```

Пример: (исходный файл: `basic_object3.js`)

```
(function() {
    var obj;
    obj = {
        firstName: "Mark",
        lastName: "Bates",
        fullName: function() {
            return "" + this.firstName + " " + this.lastName;
        }
    };
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `basic_object3.coffee`)

```
{ firstName: 'Mark', lastName: 'Bates', fullName: [Function] }
```

Когда дело доходит до создания объектов, оказывается, что в рукаве CoffeeScript припрятана еще одна хитрость. Иногда бывает необходимо на основе нескольких переменных сконструировать объект, где в качестве ключей используются имена этих переменных, например:

Пример: (исходный файл: `object_keys1.coffee`)

```
foo = 'FOO'
bar = 'BAR'
obj =
    foo: foo
    bar: bar
console.log  obj
```

Пример: (исходный файл: `object_keys1.js`)

```
(function() {
    var bar, foo, obj;
```

```
    foo = 'FOO';
    bar = 'BAR';
    obj = {
        foo: foo,
        bar: bar
    };
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `object_keys1.coffee`)

```
{ foo: 'FOO', bar: 'BAR' }
```

Не кажется ли вам такой программный код немного избыточным? Да, мне тоже. К счастью с этим согласен и CoffeeScript. Тот же самый объект можно определить немного иначе:

Пример: (исходный файл: `object_keys2.coffee`)

```
foo = 'FOO'
bar = 'BAR'
obj = {
    foo
    bar
}
console.log  obj
```

Пример: (исходный файл: `object_keys2.js`)

```
(function() {
    var bar, foo, obj;
    foo = 'FOO';
    bar = 'BAR';
    obj = {
        foo: foo,
        bar: bar
    };
    console.log(obj);
}).call(this);
```

Вывод: (исходный файл: `object_keys2.coffee`)

```
{ foo: 'FOO', bar: 'BAR' }
```

Ценой такого способа конструировать объекты является обязательное использование фигурных скобок, в противном случае компилятор CoffeeScript не поймет, что вы пытаетесь сделать.

Наконец, когда объект определяется в инструкции вызова функции, фигурные скобки становятся необязательными, независимо от формы объявления, однострочной или многострочной:

Пример: (исходный файл: `objects_into_functions.coffee`)

```
myFunc = (options) ->
    console.log options
myFunc(foo: 'Foo', bar: 'Bar')
```

Пример: (исходный файл: `objects_into_functions.js`)

```
(function() {
    var myFunc;
    myFunc = function(options) {
        return console.log(options);
    };
    myFunc({
        foo: 'Foo',
        bar: 'Bar'
    });
}).call(this);
```

Вывод: (исходный файл: `objects_into_functions.coffee`)

```
{ foo: 'Foo', bar: 'Bar' }
```

На этом, леди и джентльмены, исчерпаны все способы конструирования объектов в языке CoffeeScript.

Получение и изменение атрибутов

При работе с объектами рано или поздно потребуется получить доступ к значениям, хранящимся в этих объектах. Синтаксис доступа к этим значениям в языке CoffeeScript ничем не отличается от синтаксиса в языке JavaScript. Обратиться к атрибуту объекта можно с помощью *точечной нотации* или квадратных скобок `[]`.

Пример: (исходный файл: `object_get_attributes.coffee`)

```
obj =
    firstName: "Mark"
    lastName: "Bates"
```

```
    fullName: ->
        "#{@firstName} #{@lastName}"
console.log obj.firstName
console.log obj['lastName']
console.log  obj.fullName()
```

Пример: (исходный файл: `object_get_attributes.js`)

```
(function() {
    var obj;
    obj = {
        firstName: "Mark",
        lastName: "Bates",
        fullName: function() {
            return "" + this.firstName + " " + this.lastName;
        }
    };
    console.log(obj.firstName);
    console.log(obj['lastName']);
    console.log(obj.fullName());
}).call(this);
```

Вывод: (исходный файл: `object_get_attributes.coffee`)

```
Mark
Bates
Mark  Bates
```

То же относится и к операции присваивания значений атрибутам:
Пример: (исходный файл: `object_set_attributes.coffee`)

```
obj =
    firstName: "Mark"
    lastName: "Bates"
    fullName: ->
        "#{@firstName} #{@lastName}"
obj.firstName = 'MARK'
console.log obj.firstName
obj['lastName'] = 'BATES'
console.log  obj['lastName']
```

Пример: (исходный файл: `object_set_attributes.js`)

```
(function() {
    var obj;
    obj = {
        firstName: "Mark",
        lastName: "Bates",
        fullName: function() {
            return "" + this.firstName + " " + this.lastName;
        }
    };
    obj.firstName = 'MARK';
    console.log(obj.firstName);
    obj['lastName'] = 'BATES';
    console.log(obj['lastName']);
}).call(this);
```

Вывод: (исходный файл: `object_set_attributes.coffee`)

```
MARK
BATES
```

> **Совет.** Популярный инструмент проверки программного кода на JavaScript, JSLint [2], рекомендует всегда использовать точечную нотацию для доступа к атрибутам объекта. Я склонен согласиться с этим. Я считаю, что точечная нотация легче воспринимается и к тому же записывается короче, что всегда хорошо.

Реструктурирующее присваивание

Выше, при обсуждении массивов, мы уже говорили о присваивании значений элементов массива группе переменных. Аналогичную операцию в языке CoffeeScript можно выполнять и с объектами.

Однако синтаксис извлечения значений из объектов не так прост, как в случае с массивами. Он больше напоминает синтаксис определения объекта, где вместо пар ключ/значение перечисляются только ключи.

Сказанное выше проще объяснить на примере:

Пример: (исходный файл: `object_destructuring.coffee`)

```
book =
    title: "Distributed Programming with Ruby"
    author: "Mark Bates"
    chapter_1:
```

```
        name: "Distributed Ruby (DRb)"
        pageCount: 33
    chapter_2:
        name: "Rinda"
        pageCount: 40
{author, chapter_1: {name, pageCount}} = book
console.log "Author: #{author}"
console.log "Chapter 1: #{name}"
console.log "Page Count: #{pageCount}"
```

Пример: (исходный файл: `object_destructuring.js`)

```
(function() {
    var author, book, name, pageCount, _ref;
    book = {
        title: "Distributed Programming with Ruby",
        author: "Mark Bates",
        chapter_1: {
            name: "Distributed Ruby (DRb)",
            pageCount: 33
        },
        chapter_2: {
            name: "Rinda",
            pageCount: 40
        }
    };
    author = book.author, (_ref = book.chapter_1, name = _ref.name,
↪pageCount = _ref.pageCount);
    console.log("Author: " + author);
    console.log("Chapter 1: " + name);
    console.log("Page Count: " + pageCount);
}).call(this);
```

Вывод: (исходный файл: `object_destructuring.coffee`)

```
Author: Mark Bates
Chapter 1: Distributed Ruby (DRb)
Page Count: 33
```

Циклы и итерации

Возможность выполнения итераций по ключам объектов или по элементам массивов в большинстве приложений является насущной

необходимостью. Представьте, что имеется список книг, который нужно вывести на экран, или требуется изменить все значения в объекте. В любом случае, организовать выполнение итераций по элементам коллекций в языке CoffeeScript чрезвычайно просто. Посмотрим, как это делается.

Итерации по элементам массивов

Реализация *итераций по элементам массивов* на языке JavaScript является одним из самых нелюбимых мною занятий. Это грязная работа, при выполнении которой легко допустить ошибку. Язык CoffeeScript реализует простую инструкцию цикла, напоминающую инструкцию `for` в языке Ruby.

Синтаксис цикла `for` очень прост:

```
for <переменная_цикла> in <массив>
```

После инструкции цикла `for` добавляется программный код с отступом, который должен выполняться в каждой итерации, как это делалось в инструкциях `if` и `else`, обсуждавшихся в главе 3, «Управляющие конструкции».

Попробуем обойти символы, хранящиеся в массиве, и вывести их, преобразовав в верхний регистр:

Пример: (исходный файл: `iterating_arrays.coffee`)

```
myLetters = ["a", "b", "c", "d"]
for letter in myLetters
      console.log  letter.toUpperCase()
```

Пример: (исходный файл: `iterating_arrays.js`)

```
(function() {
    var letter, myLetters, _i, _len;
    myLetters = ["a", "b", "c", "d"];
    for (_i = 0, _len = myLetters.length; _i < _len; _i++) {
        letter = myLetters[_i];
        console.log(letter.toUpperCase());
    }
}).call(this);
```

Вывод: (исходный файл: iterating_arrays.coffee)

```
A
B
C
D
```

Ключевое слово by

Представим, что у нас имеется массив, содержащий символы, следующие в алфавитном порядке, и нам необходимо вывести каждый второй символ. Для этого, при определении цикла for, можно воспользоваться ключевым словом by:

Пример: (исходный файл: iterating_arrays_by.coffee)

```
letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
➥"n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
for letter in letters by 2
      console.log  letter
```

Пример: (исходный файл: iterating_arrays_by.js)

```
(function() {
    var letter, letters, _i, _len, _step;
    letters = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
➥"m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"];
    for (_i = 0, _len = letters.length, _step = 2; _i < _len; _i += _step) {
        letter = letters[_i];
        console.log(letter);
    }
}).call(this);
```

Вывод: (исходный файл: iterating_arrays_by.coffee)

```
a
c
e
g
i
k
m
o
q
```

```
s
u
w
y
```

После ключевого слова by можно указать любое число, а цикл for будет выполнять итерации по массиву с указанным шагом.

Ключевое слово when

С помощью ключевого слова when в цикл for можно добавить простое условие.

Допустим, у нас имеется массив с десятью числами, и нам требуется вывести числа меньше 5. Реализовать это можно так:

Пример: (исходный файл: iterating_with_when1.coffee)

```
a = [1..10]
for num in a
    if num < 5
         console.log  num
```

Пример: (исходный файл: iterating_with_when1.js)

```
(function() {
    var a, num, _i, _len;
    a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    for (_i = 0, _len = a.length; _i < _len; _i++) {
        num = a[_i];
        if (num < 5) console.log(num);
    }
}).call(this);
```

Вывод: (исходный файл: iterating_with_when1.coffee)

```
1
2
3
4
```

Тот же самый пример можно записать с использованием ключевого слова when в конце инструкции цикла for:

Пример: (исходный файл: iterating_with_when2.coffee)

```
a = [1..10]
for num in a when num < 5
      console.log  num
```

Пример: (исходный файл: `iterating_with_when2.js`)

```
(function() {
    var a, num, _i, _len;
    a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    for (_i = 0, _len = a.length; _i < _len; _i++) {
        num = a[_i];
        if (num < 5) console.log(num);
    }
}).call(this);
```

Вывод: (исходный файл: `iterating_with_when2.coffee`)

```
1
2
3
4
```

Итерации по атрибутам объектов

Итерации по атрибутам объектов в языке CoffeeScript реализуются почти так же просто, как и итерации по элементам массивов.

Синтаксис инструкции цикла `for` для выполнения итераций по атрибутам объектов выглядит так:

```
for <переменная_для_ключа>, <переменная_для_значения> of <объект>
```

Взгляните на следующий пример:

Пример: (исходный файл: `iterating_objects.coffee`)

```
person =
    firstName: "Mark"
    lastName: "Bates"
for key, value of person
      console.log  "#{key}  is  #{value}"
```

Пример: (исходный файл: `iterating_objects.js`)

```
(function() {
    var key, person, value;
    person = {
```

```
        firstName: "Mark",
        lastName: "Bates"
    };
    for (key in person) {
        value = person[key];
        console.log("" + key + " is " + value);
    }
}).call(this);
```

Вывод: (исходный файл: `iterating_objects.coffee`)

```
firstName is Mark
lastName is Bates
```

В синтаксисе цикла `for` для итераций по элементам массива и атрибутам объекта имеются два существенных отличия. Первое: в инструкции `for`, выполняющей итерации по атрибутам объекта, необходимо объявить две переменные цикла – одну для ключа и одну для значения. Второе отличие: вместо ключевого слова `in`, как в случае итераций по элементам массива, используется ключевое слово `of`.

Ключевое слово by

К сожалению, ключевое слово `by` не может использоваться при выполнении итераций по атрибутам объектов, потому что нет никакого способа перешагивать через пары ключ/значение в объектах, как в случае с элементами массивов.

Ключевое слово when

В отличие от ключевого слова `by`, ключевое слово `when` можно использовать при реализации итераций по атрибутам объектов с помощью инструкции `for`.

Следующий пример выводит пары ключ/значение, где длина значения меньше пяти:

Пример: (исходный файл: `iterating_objects_with_when.coffee`)

```
person =
    firstName: "Mark"
    lastName: "Bates"
for key, value of person when value.length < 5
      console.log "#{key} is #{value}"
```

Пример: (исходный файл: iterating_objects_with_when.js)

```
(function() {
    var key, person, value;
    person = {
        firstName: "Mark",
        lastName: "Bates"
    };
    for (key in person) {
        value = person[key];
        if (value.length < 5) console.log("" + key + " is " + value);
    }
}).call(this);
```

Вывод: (исходный файл: iterating_objects_with_when.coffee)

```
firstName is Mark
```

Ключевое слово own

В языке JavaScript имеется возможность добавлять функции и значения в любые объекты с использованием функции prototype [3]. Именно так и действуют некоторые библиотеки, такие как jQuery, расширяя массивы, строки и другие объекты специальными функциями.

Ниже приводится пример такого расширения объекта:

Пример: (исходный файл: iterating_objects_without_own.coffee)

```
myObject =
    name: "Mark"
for key, value of myObject
    console.log "#{key}: #{value}"
Object.prototype.dob = new Date(1976, 7, 24)
for key, value of myObject
    console.log "#{key}: #{value}"
anotherObject =
    name: "Bob"
for key, value of anotherObject
    console.log "#{key}: #{value}"
```

Пример: (исходный файл: iterating_objects_without_own.js)

```
(function() {
    var anotherObject, key, myObject, value;
```

```
    myObject = {
        name: "Mark"
    };
    for (key in myObject) {
        value = myObject[key];
        console.log("" + key + ": " + value);
    }
    Object.prototype.dob = new Date(1976, 7, 24);
    for (key in myObject) {
        value = myObject[key];
        console.log("" + key + ": " + value);
    }
    anotherObject = {
        name: "Bob"
    };
    for (key in anotherObject) {
        value = anotherObject[key];
        console.log("" + key + ": " + value);
    }
}).call(this);
```

Вывод: (исходный файл: `iterating_objects_without_own.coffee`)

```
name: Mark
name: Mark
dob: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
name: Bob
dob: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
```

Первый цикл, выполняющий обход атрибутов объекта `myObject`, обнаруживает только атрибут `name`, объявленный нами. Однако после добавления атрибута `dob` в прототип `prototype` объекта `Object`, он обнаруживается вторым циклом, выполняющим обход атрибутов объекта `myObject`.

Как быть, если необходимо обнаруживать только те атрибуты, которые явно принадлежат нашему объекту? В языке JavaScript можно было бы использовать функцию `hasOwnProperty`, чтобы выяснить, принадлежит ли атрибут данному объекту или его прототипу – глобальному объекту `Object`. Однако в языке CoffeeScript достаточно просто заменить инструкцию цикла `for` на инструкцию цикла `for own`:

Пример: (исходный файл: `iterating_objects_with_own.coffee`)

```
myObject =
    name: "Mark"
for own key, value of myObject
    console.log "#{key}: #{value}"
Object.prototype.dob = new Date(1976, 7, 24)
for own key, value of myObject
    console.log "#{key}: #{value}"
anotherObject =
    name: "Bob"
for own key, value of anotherObject
    console.log "#{key}: #{value}"
```

Пример: (исходный файл: `iterating_objects_with_own.js`)

```
(function() {
    var anotherObject, key, myObject, value,
        __hasProp = Object.prototype.hasOwnProperty;
    myObject = {
        name: "Mark"
    };
    for (key in myObject) {
        if (!__hasProp.call(myObject, key)) continue;
        value = myObject[key];
        console.log("" + key + ": " + value);
    }
    Object.prototype.dob = new Date(1976, 7, 24);
    for (key in myObject) {
        if (!__hasProp.call(myObject, key)) continue;
        value = myObject[key];
        console.log("" + key + ": " + value);
    }
    anotherObject = {
        name: "Bob"
    };
    for (key in anotherObject) {
        if (!__hasProp.call(anotherObject, key)) continue;
        value = anotherObject[key];
        console.log("" + key + ": " + value);
    }
}).call(this);
```

Вывод: (исходный файл: `iterating_objects_with_own.coffee`)

```
name: Mark
name: Mark
name:  Bob
```

Отлично! Теперь цикл обнаруживает только атрибуты, явно объявленные в объекте `myObject`.

Цикл while

Иногда нам, как разработчикам, бывает необходимо написать блок кода, выполнение которого должно повторяться, пока сохраняется истинность некоторого условия. Это может быть вывод некоторого текста заранее неизвестное число раз, или отображение текста «Пожалуйста, подождите», пока идет загрузка файла. В любом случае, для этих целей в языке CoffeeScript можно использовать цикл `while`.

Напишем функцию, выполняющую блок кода `n` раз:

Пример: (исходный файл: `while_loop.coffee`)

```
times = (number_of_times, callback)->
    index = 0
    while index++ < number_of_times
        callback(index)
    return null
times 5, (index)->
    console.log  index
```

Пример: (исходный файл: `while_loop.js`)

```
(function() {
    var times;
    times = function(number_of_times, callback) {
        var index;
        index = 0;
        while (index++ < number_of_times) {
            callback(index);
        }
        return null;
    };
    times(5, function(index) {
        return console.log(index);
    });
}).call(this);
```

Вывод: (исходный файл: `while_loop.coffee`)

```
1
2
3
4
5
```

В функции `times` используется цикл `while`, который продолжает вызывать функцию обратного вызова, пока значение переменной `index` меньше значения аргумента `number_of_times`.

> **Совет.** В примере, демонстрирующем цикл `while`, можно увидеть выражение `index++`. Для тех, кто не знаком с оператором ++, сообщу, что он увеличивает значение переменной на единицу и возвращает новое, увеличенное значение[1]. По своему действию он эквивалентен инструкции `index = index + 1`.

Цикл until

Как можно заключить из названия[2], цикл `until` является противоположностью циклу `while`. Цикл `while` продолжает выполнять итерации, пока условное выражение остается истинным. Цикл `until`, напротив, продолжает выполнять итерации, пока условное выражение остается ложным.

Пример с циклом `while` можно переписать, задействовав в нем цикл `until`:

Пример: (исходный файл: `until_loop.coffee`)

```
times = (number_of_times, callback)->
    index = 0
    until index++ >= number_of_times
        callback(index)
    return null
times 5, (index)->
    console.log index
```

[1] Здесь автор несколько грешит против истины. Постфиксный оператор инкремента, используемый в данном примере, увеличивает значение переменной на единицу и возвращает прежнее ее значение, а новое, увеличенное значение возвращает префиксный оператор инкремента, который в данном примере выглядел бы так: `++index`.– *Прим. перев.*

[2] Until в данном случае переводится как «пока не». – *Прим. перев.*

Пример: (исходный файл: `until_loop.js`)

```
(function() {
    var times;
    times = function(number_of_times, callback) {
        var index;
        index = 0;
        while (!(index++ >= number_of_times)) {
            callback(index);
        }
        return null;
    };
    times(5, function(index) {
        return console.log(index);
    });
}).call(this);
```

Вывод: (исходный файл: `until_loop.coffee`)

```
1
2
3
4
5
```

> **Совет.** Чтобы не путаться в этих двух циклах, запомните, что: цикл `while` (пока) выполняет итерации, пока (while) условие остается истинным. А цикл `until` (пока не) выполняет итерации, пока условие не станет (until) истинным. Кому-то это может показаться странным, однако есть люди, постоянно путающие эти циклы.

Генераторы

В большинстве примеров итераций использовались очень простые блоки кода, такие как показано ниже:

Пример: (исходный файл: `iterating_arrays.coffee`)

```
myLetters = ["a", "b", "c", "d"]
for letter in myLetters
      console.log  letter.toUpperCase()
```

Поскольку в таких ситуациях повторяемый блок кода состоит из единственной строки, появляется возможность использовать

конструкцию, которая в языке CoffeeScript называется *генератором*. Генераторы по своей сути являются обычными циклами, в которых блок кода находится в той же строке, что и инструкция цикла.

Ниже показан тот же пример, реализованный с помощью генератора:

Пример: (исходный файл: `iterating_arrays_comprehension.coffee`)

```
myLetters = ["a", "b", "c", "d"]
console.log  letter.toUpperCase()  for  letter  in  myLetters
```

Пример: (исходный файл: `iterating_arrays_comprehension.js`)

```
(function() {
    var letter, myLetters, _i, _len;
    myLetters = ["a", "b", "c", "d"];
    for (_i = 0, _len = myLetters.length; _i < _len; _i++) {
        letter = myLetters[_i];
        console.log(letter.toUpperCase());
    }
}).call(this);
```

Вывод: (исходный файл: `iterating_arrays_comprehension.coffee`)

```
A
B
C
D
```

В этом примере мы перенесли блок кода из его отдельной строки и поместили перед инструкцией цикла `for`.

Генераторы можно также использовать для сохранения значений, получаемых в цикле `for`. Воспользуемся тем же примером и сохраним результаты преобразований символов в верхний регистр в новом массиве:

Пример: (исходный файл: `iterating_arrays_comprehension_capture.coffee`)

```
myLetters = ["a", "b", "c", "d"]
upLetters = (letter.toUpperCase() for letter in myLetters)
console.log  upLetters
```

Пример: (исходный файл: `iterating_arrays_comprehension_capture.js`)

```
(function() {
    var letter, myLetters, upLetters;
    myLetters = ["a", "b", "c", "d"];
    upLetters = (function() {
        var _i, _len, _results;
        _results = [];
        for (_i = 0, _len = myLetters.length; _i < _len; _i++) {
            letter = myLetters[_i];
            _results.push(letter.toUpperCase());
        }
        return _results;
    })();
    console.log(upLetters);
}).call(this);
```

Вывод: (исходный файл: `iterating_arrays_comprehension_capture.coffee`)

```
[ 'A', 'B', 'C', 'D' ]
```

Заключая инструкцию генератора в круглые скобки, можно сохранить результаты в другой переменной. Следует отметить, что имеется также возможность сохранять значения, получаемые в многострочной версии цикла `for`:

Пример: (исходный файл: `iterating_arrays_capture.coffee`)

```
myLetters = ["a", "b", "c", "d"]
upLetters = for letter in myLetters
    letter.toUpperCase()
console.log  upLetters
```

Пример: (исходный файл: `iterating_arrays_capture.js`)

```
(function() {
    var letter, myLetters, upLetters;
    myLetters = ["a", "b", "c", "d"];
    upLetters = (function() {
        var _i, _len, _results;
        _results = [];
        for (_i = 0, _len = myLetters.length; _i < _len; _i++) {
            letter = myLetters[_i];
            _results.push(letter.toUpperCase());
        }
```

```
        return _results;
    })();
    console.log(upLetters);
}).call(this);
```

Вывод: (исходный файл: `iterating_arrays_capture.coffee`)

```
[ 'A', 'B', 'C', 'D' ]
```

> **Совет.** Я не являюсь сторонником генераторов, продемонстрированных в этом разделе. Разработчики CoffeeScript широко рекламируют «мощь» генераторов. Я согласен, что они могут быть мощным инструментом, но я также считаю, что программный код с генераторами трудно читать и еще труднее сопровождать. Однако вы можете использовать их, если считаете это целесообразным и необходимым.

Ключевое слово do

Правила видимости переменных в языке JavaScript, обсуждавшиеся в главе 2, «Основы», иногда могут доставлять настоящую головную боль, но самые большие проблемы они могут вызывать в циклах. Из-за асинхронной природы JavaScript может возникнуть ситуация, когда поток выполнения теряет связь с текущим значением переменной, даже при выполнении таких простых операций, как цикл по нескольким числам.

Взгляните на следующий пример. В этом примере необходимо было выполнить цикл по нескольким числам и вывести их. Но перед выводом каждого числа следовало выполнить задержку на одну секунду.

Пример: (исходный файл: `do.coffee`)

```
for x in [1..5]
    setTimeout ->
        console.log x
      , 1
```

Пример: (исходный файл: `do.js`)

```
(function() {
    var x;
    for (x = 1; x <= 5; x++) {
        setTimeout(function() {
```

```
            return console.log(x);
        }, 1);
    }
}).call(this);
```

Вывод: (исходный файл: `do.coffee`)

```
6
6
6
6
6
```

М-да... Это совсем не то, что нам нужно! Мы надеялись вывести числа от 1 до 5. А вместо этого пять раз было выведено число 6. Что же произошло? Дело в том, что цикл потерял связь с текущим значением переменной `x`.

Пока мы ждали секунду, чтобы вывести число, переменная продолжала наращиваться в цикле. А число 6 было выведено потому, что это значение было получено перед последней итерацией в цикле. Оно больше числа 5 и поэтому цикл прекратил итерации.

Так как же предотвратить подобные ситуации и сохранить связь с текущим значением переменной? Сделать это можно с помощью ключевого слова `do`:

Пример: (исходный файл: `do2.coffee`)

```
for x in [1..5]
    do (x) ->
        setTimeout ->
            console.log x
        , 1
```

Пример: (исходный файл: `do2.js`)

```
(function() {
    var x, _fn;
    _fn = function(x) {
        return setTimeout(function() {
            return console.log(x);
        }, 1);
    };
    for (x = 1; x <= 5; x++) {
```

```
        _fn(x);
    }
}).call(this);
```

Вывод: (исходный файл: `do2.coffee`)

```
1
2
3
4
5
```

Ключевое слово `do` создает функцию-обертку вокруг программного кода, реализующего тело цикла, которая принимает и сохраняет значение переменной, имевшееся на момент ее вызова. Очень удобно!

В заключение

Итак, теперь вы знаете все, что требуется знать о коллекциях и итерациях в языке CoffeeScript.

Сначала мы исследовали массивы и их отличительные черты в языке CoffeeScript. Мы познакомились с несколькими хитростями в языке CoffeeScript, упрощающими работу с массивами, такими как проверка вхождения значения в массив, присваивание с перестановкой и, наконец, сохранение элементов массива в нескольких переменных.

Затем мы познакомились с диапазонами. Мы узнали, как с помощью синтаксиса диапазонов создавать массивы с последовательностями чисел. Как с применением диапазонов извлекать фрагменты массивов и даже замещать фрагменты массивов другими значениями.

После диапазонов мы перешли к объектам. Мы рассмотрели различные правила конструирования объектов. Узнали, как получать и изменять значения атрибутов объектов. Как с помощью модифицированного синтаксиса объектов получать значения глубоко вложенных атрибутов и присваивать их переменным.

Затем подошла очередь итераций. Мы увидели, как можно реализовать итерации по элементам массивов и атрибутам объектов. Познакомились с ключевыми словами `by` и `when`, помогающими упростить циклы. Потом мы исследовали работу циклов `while` и `until` и их отличия друг от друга.

Мы познакомились с синтаксисом генераторов, позволяющих записывать в одной строке инструкцию цикла и блок кода.

Наконец, мы узнали, как с помощью ключевого слова `do` не потерять связь с текущим значением переменной во время выполнения итераций по коллекции.

Теперь, вооружившись знаниями о коллекциях, можно перейти к знакомству с другим типом коллекций – классам.

Примечания

2. http://rack.rubyforge.org/.
3. http://www.jslint.com/.
4. http://en.wikipedia.org/wiki/JavaScript#Prototype-based[1].

[1] Описание особенностей наследования на основе прототипов в языке JavaScript можно найти по адресу: http://javascript.ru/tutorial/object/inheritance#nasledovanie-cherez-prototip. – *Прим. перев.*

6. Классы

Классы [1] по своей сути являются шаблонами для создания новых экземпляров объектов с предопределенными функциями и переменными. Эти экземпляры могут сохранять информацию о состоянии, соответствующем отдельным объектам. За годы своего существования язык JavaScript не раз подвергался критике за отсутствие поддержки обычных классов.

В главе 5, «Коллекции и итерации», мы познакомились с объектами, поддерживаемыми в языке JavaScript. В каждом примере мы создавали совершенно новый объект и придавали ему некоторый набор функций и атрибутов. Этого вполне достаточно для работы с простыми объектами, но как быть, если потребуется организовать более сложную модель представления данных? Более того, что делать, если необходимо будет реализовать несколько таких сложных моделей представления данных? Обычно в подобных ситуациях нам на выручку приходят классы.

К счастью язык CoffeeScript обладает полноценной поддержкой классов. Но, если в JavaScript отсутствует поддержка обычных классов, как тогда CoffeeScript решает эту проблему? Если говорить коротко – посредством некоторых разумных ограничений и использования функций объектов. Более развернутый ответ вы найдете в этой главе.

Определение классов

Определение классов в языке CoffeeScript может выглядеть просто и незамысловато, как в следующей строке:

Пример: (исходный файл: `simple_class1.coffee`)

```
class Employee
```

Пример: (исходный файл: `simple_class1.js`)

```
(function() {
    var Employee;
```

```
    Employee = (function() {
        function Employee() {}
        return Employee;
    })();
}).call(this);
```

Эта строка определяет простой класс с именем `Employee`. Имея такое определение, можно создавать новые экземпляры этого класса, как показано ниже:

Пример: (исходный файл: `simple_class2.coffee`)

```
class Employee
emp1 = new Employee()
emp2 = new Employee()
```

Пример: (исходный файл: `simple_class2.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee() {}
        return Employee;
    })();
    emp1 = new Employee();
    emp2 = new Employee();
}).call(this);
```

Мы добавляем ключевое слово `new` перед именем класса и получаем совершенно новый экземпляр этого объекта, с которым можем делать все, что заблагорассудится.

> **Совет.** При создании новых экземпляров объектов с помощью ключевого слова `new` необязательно добавлять круглые скобки после имени класса, как это делается в наших примерах, но мне кажется, что они делают программный код более понятным и удобочитаемым. Вы же можете поступать, как вам кажется правильнее.

Определение функций

При *определении функций* в классах используются те же правила и синтаксис, которые применяются при определении функций в простых объектах, потому что в действительности делается именно это.

Пример: (исходный файл: simple_class_with_function.coffee)

```
class Employee
    dob: (year = 1976, month = 7, day = 24)->
        new Date(year, month, day)
emp1 = new Employee()
console.log emp1.dob()
emp2 = new Employee()
console.log  emp2.dob(1979,  3,  28)
```

Пример: (исходный файл: simple_class_with_function.js)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee() {}
        Employee.prototype.dob = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            return new Date(year, month, day);
        };
        return Employee;
    })();
    emp1 = new Employee();
    console.log(emp1.dob());
    emp2 = new Employee();
    console.log(emp2.dob(1979, 3, 28));
}).call(this);
```

Вывод: (исходный файл: simple_class_with_function.coffee)

```
Tue, 24 Aug 1976 04:00:00 GMT
Sat,  28  Apr  1979  05:00:00  GMT
```

Функция constructor

В языке CoffeeScript имеется возможность определить функцию с именем constructor, которая вызывается при создании нового экземпляра объекта. Функция constructor практически не отличается от любых других функций, которые мы рассматривали. Единственная ее особенность заключается в том, что она вызывается в момент создания нового экземпляра автоматически, без нашего участия.

Совет. Я немного согрешил против истины, когда заявил, что в момент создания нового экземпляра объекта функция `constructor` вызывается автоматически. В действительности мы вызываем ее явно, когда создаем новый экземпляр инструкцией, такой как `new Employee()`. Просто она вызывается под другим именем.

Пример: (исходный файл: `simple_class3.coffee`)

```
class Employee
    constructor: ->
        console.log "Instantiated a new Employee object"
    dob: (year = 1976, month = 7, day = 24)->
        new Date(year, month, day)
emp1 = new Employee()
console.log emp1.dob()
emp2 = new Employee()
console.log  emp2.dob(1979,  3,  28)
```

Пример: (исходный файл: `simple_class3.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee() {
            console.log("Instantiated a new Employee object");
        }
        Employee.prototype.dob = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            return new Date(year, month, day);
        };
        return Employee;
    })();
    emp1 = new Employee();
    console.log(emp1.dob());
    emp2 = new Employee();
    console.log(emp2.dob(1979, 3, 28));
}).call(this);
```

Вывод: (исходный файл: `simple_class3.coffee`)

```
Instantiated a new Employee object
Tue, 24 Aug 1976 04:00:00 GMT
```

```
Instantiated a new Employee object
Sat, 28 Apr 1979 05:00:00 GMT
```

Как видите, каждый раз, когда создается новый объект `Employee`, он выводит в консоль сообщение, чтобы известить о своем создании.

Как будет показано далее в этой главе, функция `constructor` обеспечивает простой способ быстро инициализировать новый объект его индивидуальными данными.

Область видимости в классах

Основу классов в языке CoffeeScript составляют обычные объекты, наполненные большим объемом шаблонного программного кода на JavaScript, обеспечивающего их функциональность. Так как классы – это обычные объекты, просто прикрытые красочной оберткой, к ним применяются те же *правила видимости переменных*, атрибутов и функций, что и к обычным объектам.

Вернемся к нашему классу `Employee`[1]. Работники в реальной жизни имеют имена, поэтому добавим соответствующий атрибут в наш класс `Employee`. Было бы желательно, чтобы имя передавалось при создании нового экземпляра класса `Employee` и присваивалось атрибуту, принадлежащему новому экземпляру класса `Employee`.

Пример: (исходный файл: `class_scope.coffee`)

```
class Employee
    constructor: (name)->
        @name = name
    dob: (year = 1976, month = 7, day = 24)->
        new Date(year, month, day)
emp1 = new Employee("Mark")
console.log emp1.name
console.log emp1.dob()
emp2 = new Employee("Rachel")
console.log emp2.name
console.log  emp2.dob(1979,  3,  28)
```

Пример: (исходный файл: `class_scope.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
```

[1] В переводе на русский язык – работник. – *Прим. перев.*

```
        function Employee(name) {
            this.name = name;
        }
        Employee.prototype.dob = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            return new Date(year, month, day);
        };
        return Employee;
    })();
    emp1 = new Employee("Mark");
    console.log(emp1.name);
    console.log(emp1.dob());
    emp2 = new Employee("Rachel");
    console.log(emp2.name);
    console.log(emp2.dob(1979, 3, 28));
}).call(this);
```

Вывод: (исходный файл: `class_scope.coffee`)

```
Mark
Tue, 24 Aug 1976 04:00:00 GMT
Rachel
Sat, 28 Apr 1979 05:00:00 GMT
```

Как уже говорилось выше, функция `constructor` ничем не отличается от любых других функций, поэтому для нее действуют те же правила видимости и определения. Зная, что теперь в аргументе `name` передается имя, мы присваиваем значение этого аргумента атрибуту экземпляра объекта с помощью инструкции `@name = name`. Напомню: в главе 3, «Управляющие конструкции», говорилось, что символ `@` является псевдонимом ключевого слова `this`.

Определив значение атрибута в экземпляре объекта, к нему можно обращаться как к любому другому атрибуту.

Существует более простой способ достижения той же цели, с помощью одной из моих самых любимых особенностей языка CoffeeScript (которую мне хотелось бы иметь в других языках). Мы можем сократить определение функции `constructor`, как показано ниже:

Пример: (исходный файл: `class_scope1.coffee`)

```
class Employee
    constructor: (@name)->
```

```
    dob: (year = 1976, month = 7, day = 24)->
        new Date(year, month, day)
emp1 = new Employee("Mark")
console.log emp1.name
console.log emp1.dob()
emp2 = new Employee("Rachel")
console.log emp2.name
console.log  emp2.dob(1979,  3,  28)
```

Пример: (исходный файл: `class_scope1.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee(name) {
            this.name = name;
        }
        Employee.prototype.dob = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            return new Date(year, month, day);
        };
        return Employee;
    })();
    emp1 = new Employee("Mark");
    console.log(emp1.name);
    console.log(emp1.dob());
    emp2 = new Employee("Rachel");
    console.log(emp2.name);
    console.log(emp2.dob(1979, 3, 28));
}).call(this);
```

Вывод: (исходный файл: `class_scope1.coffee`)

```
Mark
Tue, 24 Aug 1976 04:00:00 GMT
Rachel
Sat,  28  Apr  1979  05:00:00  GMT
```

Добавив оператор `@` перед именем аргумента `name` в определении функции `constructor`, мы сообщили компилятору CoffeeScript, что он должен сгенерировать программный код на JavaScript, который

присвоит значение этого аргумента атрибуту с тем же именем, в данном случае – атрибуту с именем `name`.

Также мы легко можем обращаться к этому атрибуту в других функциях класса. Вернемся к нашему примеру и добавим в него метод вывода имени работника и даты его рождения:

Пример: (исходный файл: `class_scope2.coffee`)

```
class Employee
    constructor: (@name)->
    dob: (year = 1976, month = 7, day = 24)->
        new Date(year, month, day)
    printInfo: (year = 1976, month = 7, day = 24)->
        console.log "Name: #{@name}"
        console.log "DOB: #{@dob(year, month, day)}"
emp1 = new Employee("Mark")
emp1.printInfo(1976, 7, 24)
emp2 = new Employee("Rachel")
emp2.printInfo(1979,  3,  28)
```

Пример: (исходный файл: `class_scope2.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee(name) {
            this.name = name;
        }
        Employee.prototype.dob = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            return new Date(year, month, day);
        };
        Employee.prototype.printInfo = function(year, month, day) {
            if (year == null) year = 1976;
            if (month == null) month = 7;
            if (day == null) day = 24;
            console.log("Name: " + this.name);
            return console.log("DOB: " + (this.dob(year, month, day)));
        };
        return Employee;
    })();
    emp1 = new Employee("Mark");
```

```
    emp1.printInfo(1976, 7, 24);
    emp2 = new Employee("Rachel");
    emp2.printInfo(1979, 3, 28);
}).call(this);
```

Вывод: (исходный файл: `class_scope2.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
```

Заканчивать этот раздел, оставляя программный код в таком виде, было бы неправильным. Я не хочу постоянно передавать год, месяц и день, когда мне потребуется вывести информацию о работнике. Я хочу передать дату рождения функции `constructor` и затем извлекать ее, когда это будет нужно. Поэтому добавим такую возможность:

Пример: (исходный файл: `class_scope_refactor_1.coffee`)

```
class Employee
    constructor: (@name, @dob)->
    printInfo: ->
        console.log "Name: #{@name}"
        console.log "DOB: #{@dob}"
emp1 = new Employee("Mark", new Date(1976, 7, 24))
emp1.printInfo()
emp2 = new Employee("Rachel", new Date(1979, 3, 28))
emp2.printInfo()
```

Пример: (исходный файл: `class_scope_refactor_1.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee(name, dob) {
            this.name = name;
            this.dob = dob;
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.name);
            return console.log("DOB: " + this.dob);
        };
        return Employee;
```

```
    })();
    emp1 = new Employee("Mark", new Date(1976, 7, 24));
    emp1.printInfo();
    emp2 = new Employee("Rachel", new Date(1979, 3, 28));
    emp2.printInfo();
}).call(this);
```

Вывод: (исходный файл: `class_scope_refactor_1.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
```

Такой код определенно выглядит чище и «суше» (принцип программирования DRY, Don't Repeat Yourself – не повторяйся)[1]. [2] Когда я вижу в определении функции два аргумента, я начинаю задаваться вопросом – не будет ли она принимать большее число аргументов? Если ответ «да», я пересматриваю определение моей функции. В данном случае меня волнует объявление функции `constructor`. Как быть, если позднее ей потребуется передать и другие сведения, такие как размер заработной платы, название отдела, имя руководителя и т.д.? Попробуем реорганизовать программный код:

Пример: (исходный файл: `class_scope_refactor_2.coffee`)

```
class Employee
    constructor: (@attributes)->
    printInfo: ->
        console.log "Name: #{@attributes.name}"
        console.log "DOB: #{@attributes.dob}"
        if @attributes.salary
            console.log "Salary: #{@attributes.salary}"
        else
            console.log "Salary: Unknown"
emp1 = new Employee
    name: "Mark"
    dob: new Date(1976, 7, 24)
    salary: "$1.00"
emp1.printInfo()
emp2 = new Employee
    name: "Rachel"
```

[1] Здесь игра слов: DRY переводится, как «сухой». – *Прим. перев.*

```
    dob: new Date(1979, 3, 28)
emp2.printInfo()
```

Пример: (исходный файл: `class_scope_refactor_2.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee(attributes) {
            this.attributes = attributes;
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.attributes.name);
            console.log("DOB: " + this.attributes.dob);
            if (this.attributes.salary) {
                return console.log("Salary: " + this.attributes.salary);
            } else {
                return console.log("Salary: Unknown");
            }
        };
        return Employee;
    })();
    emp1 = new Employee({
        name: "Mark",
        dob: new Date(1976, 7, 24),
        salary: "$1.00"
    });
    emp1.printInfo();
    emp2 = new Employee({
        name: "Rachel",
        dob: new Date(1979, 3, 28)
    });
    emp2.printInfo();
}).call(this);
```

Вывод: (исходный файл: `class_scope_refactor_2.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
Salary: $1.00
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
Salary:  Unknown
```

Теперь при создании экземпляра класса `Employee` можно передать любое количество аргументов и все они будут доступны через атрибут `attributes` объекта. Так намного лучше и проще для дальнейшего расширения.

Как видно в примере, при создании первого экземпляра `Employee` был передан третий атрибут, `salary`, опущенный при создании второго экземпляра. Теперь можно передавать хоть сотни разных атрибутов, и это не потребует вносить изменения в программный код.

Прежде чем покинуть этот раздел, решим еще одну проблему. Некоторым из вас могла прийти в голову замечательная, на первый взгляд, идея: что если воспользоваться знаниями, полученными в главе Chapter 5, «Коллекции и итерации», реализовать обход содержимого аргумента `attributes` и организовать добавление каждой пары ключ/значение непосредственно в объект, чтобы впоследствии не приходилось постоянно вызывать `@attributes` в своем программном коде?

Давайте посмотрим, как это можно сделать, и вы убедитесь, что это не самая удачная мысль:

Пример: (исходный файл: `class_scope_refactor_3.coffee`)

```
class Employee
    constructor: (@attributes)->
        for key, value of @attributes
            @[key] = value
    printInfo: ->
        console.log "Name: #{@name}"
        console.log "DOB: #{@dob}"
        if @salary
            console.log "Salary: #{@salary}"
        else
            console.log "Salary: Unknown"
emp1 = new Employee
    name: "Mark"
    dob: new Date(1976, 7, 24)
    salary: "$1.00"
emp1.printInfo()
emp2 = new Employee
    name: "Rachel",
    dob: new Date(1979, 3, 28)
    printInfo: ->
        console.log "I've hacked your code!"
emp2.printInfo()
```

Пример: (исходный файл: `class_scope_refactor_3.js`)

```
(function() {
    var Employee, emp1, emp2;
    Employee = (function() {
        function Employee(attributes) {
            var key, value, _ref;
            this.attributes = attributes;
            _ref = this.attributes;
            for (key in _ref) {
                value = _ref[key];
                this[key] = value;
            }
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.name);
            console.log("DOB: " + this.dob);
            if (this.salary) {
                return console.log("Salary: " + this.salary);
            } else {
                return console.log("Salary: Unknown");
            }
        };
        return Employee;
    })();
    emp1 = new Employee({
        name: "Mark",
        dob: new Date(1976, 7, 24),
        salary: "$1.00"
    });
    emp1.printInfo();
        emp2 = new Employee({
        name: "Rachel",
        dob: new Date(1979, 3, 28),
        printInfo: function() {
            return console.log("I've hacked your code!");
        }
    });
    emp2.printInfo();
}).call(this);
```

Вывод: (исходный файл: `class_scope_refactor_3.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
```

```
Salary: $1.00
I've hacked your code!
```

Ого! Как легко мы смогли переопределить функцию `printInfo`, передав ее в списке аргументов при создании второго экземпляра. Да, такой код определенно легче читается, но он также легко взламывается, а кому этого хочется? Программный код на JavaScript и так достаточно просто изменить, так зачем еще больше упрощать эту возможность? Закончим на этом и пойдем дальше, притворившись, что мы не вносили последние изменения.

Наследование классов

При создании объектно-ориентированных программ разработчики часто сталкиваются с необходимостью *наследования классов*. [3] Наследование позволяет взять за основу некоторый класс, такой как `Employee`, и создать измененную версию этого класса.

На производстве все являются работниками, но не все руководителями. Поэтому определим класс `Manager`, наследующий, или расширяющий, класс `Employee`:

Пример: (исходный файл: `manager1.coffee`)

```
class Employee
    constructor: (@attributes)->
    printInfo: ->
        console.log "Name: #{@attributes.name}"
        console.log "DOB: #{@attributes.dob}"
        console.log "Salary: #{@attributes.salary}"
class Manager extends Employee
employee = new Employee
    name: "Mark"
    dob: new Date(1976, 7, 24)
    salary: 50000
employee.printInfo()
manager = new Manager
    name: "Rachel"
    dob: new Date(1979, 3, 28)
    salary: 100000
manager.printInfo()
```

Пример: (исходный файл: `manager1.js`)

```
(function() {
    var Employee, Manager, employee, manager,
        __hasProp = Object.prototype.hasOwnProperty,
        __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function ctor() {
this.constructor = child; } ctor.prototype = parent.prototype;
child.prototype = new ctor; child.__super__ = parent.prototype; return
child; };
    Employee = (function() {
        function Employee(attributes) {
            this.attributes = attributes;
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.attributes.name);
            console.log("DOB: " + this.attributes.dob);
            return console.log("Salary: " + this.attributes.salary);
        };
        return Employee;
    })();
    Manager = (function(_super) {
        __extends(Manager, _super);
        function Manager() {
            Manager.__super__.constructor.apply(this, arguments);
        }
        return Manager;
    })(Employee);
    employee = new Employee({
        name: "Mark",
        dob: new Date(1976, 7, 24),
        salary: 50000
    });
    employee.printInfo();
        manager = new Manager({
        name: "Rachel",
        dob: new Date(1979, 3, 28),
        salary: 100000
    });
    manager.printInfo();
}).call(this);
```

Вывод: (исходный файл: `manager1.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
```

```
Salary: 50000
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
Salary:  100000
```

Совет. В реальном мире нам, вероятно, пришлось бы определять роли для разных категорий работников, но для целей нашего обсуждения примем, что такой подход является лучшим решением проблемы разделения работников на категории.

Определив класс `Manager` как `class Manager`, мы получили бы простейший класс, но использовав ключевое слово `extends` и указав имя наследуемого класса `Employee`, мы разом передали классу `Manager` все функциональные возможности класса `Employee`.

Сделаем еще шаг и посмотрим, как переопределять методы в подклассах. Добавим в класс `Employee` функцию `bonus`, возвращающую число `0`. Обычные работники очевидно не получают надбавки. Однако руководители получают 10% надбавку, поэтому нам необходимо гарантировать, что при вызове функции `bonus` для объекта, представляющего руководителя, мы получим правильное значение.

Пример: (исходный файл: `manager2.coffee`)

```
class Employee
    constructor: (@attributes)->
    printInfo: ->
        console.log "Name: #{@attributes.name}"
        console.log "DOB: #{@attributes.dob}"
        console.log "Salary: #{@attributes.salary}"
        console.log "Bonus: #{@bonus()}"
    bonus: ->
        0
class Manager extends Employee
    bonus: ->
        @attributes.salary * .10
employee = new Employee
    name: "Mark"
    dob: new Date(1976, 7, 24)
    salary: 50000
employee.printInfo()
manager = new Manager
    name: "Rachel"
    dob: new Date(1979, 3, 28)
```

```
    salary: 100000
manager.printInfo()
```

Пример: (исходный файл: `manager2.js`)

```
(function() {
    var Employee, Manager, employee, manager,
        __hasProp = Object.prototype.hasOwnProperty,
        __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function
ctor() { this.constructor = child; } ctor.prototype = parent.prototype;
child.prototype = new ctor; child.__super__ = parent.prototype; return child; };
    Employee = (function() {
        function Employee(attributes) {
            this.attributes = attributes;
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.attributes.name);
            console.log("DOB: " + this.attributes.dob);
            console.log("Salary: " + this.attributes.salary);
            return console.log("Bonus: " + (this.bonus()));
        };
        Employee.prototype.bonus = function() {
            return 0;
        };
        return Employee;
    })();
    Manager = (function(_super) {
        __extends(Manager, _super);
        function Manager() {
            Manager.__super__.constructor.apply(this, arguments);
        }
        Manager.prototype.bonus = function() {
            return this.attributes.salary * .10;
        };
        return Manager;
    })(Employee);
    employee = new Employee({
        name: "Mark",
        dob: new Date(1976, 7, 24),
        salary: 50000
    });
    employee.printInfo();
    manager = new Manager({
```

```
        name: "Rachel",
        dob: new Date(1979, 3, 28),
        salary: 100000
    });
    manager.printInfo();
}).call(this);
```

Вывод: (исходный файл: `manager2.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
Salary: 50000
Bonus: 0
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
Salary: 100000
Bonus: 10000
```

Переопределение функций в подклассах выполняется как повторное определение функции с новой реализацией. Но как быть, если желательно вызвать оригинальную функцию и выполнить несколько дополнительных операций. Пример такого вызова следует ниже.

Работники будут искренне разочарованы, когда увидят в выводе функции `printInfo`, что у них размер надбавки равен нулю, поэтому мы сейчас полностью исключим вывод этой информации для работников, но оставим ее для руководителей. Сделать это можно с помощью ключевого слова `super`:

Пример: (исходный файл: `manager3.coffee`)

```
class Employee
    constructor: (@attributes)->
    printInfo: ->
        console.log "Name: #{@attributes.name}"
        console.log "DOB: #{@attributes.dob}"
        console.log "Salary: #{@attributes.salary}"
        bonus: ->
            0
class Manager extends Employee
    bonus: ->
        @attributes.salary * .10
    printInfo: ->
        super
        console.log "Bonus: #{@bonus()}"
```

```
employee = new Employee
    name: "Mark"
    dob: new Date(1976, 7, 24)
    salary: 50000
employee.printInfo()
manager = new Manager
    name: "Rachel"
    dob: new Date(1979, 3, 28)
    salary: 100000
manager.printInfo()
```

Пример: (исходный файл: `manager3.js`)

```
(function() {
    var Employee, Manager, employee, manager,
        __hasProp = Object.prototype.hasOwnProperty,
        __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function ctor() {
this.constructor = child; } ctor.prototype = parent.prototype; child.prototype =
new ctor; child.__super__ = parent.prototype; return child; };
    Employee = (function() {

        function Employee(attributes) {
            this.attributes = attributes;
        }
        Employee.prototype.printInfo = function() {
            console.log("Name: " + this.attributes.name);
            console.log("DOB: " + this.attributes.dob);
            return console.log("Salary: " + this.attributes.salary);
        };
        Employee.prototype.bonus = function() {
            return 0;
        };
        return Employee;
    })();
    Manager = (function(_super) {
        __extends(Manager, _super);
        function Manager() {
            Manager.__super__.constructor.apply(this, arguments);
        }
        Manager.prototype.bonus = function() {
            return this.attributes.salary * .10;
        };
        Manager.prototype.printInfo = function() {
            Manager.__super__.printInfo.apply(this, arguments);
```

```
            return console.log("Bonus: " + (this.bonus()));
        };
        return Manager;
    })(Employee);
    employee = new Employee({
        name: "Mark",
        dob: new Date(1976, 7, 24),
        salary: 50000
    });
    employee.printInfo();
        manager = new Manager({
        name: "Rachel",
        dob: new Date(1979, 3, 28),
        salary: 100000
    });
    manager.printInfo();
}).call(this);
```

Вывод: (исходный файл: `manager3.coffee`)

```
Name: Mark
DOB: Tue Aug 24 1976 00:00:00 GMT-0400 (EDT)
Salary: 50000
Name: Rachel
DOB: Sat Apr 28 1979 00:00:00 GMT-0500 (EST)
Salary: 100000
Bonus:  10000
```

В функции `printInfo`, объявленной в классе `Manager`, сначала выполняется обращение к функции `super`. При этом вызывается оригинальная функция `printInfo` из класса `Employee` и ей передаются все аргументы, которые были переданы вызывающей функции. Затем выводится информация о размере надбавки для руководителя.

Совет. Вызов `super` можно выполнить в любой точке функции, переопределяющей ее – он не обязательно должен быть первым (как в некоторых языках). Более того, метод `super` вообще может не вызываться.

Совет. Вызывая функцию `super`, нет необходимости передавать ей аргументы явно. По умолчанию, все аргументы, полученные переопределяющей функцией, автоматически будут переданы функции `super`. Однако существует возможность явно передавать функции `super` любые аргументы. Это может пригодиться, когда потребуется дополнить или изменить аргументы перед тем, как они попадут в метод `super`.

Функции класса

Функции класса не требуют для их вызова наличия экземпляра класса. Эти функции могут быть невероятно полезными. Одним из применений такой возможности является организация своеобразных пространств имен для своих функций. Примером таких функций в языке JavaScript может служить `Math.random()`. Чтобы получить случайное число не нужно создавать новый объект `Math`. Но, благодаря включению функции `random` в класс `Math`, устраняется риск, что какая-то другая, одноименная функция затрет ее.

> **Совет.** В действительности `Math` не является классом – это обычный объект, играющий роль пространства имен. Иногда я немножко подправляю истину, чтобы лучше объяснить свою точку зрения.

Функции класса могут также использоваться для выполнения операций, затрагивающих множество экземпляров класса, таких как поиск их в базе данных.

Мы напишем пару функций класса для нашего класса `Employee`. Поскольку в нашем распоряжении нет полноценной базы данных, мы просто будем следить за созданными экземплярами класса `Employee` и выводить их количество. Следить за созданными экземплярами будет функция класса `hire`, которая будет получать их и добавлять в массив, выступающий в роли импровизированной базы данных. Также будет создана функция класса, возвращающая общее количество экземпляров в нашей базе данных.

Пример: (исходный файл: `class_level.coffee`)

```
class Employee
    constructor: ->
        Employee.hire(@)
    @hire: (employee) ->
        @allEmployees ||= []
        @allEmployees.push employee
    @total: ->
        console.log "There are #{@allEmployees.length} employees."
        @allEmployees.length
new Employee()
new Employee()
new Employee()
Employee.total()
```

Пример: (исходный файл: `class_level.js`)

```
(function() {
    var Employee;
    Employee = (function() {
        function Employee() {
            Employee.hire(this);
        }
        Employee.hire = function(employee) {
            this.allEmployees || (this.allEmployees = []);
            return this.allEmployees.push(employee);
        };
        Employee.total = function() {
            console.log("There are " + this.allEmployees.length +
                        " employees.");
            return this.allEmployees.length;
        };
        return Employee;
    })();
    new Employee();
    new Employee();
    new Employee();
    Employee.total();
}).call(this);
```

Вывод: (исходный файл: `class_level.coffee`)

```
There are 3 employees.
```

Как же создаются функции класса? Наличие символа `@` перед именем функции сообщает компилятору CoffeeScript, что определяется функция класса. В JavaScript, когда символ `@` окажется замещен ключевым словом `this.`, ссылка `this` будет ссылаться на класс `Employee`, а не на экземпляр этого класса.

Внутри функций класса допускается использовать только другие функции и атрибуты класса.

> **Совет.** Обычно в своих приложениях я создаю определения классов, которые являются лишь связкой методов класса. Такой прием отлично подходит для организации вспомогательных пакетов и обеспечивает организацию функций в непересекающиеся пространства имен. Если позднее мне это потребуется, я смогу создать от них дочерние классы и переопределить те или иные функции для своих нужд.

Однако при работе с функциями и атрибутами класса ключевое слово `super` имеет ограниченное применение. Его можно использовать для вызова оригинальной функции, переопределенной в дочернем классе. Но, если оригинальная функция, вызываемая с помощью ключевого слова `super`, ссылается на какие-либо атрибуты класса, это приведет к появлению большой и толстой ошибки во время выполнения.

Посмотрим, что произойдет, если переопределить функцию класса `total` в классе `Manager` и вызвать в ней функцию `super`:

Пример: (исходный файл: `class_level_super.coffee`)

```
class Employee
    constructor: ->
        Employee.hire(@)
    @hire: (employee) ->
        @allEmployees ||= []
        @allEmployees.push employee
    @total: ->
        console.log "There are #{@allEmployees.length} employees."
        @allEmployees.length
class Manager extends Employee
    @total: ->
        console.log "There are 0 managers."
        super
new Employee()
new Employee()
new Employee()
Manager.total()
```

Пример: (исходный файл: `class_level_super.js`)

```
(function() {
    var Employee, Manager,
        __hasProp = Object.prototype.hasOwnProperty,
        __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function ctor() {
this.constructor = child; } ctor.prototype = parent.prototype;
child.prototype = new ctor; child.__super__ = parent.prototype;
return child; };
    Employee = (function() {
        function Employee() {
            Employee.hire(this);
        }
```

```
        Employee.hire = function(employee) {
            this.allEmployees || (this.allEmployees = []);
            return this.allEmployees.push(employee);
        };
        Employee.total = function() {
            console.log("There are " + this.allEmployees.length +
                        " employees.");
            return this.allEmployees.length;
        };
        return Employee;
    })();
    Manager = (function(_super) {
        __extends(Manager, _super);
        function Manager() {
            Manager.__super__.constructor.apply(this, arguments);
        }
        Manager.total = function() {
            console.log("There are 0 managers.");
            return Manager.__super__.constructor.total.apply(this,
                                                             arguments);
        };
        return Manager;
    })(Employee);
    new Employee();
    new Employee();
    new Employee();
    Manager.total();
}).call(this);
```

Вывод: (исходный файл: `class_level_super.coffee`)

```
There are 0 managers.
TypeError: Cannot read property 'length' of undefined
    at Function.<anonymous> (.../classes/class_level_super.coffee:18:51)
    at Function.total (.../classes/class_level_super.coffee:36:50)
    at Object.<anonymous> (.../classes/class_level_super.coffee:49:11)
    at Object.<anonymous> (.../classes/class_level_super.coffee:51:4)
    at Module._compile (module.js:432:26)
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/coffee-script.js:68:25)
    at /usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/command.js:135:29
    at /usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/command.js:110:18
```

```
    at [object Object].<anonymous> (fs.js:114:5)
      at [object Object].emit (events.js:64:17)
```

Как видите, при попытке получить значение свойства `length` атрибута-массива `@allEmployees` была получена ошибка:

```
TypeError: Cannot read property 'length' of undefined
```

Причина такого поведения проста, но может потребоваться некоторое время, чтобы понять ее. В языке JavaScript отсутствует истинное наследование, поэтому компилятору CoffeeScript приходится использовать разные ухищрения, чтобы создать иллюзию поддержки классов и наследования. Дело в том, что подкласс `Manager` – это совершенно другой объект, отличный от класса `Employee`, а атрибуты класса определяются в классе `Employee`, а не в классе `Manager`, поэтому класс `Manager` не может обращаться к ним. Я рассказал вам об этом, чтобы хоть немного прояснить ситуацию.

> **Совет.** Я считаю, что лучше избегать использования ключевого слова `super` в функциях класса. Кроме того, я стараюсь делать все свои функции класса самодостаточными и автономными, чтобы не сталкиваться с подобными проблемами.

Функции прототипа

В JavaScript имеется возможность добавлять функции и атрибуты ко всем экземплярам объекта, включая их в прототип объекта – атрибут с именем `prototype`.

В CoffeeScript то же самое можно сделать с помощью оператора `::`. Воспользуемся им и добавим функцию `size` во все экземпляры массивов. Функция `size` будет возвращать значение свойства `length` массива.

Пример: (исходный файл: `prototypes.coffee`)

```
myArray = [1..10]
try
    console.log myArray.size()
catch error
    console.log error
Array::size = -> @length
console.log myArray.size()
myArray.push(11)
console.log  myArray.size()
```

Пример: (исходный файл: prototypes.js)

```
(function() {
    var myArray;
    myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    try {
        console.log(myArray.size());
    } catch (error) {
        console.log(error);
    }
    Array.prototype.size = function() {
        return this.length;
    };
    console.log(myArray.size());
    myArray.push(11);
    console.log(myArray.size());
}).call(this);
```

Вывод: (исходный файл: prototypes.coffee)

```
[TypeError: Object 1,2,3,4,5,6,7,8,9,10 has no method 'size']
10
11
```

Первая попытка вызвать функцию size массива вызвала ошибку, потому что в этот момент данная функция еще не существует. Затем мы добавили функцию size в прототип класса Array, после чего в следующих двух попытках массив стал вести себя, как мы и ожидали.

> **Совет.** Оператор :: был введен исключительно для удобства. Он не препятствует использованию атрибута prototype непосредственно, но я считаю, что необходимость ввода лишних символов не стоит того.

Привязка (-> и =>)

JavaScript – это язык программирования для разработки *асинхронных* [4], или управляемых событиями программ. Когда в *синхронной* программе вызывается некоторая функция, выполнение остальной части программы приостанавливается, пока упомянутая функция не вернет управление. В JavaScript это не всегда так. В общем случае выполнение программы может быть продолжено и после вызова

функции, до ее завершения. Если это поможет, такой стиль программирования можно описать словами «запустил и забыл» – программа запускает функцию и забывает о ней. Взгляните на следующий пример, демонстрирующий работу асинхронной программы.

Пример: (исходный файл: `fire_and_forget.coffee`)

```
fire = (msg, wait)->
    setTimeout ->
        console.log msg
    , wait
fire("Hello", 3000)
fire("World",  1000)
```

Пример: (исходный файл: `fire_and_forget.js`)

```
(function() {
    var fire;
    fire = function(msg, wait) {
        return setTimeout(function() {
            return console.log(msg);
        }, wait);
    };
    fire("Hello", 3000);
    fire("World", 1000);
}).call(this);
```

Вывод: (исходный файл: `fire_and_forget.coffee`)

```
World
Hello
```

Как видите, программа сначала вывела слово «World», а затем «Hello». Синхронная программа сначала вывела бы слово «Hello», а затем, несколько секунд спустя, слово «World». Асинхронное программирование обладает мощным потенциалом, но также может оказаться трудоемким и сложным делом. Рассмотрим, как применение асинхронного программирования начинает вносить путаницу в программы.

Напишем метод `log`. Этот метод будет выводить сообщение о готовности обратиться к функции обратного вызова, затем вызывать эту функцию и, наконец, выводить сообщение о том, что функция была выполнена.

Пример: (исходный файл: unbound.coffee)

```
class User
    constructor: (@name) ->
    sayHi: ->
        console.log "Hello #{@name}"
bob = new User('bob')
mary = new User('mary')
log = (callback)->
    console.log "about to execute callback..."
    callback()
    console.log "...executed callback"
log(bob.sayHi)
log(mary.sayHi)
```

Пример: (исходный файл: unbound.js)

```
(function() {
    var User, bob, log, mary;
    User = (function() {
        function User(name) {
            this.name = name;
        }
        User.prototype.sayHi = function() {
            return console.log("Hello " + this.name);
        };
        return User;
    })();
    bob = new User('bob');
    mary = new User('mary');
    log = function(callback) {
        console.log("about to execute callback...");
        callback();
        return console.log("...executed callback");
    };
    log(bob.sayHi);
    log(mary.sayHi);
}).call(this);
```

Вывод: (исходный файл: unbound.coffee)

```
about to execute callback...
Hello undefined
...executed callback
```

```
about to execute callback...
Hello undefined
...executed  callback
```

Я был полностью уверен, что данный программный код поприветствует двух определенных пользователей. Что же произошло? Когда вызывается функция `log`, ей передается функция обратного вызова, которая теряет оригинальный контекст, откуда она была вызвана. Функция обратного вызова больше не имеет ссылки на переменную `name` нашего класса. Подобные проблемы типичны для языка JavaScript, особенно при использовании библиотек, таких как jQuery, для реализации AJAX-запросов или привязки обработчиков событий.

Возможно ли решить эту проблему? Как вернуть функцию обратного вызова в ее контекст? Сделать это можно, заменив оператор `->` оператором `=>`, известным также как «толстая стрелка», в определении функции `sayHi` в классе `User`. Ниже приводится тот же самый пример, но на этот раз я заменил в нем `sayHi: ->` на `sayHi: =>`. Посмотрим, что из этого получилось:

Пример: (исходный файл: `bound.coffee`)

```
class User
    constructor: (@name) ->
    sayHi: =>
        console.log "Hello #{@name}"
bob = new User('bob')
mary = new User('mary')
log = (callback)->
    console.log "about to execute callback..."
    callback()
    console.log "...executed callback"
log(bob.sayHi)
log(mary.sayHi)
```

Пример: (исходный файл: `bound.js`)

```
(function() {
    var User, bob, log, mary,
        __bind = function(fn, me){ return function(){ return fn.apply(me,
arguments); }; };
    User = (function() {
        function User(name) {
            this.name = name;
```

```
            this.sayHi = __bind(this.sayHi, this);
        }
        User.prototype.sayHi = function() {
            return console.log("Hello " + this.name);
        };
        return User;
    })();
    bob = new User('bob');
    mary = new User('mary');
    log = function(callback) {
        console.log("about to execute callback...");
        callback();
        return console.log("...executed callback");
    };
    log(bob.sayHi);
    log(mary.sayHi);
}).call(this);
```

Вывод: (исходный файл: `bound.coffee`)

```
about to execute callback...
Hello bob
...executed callback
about to execute callback...
Hello mary
...executed  callback
```

Отлично! Один простой символ решил проблему. Давайте сравним код на JavaScript, сгенерированный для примеров со связанным и несвязанным методом, чтобы лучше понять, что изменил единственный символ в нашем коде:

Пример: (исходный файл: `unbound.js`)

```
(function() {
    var User, bob, log, mary;
    User = (function() {
        function User(name) {
            this.name = name;
        }
        User.prototype.sayHi = function() {
            return console.log("Hello " + this.name);
        };
        return User;
```

```
    })();
    bob = new User('bob');
    mary = new User('mary');
    log = function(callback) {
        console.log("about to execute callback...");
        callback();
        return console.log("...executed callback");
    };
    log(bob.sayHi);
    log(mary.sayHi);
}).call(this);
```

Пример: (исходный файл: `bound.js`)

```
(function() {
    var User, bob, log, mary,
        __bind = function(fn, me){ return function(){ return fn.apply(me,
arguments); }; };
    User = (function() {
        function User(name) {
            this.name = name;
            this.sayHi = __bind(this.sayHi, this);
        }
        User.prototype.sayHi = function() {
            return console.log("Hello " + this.name);
        };
        return User;
    })();
    bob = new User('bob');
    mary = new User('mary');
    log = function(callback) {
        console.log("about to execute callback...");
        callback();
        return console.log("...executed callback");
    };
    log(bob.sayHi);
    log(mary.sayHi);
}).call(this);
```

Я дам некоторые пояснения к полученным результатам, но если вы не знаете, что делает функция `apply`, я предлагаю прочитать о ней в книге, посвященной языку JavaScript. Между сгенерированными файлами JavaScript имеются два отличия. Первое – наличие

функции `__bind` в примере, использующем оператор `=>`. Эта функция принимает два параметра: функцию, которую требуется привязать, и контекст, к которому требуется привязать функцию в первом параметре. Функция `__bind` возвращает новую функцию, вызывающую оригинальную функцию с помощью метода `apply`, которому передается требуемый контекст.

Второе отличие находится в конструкторе класса `User`. Он переопределяет функцию `sayHi` вызовом функции `__bind`, которой передает определение оригинальной функции `sayHi` и экземпляр класса в виде контекста.

Если что-то вам показалось непонятным, не переживайте – вы не одни. Контекст и привязка к контексту – очень сложная тема в языке JavaScript. Если она вам непонятна, могу порекомендовать два пути: первый – найти хороший учебник по языку JavaScript, и второй – прочитать статью «Understanding JavaScript Function Invocation and 'this.'» [5], где Йехуда Ктц (Yehuda Katz) доходчиво объясняет все это.

Если вы понимаете суть происходящего в программном коде на JavaScript, наверняка на вашем лице сейчас появилась улыбка от того, что вам больше не придется сталкиваться с проблемой связывания функций. Вы можете просто использовать оператор `=>` и получать удовольствие от жизни. Практическое применение этого оператора будет продемонстрировано в главе 11, «Пример: список задач, часть 2 (клиентская на основе jQuery)».

В заключение

По-моему мы отлично развлеклись. Классы в CoffeeScript, по крайней мере для меня, являются одной из самых привлекательных сторон языка. Я надеюсь, что эта глава помогла вам убедиться в этом.

Мы много чего рассмотрели в этой главе. Мы узнали, что такое классы в языке CoffeeScript и как определить самый простой класс.

Затем мы познакомились со «специальной» функцией `constructor` и долго говорили об области видимости классов.

Мы побеседовали о том, как наследовать классы в языке CoffeeScript с помощью ключевого слова `extends`, а также узнали как в дочерних классах расширять оригинальные функции родительского класса с помощью ключевого слова `super`.

После этого мы подробно обсудили функции класса и функции прототипа. Кроме того мы узнали о проблемах, сопутствующих

ключевому слову `super`, при неосторожном использовании в функциях класса.

Закончили мы эту главу изучением довольно сложной, но очень мощной концепции привязки функций с использованием оператора `=>`, также известного, как «толстая стрелка».

На данный момент вы узнали достаточно много о языке CoffeeScript. Поздравляю! Вы готовы покинуть школу. Здесь заканчивается первая часть книги «Основы CoffeeScript». Во торой части, «Практическое применение CoffeeScript» мы попробуем применить на практике знания, полученные в первой части, задействовав несколько популярных JavaScript-библиотек.

Примечания

1. http://ru.wikipedia.org/wiki/Класс_(программирование).
2. http://en.wikipedia.org/wiki/DRY.
3. http://ru.wikipedia.org/wiki/Наследование_(программирование).
4. http://en.wikipedia.org/wiki/Asynchronous_I/O.
5. http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this/.

Часть II: Практическое применение CoffeeScript

В этой половине книги мы соберем воедино все полученные сведения о языке CoffeeScript и попытаемся применить их на практике. Это поможет вам закрепить приобретенные знания, и попутно я познакомлю вас с некоторыми дополнительными советами и рекомендациями. При этом мы будем использовать один из самых популярных инструментов, применяемых при разработке программ на CoffeeScript.

Мы будем создавать файлы сборки Cakefile, чтобы обеспечить выполнение типичных операций с нашими приложениями. Увидим, насколько просто реализуется тестирование программного кода на CoffeeScript с помощью самого языка CoffeeScript и платформы тестирования Jasmine. Затем мы совершим короткий тур по фреймворку Node.js и создадим небольшое серверное приложение, выполняющее компиляцию файлов с программным кодом на CoffeeScript по мере необходимости.

В заключение мы напишем полноценное приложение – классическое приложение «управления задачами». Разработка этого приложения будет вестись на протяжении трех глав. В первой из них будет создана серверная часть приложения, реализующая хранение и обслуживание задач. Во второй главе будет создан интерфейс приложения с использованием популярной библиотеки jQuery. В третьей главе мы заменим библиотеку jQuery [1] фреймворком Backbone.js [2]. В этом приложении мы задействуем ряд очень интересных библиотек: jQuery, Backbone.js, Mongoose [3], Express [4] и, конечно же, CoffeeScript.

Так как знакомство с новыми технологиями будет происходить в процессе создания приложения, мы не будем углубляться в их изучение. Вместо этого мы затронем лишь самые основные моменты и будем рассматривать только то, что необходимо для работы. Итак, нас ждет самая интересная часть книги! Вперед!

Примечания

1. http://jquery.com.
2. http://documentcloud.github.com/backbone/.
3. http://mongoosejs.com/.
4. http://expressjs.com/.

7. Инструмент сборки Cake и файлы сборки Cakefile

В составе CoffeeScript имеется простой инструмент сборки под названием *Cake*, который по своей природе очень напоминает инструмент сборки Rake [1] для языка Ruby. Инструмент Cake позволяет определять простые задания, с целью помочь в создании проектов на CoffeeScript. Представьте, что вам требуется организовать запуск тестов или сборку файлов. Инструмент Cake позволяет легко определять такие задания в файле сборки с именем *Cakefile*.

В этой главе вы узнаете, как определять задания для инструмента Cake и как выполнять их, и, конечно же, все это будет делаться с использованием языка CoffeeScript.

> **Совет.** «Минутку! А как установить Cake?» Делать этого не потребуется! Во время установки CoffeeScript автоматически устанавливается и инструмент Cake, со своей утилитой командной строки, которая так и называется cake. Вам не нужно выполнять никаких дополнительных действий, чтобы получить доступ к этому инструменту.

Вступление

Прежде, чем мы напишем первое задание для Cake, необходимо разобраться с некоторыми особенностями работы этого инструмента. Первое, что необходимо знать об инструменте Cake, – все задания должны находиться в файле с именем Cakefile и этот файл должен располагаться в каталоге, где должны выполняться задания. Обычно это корневой каталог проекта.

Единственное, что следует помнить об инструменте Cake и файлах Cakefile, – файлы Cakefile должны быть написаны на языке CoffeeScript. Кроме того, при создании заданий в файлах Cakefile можно использовать несколько специальных функций. Мы познакомимся с ними поближе, когда будем определять задания для инструмента Cake.

Создание заданий для Cake

Давайте создадим первое задание для инструмента Cake – простое задание «hello world»:

Пример: (исходный файл: example1/Cakefile)

```
task "greet", "Say hi to the nice people", ->
      console.log "Hello, World!"
```

Пример: (исходный файл: example1/Cakefile.js)

```
(function() {
    task("greet", "Say hi to the nice people", function() {
        return console.log("Hello, World!");
    });
}).call(this);
```

> **Совет.** *В своей работе вам не придется сталкиваться с программным кодом на JavaScript, генерируемым на основе содержимого файлов Cakefile, тем не менее, я решил включить его в примеры для этой главы, чтобы вы лучше понимали, как действует CoffeeScript.*

Чтобы определить задание для инструмента Cake, следует вызвать функцию task, автоматически добавляемую во все файлы Cakefile. В первом аргументе этой функции передается имя задания, которое будет указываться в команде выполнения этого задания. Во втором необязательном аргументе передается текстовое описание задания. Если описание определено, оно будет отображено при выводе списка доступных заданий. В последнем аргументе функции task передается ссылка на функцию, которая будет вызвана для выполнения задания. Эта функция выполняет всю работу, связанную с заданием.

Чтобы вывести список доступных заданий, можно воспользоваться утилитой командной строки cake:

```
> cake
```

Если теперь запустить наш файл Cakefile, на экране появятся следующие строки:

Вывод: (исходный файл: example1/Cakefile)

```
Cakefile defines the following tasks:
cake greet                      # Say hi to the nice people
```

Как видите, на экран было выведено имя задания, созданного нами, с предшествующей ему командой `cake`, а также наше описание задания.

> **Совет.** Возможность вывода списка заданий становится чрезвычайно удобной, как только вы начинаете использовать библиотеки, имеющие собственные задания для инструмента Cake.

Выполнение заданий

Теперь, когда задание создано, как запустить его? Легко. При запуске команды `cake` достаточно сообщить ей, какое задание следует выполнить. Просто введите команду `cake` и имя требуемого задания, как показано ниже:

```
> cake greet
```

Эта команда выполнит задание `greet`.

Вывод:

```
Hello, World!
```

Использование параметров

Мы написали первое задание и узнали, как его выполнить, но что если для выполнения задания потребуется передать какие-нибудь аргументы? Например, заданию `greet` может потребоваться передать параметр, уточняющий текст приветствия. Посмотрим, как это сделать.

Прежде всего, чтобы иметь возможность передать параметр, его нужно определить. Делается это с помощью специальной функции `option`, определяемой инструментом Cake. Эта функция принимает три аргумента. Первый – *«краткая» форма параметра*, второй – *«длинная» форма параметра* и последний аргумент – простое описание назначения параметра. Вернемся к нашему заданию `greet` и определим параметр, с помощью которого можно будет уточнять текст приветствия.

> **Совет.** Когда я говорю о «краткой» или «длинной» форме параметра, я подразумеваю, как много символов придется вводить пользователю с клавиатуры. Например, `-n` – краткая форма, а `--name` – длинная форма параметра.

Пример: (исходный файл: example2/Cakefile)

```
option '-n', '--name [NAME]', 'name you want to greet'
task "greet", "Say hi to someone", (options)->
    message = "Hello, "
    if options.name?
        message += options.name
    else
        message += "World"
    console.log  message
```

Пример: (исходный файл: example2/Cakefile.js)

```
(function() {
    option('-n', '--name [NAME]', 'name you want to greet');
    task("greet", "Say hi to someone", function(options) {
        var message;
        message = "Hello, ";
        if (options.name != null) {
            message += options.name;
        } else {
            message += "World";
        }
        return console.log(message);
    });
}).call(this);
```

В этой реализации задания вызывается функция option, которой передается три обязательных аргумента. Первый аргумент, -n, определяет краткую форму параметра. Второй аргумент, --name, определяет длинную форму параметра. Обратите внимание, как в этом примере определяется длинная форма параметра, я добавил в нее строку [NAME], которая сообщает инструменту Cake, что здесь ожидается некоторое значение. Если в определении длинной формы параметра не указать нечто, похожее на [NAME], Cake сгенерирует ошибку при попытке передать значение вместе с параметром. Последний аргумент является описанием.

> **Совет.** Несмотря на то, что в функции option обязательными являются все три аргумента, по-настоящему необходимы только два последних из них. Вы должны определить обе формы параметра, краткую и длинную, а также описание, однако краткая форма не является обязательной. Если по каким-то причинам вы не желаете определять краткую форму параметра, передайте функции option пустую строку или значение null в первом аргументе.

Если теперь попробовать запустить команду cake, чтобы вывести список доступных заданий, вы должны увидеть следующее:

Вывод: (исходный файл: example2/Cakefile)

```
Cakefile defines the following tasks:
cake greet                # Say hi to someone
      -n, --name           name you want to greet
```

Внизу можно видеть список параметров для заданий.

> **Совет.** Хочу отметить некоторые недостатки параметров заданий Cake. Параметры не связаны с определенными заданиями – они определяются для всех заданий сразу. Если в дополнение к заданию greet определить еще одно задание, оба они смогут принимать параметр --name. Хотя это не такая уж большая проблема, тем не менее, необходимо с особой осторожностью подходить к выбору имен параметров и сопровождать их соответствующими описаниями.

Если еще раз взглянуть на задание greet, можно заметить, что функции, реализующей это задание, передается объект options:

Пример: (исходный файл: example2/Cakefile)

```
option '-n', '--name [NAME]', 'name you want to greet'
task "greet", "Say hi to someone", (options)->
    message = "Hello, "
    if options.name?
        message += options.name
    else
        message += "World"
    console.log message
```

Пример: (исходный файл: example2/Cakefile.js)

```
(function() {
    option('-n', '--name [NAME]', 'name you want to greet');
    task("greet", "Say hi to someone", function(options) {
        var message;
        message = "Hello, ";
        if (options.name != null) {
            message += options.name;
        } else {
            message += "World";
        }
```

```
        return console.log(message);
    });
}).call(this);
```

С помощью объекта `options` можно определить, было ли вызвано задание с параметром `--name`. В этом случае выводится текст приветствия, содержащий указанное имя, иначе используется универсальный текст приветствия.

Выполнить задание `greet` с параметром `--name` можно следующим образом:

```
> cake -n Mark greet
```

Вывод: (исходный файл: `example2a/Cakefile`)

```
Hello, Mark
```

Если необходимо сделать параметр обязательным, для этого следует вручную реализовать проверку наличия параметра в определении задания и возбудить ошибку в случае его отсутствия:

Пример: (исходный файл: `example2a/Cakefile`)

```
option '-n', '--name [NAME]', 'name you want to greet'
task "greet", "Say hi to someone", (options)->
    throw new Error("[NAME] is required") unless options.name?
    console.log "Hello, #{options.name}"
```

Пример: (исходный файл: `example2a/Cakefile.js`)

```
(function() {
    option('-n', '--name [NAME]', 'name you want to greet');
    task("greet", "Say hi to someone", function(options) {
        if (options.name == null) throw new Error("[NAME] is required");
        return console.log("Hello, " + options.name);
    });
}).call(this);
```

Вывод: (исходный файл: `example2a/Cakefile`)

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
```

```
Error: [NAME] is required
    at Object.action (.../cake/example2a/Cakefile:6:37)
    at /usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/cake.js:39:26
    at Object.run (/usr/local/lib/node_modules/coffee-script/lib/
↪coffee-script/cake.js:62:21)
    at Object.<anonymous> (/usr/local/lib/node_modules/
↪coffee-script/bin/cake:7:38)
    at Module._compile (module.js:432:26)
    at Object..js (module.js:450:10)
    at Module.load (module.js:351:31)
    at Function._load (module.js:310:12)
    at Array.0 (module.js:470:10)
      at  EventEmitter._tickCallback  (node.js:192:40)
```

Совет. При использовании параметров заданий важно помнить, что все параметры помещаются в командной строке перед именем задания. Если поместить параметры после имени задания, вы получите сообщение об ошибке. Лично я предпочел бы иметь возможность вводить команду, как: `cake greet -n Mark`, но, к сожалению, на данный момент это невозможно.

Вызов других заданий

Иногда бывает необходимо *вызывать одни задания из других*. Например: обычно для проектов создаются два задания, одно из которых очищает каталоги сборки, а другое компилирует и собирает проект. Напишем эти два задания:

Пример: (исходный файл: `example3/Cakefile`)

```
task "clean", "Clean up build directories", ->
    console.log "cleaning up..."
task "build", "Build the project files", ->
    console.log  "building..."
```

Пример: (исходный файл: `example3/Cakefile.js`)

```
(function() {
    task("clean", "Clean up build directories", function() {
        return console.log("cleaning up...");
    });
    task("build", "Build the project files", function() {
```

```
        return console.log("building...");
    });
}).call(this);
```

Вывод: (исходный файл: example3/Cakefile)

```
Cakefile defines the following tasks:
cake clean               # Clean up build directories
cake build               # Build the project files
```

После использования этих заданий в течение некоторого времени, вы заметите, что всегда сначала выполняете задание clean, а затем задание build. Оба задания можно выполнить одной командой, как показано ниже:

```
> cake clean build
```

Она выполнит оба задания. Но, что если появится третье задание, package, создающее дистрибутив проекта? Прежде чем создать пакет дистрибутива, было бы желательно убедиться прежде, что проект собран, а перед сборкой – что каталоги сборки очищены. Сделать это можно было бы так:

```
> cake clean build package
```

Проблема в том, что при таком подходе легко ошибиться. Например, можно забыть указать задание build или clean . Что в этом случае получится? К счастью, инструмент Cake позволяет вызывать одни задания из других. Для этого нужно вызвать функцию invoke, определяемую инструментом Cake, и передать ей имя требуемого задания:

Пример: (исходный файл: example4/Cakefile)

```
task "clean", "Clean up build directories", ->
    console.log "cleaning up..."
task "build", "Build the project files", ->
    console.log "building..."
task "package", "Clean, build, and package the project", ->
    invoke "clean"
    invoke "build"
    console.log "packaging..."
```

Пример: (исходный файл: example4/Cakefile.js)

```
(function() {
    task("clean", "Clean up build directories", function() {
        return console.log("cleaning up...");
    });
    task("build", "Build the project files", function() {
        return console.log("building...");
    });
    task("package", "Clean, build, and package the project", function() {
        invoke("clean");
        invoke("build");
        return console.log("packaging...");
    });
}).call(this);
```

Вывод: (исходный файл: `example4/Cakefile`)

```
Cakefile defines the following tasks:
cake clean               # Clean up build directories
cake build               # Build the project files
cake package                    # Clean, build, and package the project
```

Теперь запустите задание `package` и убедитесь, что оба задания, `clean` и `build`, также были выполнены:

```
> cake package
```

Вывод: (исходный файл: `example4/Cakefile`)

```
cleaning up...
building...
packaging...
```

> **Совет.** Обратите внимание, что когда вызываются другие задания, они выполняются асинхронно. Поэтому в нашем примере не гарантируется, что задание `clean` завершится до того, как будет запущено задание `build`. Такое поведение может приводить к ошибкам, поэтому будьте внимательнее. То же относится и к случаю, когда одной командой `cake` выполняется цепочка заданий. Будьте осмотрительны!

В заключение

В этой главе мы познакомились с инструментом сборки Cake, распространяемым в составе CoffeeScript. Вы узнали, как определять

новые задания и как получить список доступных заданий. Вы научились запускать задания, передавать им параметры и выполнять несколько заданий одновременно. Вам может показаться, что вы узнали не так уж и много об особенностях создания заданий для инструмента Cake. Однако это все, что он может предложить. За вами остается наполнить эти задания необходимой реализацией. Проект Node.js [2], о котором рассказывается в главе 9, «Введение в Node.js», является отличной отправной точкой для поиска дополнительных примеров, где можно найти модули, реализующие чтение и запись в файлы и каталоги, компиляцию файлов на языке CoffeeScript, выполнение HTTP-запросов, и многое другое. В других главах вы найдете файлы Cakefile, реализующие такие операции, как запуск тестов, поэтому обратите внимание и на эти примеры.

Cake – отличный инструмент, и замечательно, что он устанавливается автоматически, вместе с CoffeeScript, но на мой взгляд он обладает слишком ограниченными и не всегда удобными возможностями для большинства заданий, которые мне приходится писать. Используя модули из пакета Node.js и другие доступные модули, я могу писать весьма мощные задания, но иногда я все-таки возвращаюсь к своим корням в Ruby и использую инструмент Rake, потому что он является более развитым, нежели Cake.

Однако я настоятельно рекомендую попытаться использовать инструмент Cake. Он может оказаться именно тем, что вам нужно. Если вы ограничены в выборе используемых и установленных языков, тогда у вас просто не остается выбора.

Примечания

1. https://github.com/jimweirich/rake.
2. http://nodejs.org/.

8. Тестирование с помощью Jasmine

Для меня нет большего греха, чем писать программный код без тестов. Я свято убежден, что тестирование не является чем-то, от чего можно отказаться. [1] Тестирование является обязательным условием. Я настолько силен в своей вере, что на практике использую принцип, известный как *«разработка через тестирование»* (Test-Driven Development, TDD). [2] TDD, то есть разработка через тестирование, – это философия разработки программного кода, требующая, чтобы сначала создавались контрольные примеры (или тесты) и только потом сам программный код. Используя этот принцип можно спать спокойно, зная, что программный код хорошо протестирован, а если что-то произойдет, вы легко и быстро сможете исправить ошибку, пребывая в уверенности, что при этом ничего больше не сломали.

Овладение искусством TDD может оказаться непростым делом – многие просто не знают, с чего начать. Если вам необходим какой-нибудь учебник, который поможет вам использовать принцип разработки через тестирование, могу порекомендовать небольшую заметку в блоге, написанную вашим покорным слугой, под названием «How to Become a Test-driven Developer» (Как начать использовать принцип разработки через тестирование). [3]

В этой главе я хочу представить инструмент, который считаю одним из лучших инструментов для тестирования приложений на JavaScript – Jasmine. [4]

> **Совет.** Jasmine – это не только инструмент тестирования приложений на JavaScript. В действительности он обладает намного более широкими возможностями. Я считаю Jasmine одним из лучших инструментов, потому что он близко напоминает мою любимую платформу тестирования для Ruby, Rspec. [5] Из других инструментов тестирования программного кода на JavaScript могу порекомендовать обратить внимание на QUnit, [6] JsTestDriver, [7] и YUI Test. [8]

Здесь я не буду раскрывать все подробности об инструменте Jasmine. В Интернете можно найти массу статей и видеороликов,

детально описывающих работу с Jasmine. Я лишь покажу, как использовать Jasmine для тестирования программного кода на CoffeeScript, чтобы вы могли увидеть, что это за инструмент и познакомиться с его возможностями.

Установка Jasmine

Обычно, как я уже говорил в начале книги, я не люблю описывать процесс установки инструментов в своих книгах. Причина достаточно очевидна – инструкции быстро устаревают уже к тому моменту, когда я заканчиваю их писать, не говоря уже о моменте выхода книги. Та же проблема наблюдается и здесь.

Тем не менее, я отступлю от своего правила и расскажу об одном из лучших способов *установки инструмента Jasmine* и настройки в нем поддержки CoffeeScript. Таким способом является установка и использование пакета jasmine-headless-webkit [9] для Ruby. Поскольку этот пакет изначально предназначен для Ruby, он требует установки поддержки языка Ruby. Он также тянет за собой несколько других зависимостей. По ссылке в примечании приводятся подробные инструкции, описывающие процесс установки.

Существуют и другие, менее замысловатые способы установки и настройки Jasmine. Однако они не позволяют получить встроенную поддержку языка CoffeeScript и требуют компиляции исходных текстов программ и тестов, а кому захочется иметь дело со всем этим?

Нет ничего плохого, если вы предпочтете использовать другую версию Jasmin вместо jasmine-headless-webkit. Однако в этой главе мы будем использовать именно jasmine-headless-webkit, поэтому я рекомендую установить этот пакет, если вы собираетесь опробовать примеры, представленные здесь.

Настройка Jasmine

Я буду полагать, что вы уже установили инструмент Jasmine и готовы двигаться дальше. Итак, начнем.

В этой главе мы создадим простое приложение калькулятора. Оно будет выполнять самые основные операции, свойственные калькуляторам: сложение, вычитании, умножение и деление. Создайте новую папку для проекта. Внутри этой папки выполните следующую команду для *настройки инструмента Jasmine*:

```
> jasmine init
```

В результате выполнения этой команды в каталоге проекта должно появиться несколько новых файлов и папок, как показано ниже:

```
public/
    javascripts/
        Player.js
        Song.js
Rakefile
spec/
    javascripts/
        helpers/
            SpecHelper.js
        PlayerSpec.js
    support/
        jasmine.yml
        jasmine_config.rb
        jasmine_runner.rb
```

В процессе настройки Jasmine добавляет несколько файлов примеров, чтобы помочь нам понять, как работает этот инструмент, и убедиться, что все настройки выполнены правильно. Теперь проверим настройки:

```
> jasmine-headless-webkit -c
```

Если вам сопутствует удача, на экране должны появиться примерно такие строки:

```
Running Jasmine specs...
.....
PASS: 5 tests, 0 failures, 0.009 secs.
```

Отлично! Инструмент Jasmine настроен и готов к работе. Пока все выглядит хорошо. Хорошо, кроме этой жуткой команды, которую придется набирать в командной строке всякий раз, когда потребуется выполнить тесты. Попробуем воспользоваться знаниями, полученными в главе 7, «Инструмент сборки Cake и файлы сборки Cakefile», и написать задание для инструмента Cake, которое упростит нам жизнь.

Сначала я покажу содержимое файла Cakefile, а потом объясню его.

Пример: (исходный файл: `calc.1/Cakefile`)

```
exec = require('child_process').exec
task "test", (options) =>
    exec "jasmine-headless-webkit -c", (error, stdout, stderr)->
        console.log  stdout
```

Пример: (исходный файл: `calc.1/Cakefile.js`)

```
(function() {
    var exec,
        _this = this;
    exec = require('child_process').exec;
    task("test", function(options) {
        return exec("jasmine-headless-webkit -c",
                    function(error, stdout, stderr) {
                        return console.log(stdout);
                    });
    });
}).call(this);
```

Вывод: (исходный файл: `calc.1/Cakefile`)

```
Cakefile defines the following tasks:
cake  test
```

Это определенно не самое сложное задание для Cake, но оно заслуживает некоторых пояснений. Основное волшебство этого задания сосредоточено в первой строке. Эта строка импортирует модуль `child_process` из комплекта инструментов Node.js [10]. Не волнуйтесь, если вам не понятно, что это такое и как оно работает – более подробно об импортировании модулей будет рассказываться в главе 9, «Введение в Node.js».

В составе модуля `child_process` мы получаем функцию `exec`. Эта функция позволяет выполнять команды в консоли и захватывать вывод этих команд – именно это и делает наше задание для Cake.

Мы создали задание с именем `test`, которое выполняет команду `jasmine-headless-webkit -c`, запускающую тесты. По окончании тестирования вызывается наша функция обратного вызова, и результаты тестирования выводятся в консоль.

Теперь, если ввести команду:

```
> cake test
```

на экране должны появиться результаты тестирования, подобные тем, что мы видели выше.

Теперь, немного упростив команду, выполним заключительные настройки проекта, перед тем как углубиться в создание тестов.

Для начала немного почистим папки проекта и удалим все ненужные файлы примеров, сгенерированные инструментом Jasmin. Удалите все файлы и папки, чтобы получить следующую структуру дерева каталогов проекта:

```
src/
Rakefile
spec/
    javascripts/
        helpers/
    support/
        jasmine.yml
        jasmine_config.rb
        jasmine_runner.rb
```

Хорошо, мы почти закончили. Последнее, что осталось сделать, – изменить настройки в файле `spec/javascripts/support/jasmine.yml` с учетом особенностей нашего проекта:

Пример: (исходный файл: `calc.2/spec/javascripts/support/jasmine.yml`)

```
src_files:
    - "**/*.coffee"
helpers:
    - "helpers/**/*.coffee"
spec_files:
    - /**/*_spec.coffee
src_dir: "src"
spec_dir:  spec/javascripts
```

Теперь все готово к изучению возможностей инструмента Jasmine!

Введение в Jasmine

Итак, как же выглядят тесты, выполняемые инструментом Jasmine? Создадим один такой тест, и вы увидите.

Создайте в каталоге `spec` новый файл с именем `calculator_spec.coffee`. В нем будут собраны все тесты, или, как их еще называют,

спецификации, для калькулятора. Сначала взгляните, как выглядит наш первый тест, а затем мы обсудим его, чтобы лучше понять, как он действует.

Пример: (исходный файл: `calc.3/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
it "does something", ->
    expect(1 + 1).toEqual 2
    expect(1  +  1).not.toEqual  3
```

Пример: (исходный файл: `calc.3/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        return it("does something", function() {
            expect(1 + 1).toEqual(2);
            return expect(1 + 1).not.toEqual(3);
        });
    });
}).call(this);
```

Прежде всего необходимо создать *блок описания*. Этот блок определяет объект испытаний. Обычно в этом блоке описываются классы или функции. В данном случае был описан класс `Calculator`, который мы собираемся создать. Первым аргументом функции `describe` передается строка, представляющая объект испытаний – «Calculator». Во втором аргументе передается функция, содержащая все тесты, связанные с объектом испытаний.

Внутри функции обратного вызова, которая передается функции `describe`, определяются блоки `it`. Блок `it`, напоминающий обычную функцию, или блок, `describe`, и принимает два аргумента. Первый аргумент – строка, описывающая, что планируется сделать в ходе тестирования. В этом примере мы собираемся протестировать «некоторые операции» «калькулятора». Второй аргумент – функция, содержащая утверждения, истинность которых требуется проверить.

В нашем тесте, проверяющем «некоторые операции», утверждается, что `1 + 1` равно `2`. Мы также торжественно утверждаем, что `1 + 1` не равно `3`. Тестирование выполняется с помощью методов сопоставления. В данном случае мы используем метод `toEqual`.

Методы сопоставления следуют одному простому правилу: если он возвращает `true`, считается, что тест пройден, в противном случае считается, что тест не пройден.

А что делает функция `expect` и зачем она нужна? Функция `expect` принимает в виде аргумента проверяемую инструкцию, в нашем случае – выражение `1 + 1`, и возвращает специальный объект, владеющий методами сопоставления. В двух словах, эта функция упрощает использование инструмента Jasmine. Кроме того, она улучшает удобочитаемость тестов.

> **Совет.** В Jasmine имеется множество удобных методов сопоставления, которые можно использовать для проверки практически любых утверждений. Полный перечень методов можно найти на сайте с документацией [11] для инструмента Jasmine.

Модульное тестирование

Теперь, когда мы получили некоторое представление, как писать тесты для инструмента Jasmine, рассмотрим подробно, как будут выглядеть тесты для нашего класса `Calculator`. Для начала удалим примеры реализации теста и затем добавим блоки `describe` для тестирования операций сложения, вычитания, умножения и деления:

Пример: (исходный файл: `calc.4/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    describe "#add", ->
        it "adds two numbers", ->
    describe "#subtract", ->
        it "subtracts two numbers", ->
    describe "#multiply", ->
        it "multiplies two numbers", ->
    describe "#divide", ->
        it "divides two numbers", ->
```

Пример: (исходный файл: `calc.4/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
```

```
        describe("#add", function() {
            return it("adds two numbers", function() {});
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {});
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {});
        });
        return describe("#divide", function() {
            return it("divides two numbers", function() {});
        });
    });
}).call(this);
```

Посмотрев на этот фрагмент, вы можете задаться вопросом: зачем потребовалось вставлять несколько блоков `describe` в начальный блок `describe`? Такая организация выбрана потому, что мы собираемся писать тесты для каждой из четырех функций в классе `Calculator`. Каждая из этих функций является предметом испытаний. Вслед за описаниями предметов испытаний следуют блоки `it`, определяющие реализацию тестирования.

Выполнить тестирование можно следующей командой:

```
> cake test
```

Вывод этой команды должен иметь следующий вид:

```
Running Jasmine specs...
....
PASS: 4 tests, 0 failures, 0.02 secs.
```

> **Совет.** Перед именем каждой тестируемой функции в нашем тесте стоит символ #, который указывает, что эта функция является функцией экземпляра. Если бы в тесте описывались функции класса, вместо символа # использовался бы символ точки (.).

Все тесты в нашем примере выполняются благополучно, потому что пока внутри блоков `it` ничего нет. Наполним их:

Пример: (исходный файл: `calc.5/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    describe "#add", ->
        it "adds two numbers", ->
            calculator = new Calculator()
            expect(calculator.add(1, 1)).toEqual 2
    describe "#subtract", ->
        it "subtracts two numbers", ->
            calculator = new Calculator()
            expect(calculator.subtract(10, 1)).toEqual 9
    describe "#multiply", ->
        it "multiplies two numbers", ->
            calculator = new Calculator()
            expect(calculator.multiply(5, 4)).toEqual 20
    describe "#divide", ->
        it "divides two numbers", ->
            calculator = new Calculator()
            expect(calculator.divide(20, 5)).toEqual 4
```

Пример: (исходный файл: `calc.5/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        describe("#add", function() {
            return it("adds two numbers", function() {
                var calculator;
                calculator = new Calculator();
                return expect(calculator.add(1, 1)).toEqual(2);
            });
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {
                var calculator;
                calculator = new Calculator();
                return expect(calculator.subtract(10, 1)).toEqual(9);
            });
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {
                var calculator;
                calculator = new Calculator();
                return expect(calculator.multiply(5, 4)).toEqual(20);
            });
        });
```

```
        return describe("#divide", function() {
            return it("divides two numbers", function() {
                var calculator;
                calculator = new Calculator();
                return expect(calculator.divide(20, 5)).toEqual(4);
            });
        });
    });
}).call(this);
```

Теперь мы подобрались к самой сути. У нас есть прекрасно выглядящие тесты, описывающие и тестирующие четыре функции, которые мы добавим в класс `Calculator`. А что произойдет, если запустить эти тесты прямо сейчас?

```
Running Jasmine specs...
FFFF
Calculator #add adds two numbers.
↪(../jasmine/calc.5/spec/javascripts/calculator_spec.coffee:5)
  ReferenceError: Can't find variable: Calculator in ../jasmine/calc.5/spec/
↪ javascripts/calculator_spec.coffee (line ~6)
Calculator #subtract subtracts two numbers.
↪(../jasmine/calc.5/spec/javascripts/calculator_spec.coffee:11)
  ReferenceError: Can't find variable: Calculator in ../jasmine/calc.5/spec/
↪ javascripts/calculator_spec.coffee (line ~13)
Calculator #multiply multiplies two numbers.
↪(../jasmine/calc.5/spec/javascripts/calculator_spec.coffee:17)
  ReferenceError: Can't find variable: Calculator in ../jasmine/calc.5/spec/
↪ javascripts/calculator_spec.coffee (line ~20)
Calculator #divide divides two numbers.
↪(../jasmine/calc.5/spec/javascripts/calculator_spec.coffee:23)
  ReferenceError: Can't find variable: Calculator in ../jasmine/calc.5/spec/
↪ javascripts/calculator_spec.coffee (line ~27)
FAIL: 4 tests, 4 failures, 0.017 secs.
180 Chapter 8 Testing with Jasmine
```

Все тесты провалились, и по веской причине – класс `Calculator` пока не реализован! Сделаем это, тем более что класс очень прост:

Пример: (исходный файл: `calc.6/src/calculator.coffee`)

```
class @Calculator
    add: (a, b) ->
        a + b
```

```
    subtract: (a, b) ->
        a - b
    multiply: (a, b) ->
        a * b
    divide: (a, b) ->
        a / b
```

Пример: (исходный файл: `calc.6/src/calculator.js`)

```
(function() {
    this.Calculator = (function() {
        function Calculator() {}
        Calculator.prototype.add = function(a, b) {
            return a + b;
        };
        Calculator.prototype.subtract = function(a, b) {
            return a - b;
        };
        Calculator.prototype.multiply = function(a, b) {
            return a * b;
        };
        Calculator.prototype.divide = function(a, b) {
            return a / b;
        };
        return Calculator;
    })();
}).call(this);
```

Если теперь снова запустить тестирование, все тесты будут пройдены успешно:

```
Running Jasmine specs...
....
PASS: 4 tests, 0 failures, 0.021 secs.
```

До и после

Наши тесты выглядят замечательно, но в каждом тесте выполняется множество повторяющихся действий. В каждом тесте создается новый экземпляр класса `Calculator`. Инструмент Jasmine позволяет избавиться от повторяющихся операций за счет реализации функции `beforeEach`.

Совет. Как можно было бы догадаться, инструмент Jasmin позволяет также определить функцию `afterEach`. Ее удобно использовать для приведения базы данных или другого хранилища в исходное состояние, предшествовавшее запуску предыдущего теста.

Переместим создание экземпляра класса `Calculator` в определение функции `beforeEach`. Для этого достаточно определить свою функцию `beforeEach`, которая должна вызываться перед выполнением каждого блока `it` в текущем, а также во всех последующих блоках `describe`.

Пример: (исходный файл: `calc.7/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    beforeEach ->
        @calculator = new Calculator()
    describe "#add", ->
        it "adds two numbers", ->
            expect(@calculator.add(1, 1)).toEqual 2
    describe "#subtract", ->
        it "subtracts two numbers", ->
            expect(@calculator.subtract(10, 1)).toEqual 9
    describe "#multiply", ->
        it "multiplies two numbers", ->
            expect(@calculator.multiply(5, 4)).toEqual 20
    describe "#divide", ->
        it "divides two numbers", ->
            expect(@calculator.divide(20, 5)).toEqual 4
```

Пример: (исходный файл: `calc.7/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        beforeEach(function() {
            return this.calculator = new Calculator();
        });
        describe("#add", function() {
            return it("adds two numbers", function() {
                return expect(this.calculator.add(1, 1)).toEqual(2);
            });
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {
```

```
                return expect(this.calculator.subtract(10, 1)).toEqual(9);
            });
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {
                return expect(this.calculator.multiply(5, 4)).toEqual(20);
            });
        });
        return describe("#divide", function() {
            return it("divides two numbers", function() {
                return expect(this.calculator.divide(20, 5)).toEqual(4);
            });
        });
    });
}).call(this);
```

> **Совет.** Область действия функций `beforeEach` и `afterEach` простирается вниз до конца текущего блока `describe`, подобно водопаду, и распространяется на все вложенные блоки `describe`, независимо от глубины их вложенности.

При создании функций `beforeEach` важно помнить, что их может быть сколько угодно, и они могут помещаться на любой уровень вложенности. Убедимся в этом на примере класса `Calculator`.

Добавим в класс `Calculator` флаг, определяющий режим научных вычислений.

Пример: (исходный файл: `calc.8/src/calculator.coffee`)

```
class @Calculator
    constructor: (@scientific = false)->
    add: (a, b) ->
        a + b
    subtract: (a, b) ->
        a - b
    multiply: (a, b) ->
        a * b
    divide: (a, b) ->
        a / b
```

Пример: (исходный файл: `calc.8/src/calculator.js`)

```
(function() {
    this.Calculator = (function() {
```

```
        function Calculator(scientific) {
            this.scientific = scientific != null ? scientific : false;
        }
        Calculator.prototype.add = function(a, b) {
            return a + b;
        };
        Calculator.prototype.subtract = function(a, b) {
            return a - b;
        };
        Calculator.prototype.multiply = function(a, b) {
            return a * b;
        };
        Calculator.prototype.divide = function(a, b) {
            return a / b;
        };
        return Calculator;
    })();
}).call(this);
```

Затем добавим тест, утверждающий, что по умолчанию режим научных вычислений выключен:

Пример: (исходный файл: `calc.8/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    beforeEach ->
        @calculator = new Calculator()
    it "is not in scientific mode by default", ->
        expect(@calculator.scientific).toBeFalse()
    describe "#add", ->
        it "adds two numbers", ->
            expect(@calculator.add(1, 1)).toEqual 2
    describe "#subtract", ->
        it "subtracts two numbers", ->
            expect(@calculator.subtract(10, 1)).toEqual 9
    describe "#multiply", ->
        it "multiplies two numbers", ->
            expect(@calculator.multiply(5, 4)).toEqual 20
    describe "#divide", ->
        it "divides two numbers", ->
            expect(@calculator.divide(20, 5)).toEqual 4
```

Пример: (исходный файл: `calc.8/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        beforeEach(function() {
            return this.calculator = new Calculator();
        });
        it("is not in scientific mode by default", function() {
            return expect(this.calculator.scientific).toBeFalse();
        });
        describe("#add", function() {
            return it("adds two numbers", function() {
                return expect(this.calculator.add(1, 1)).toEqual(2);
            });
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {
                return expect(this.calculator.subtract(10, 1)).toEqual(9);
            });
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {
                return expect(this.calculator.multiply(5, 4)).toEqual(20);
            });
        });
        return describe("#divide", function() {
            return it("divides two numbers", function() {
                return expect(this.calculator.divide(20, 5)).toEqual(4);
            });
        });
    });
}).call(this);
Running Jasmine specs...
....
PASS: 5 tests, 0 failures, 0.021 secs.
```

Теперь добавим еще один блок `describe`, описывающий класс `Calculator`, который действует в режиме научных вычислений, и добавим в этот блок функцию `beforeEach`, создающую новый экземпляр класса `Calculator` с включенным режимом научных вычислений. Напишем также тест, проверяющий включение режима научных вычислений:

Пример: (исходный файл: `calc.9/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    beforeEach ->
        @calculator = new Calculator()
    it "is not in scientific mode by default", ->
        expect(@calculator.scientific).toBeFalse()
    describe "scientific mode", ->
        beforeEach ->
            @calculator = new Calculator(true)
        it "is in scientific mode when set", ->
            expect(@calculator.scientific).toBeTruth()
    describe "#add", ->
        it "adds two numbers", ->
            expect(@calculator.add(1, 1)).toEqual 2
    describe "#subtract", ->
        it "subtracts two numbers", ->
            expect(@calculator.subtract(10, 1)).toEqual 9
    describe "#multiply", ->
        it "multiplies two numbers", ->
            expect(@calculator.multiply(5, 4)).toEqual 20
    describe "#divide", ->
        it "divides two numbers", ->
            expect(@calculator.divide(20, 5)).toEqual 4
```

Пример: (исходный файл: `calc.9/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        beforeEach(function() {
            return this.calculator = new Calculator();
        });
        it("is not in scientific mode by default", function() {
            return expect(this.calculator.scientific).toBeFalse();
        });
        describe("scientific mode", function() {
            beforeEach(function() {
                return this.calculator = new Calculator(true);
            });
            return it("is in scientific mode when set", function() {
                return expect(this.calculator.scientific).toBeTruth();
            });
        });
        describe("#add", function() {
```

```
            return it("adds two numbers", function() {
                return expect(this.calculator.add(1, 1)).toEqual(2);
            });
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {
                return expect(this.calculator.subtract(10, 1)).toEqual(9);
            });
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {
                return expect(this.calculator.multiply(5, 4)).toEqual(20);
            });
        });
        return describe("#divide", function() {
            return it("divides two numbers", function() {
                return expect(this.calculator.divide(20, 5)).toEqual(4);
            });
        });
    });
}).call(this);
Running Jasmine specs...
......
PASS: 6 tests, 0 failures, 0.017 secs.
```

Собственные методы сопоставления

Прежде чем завершить разработку класса `Calculator`, попробуем упростить наши тесты за счет использования *собственного метода сопоставления*, проверяющего режим вычислений калькулятора. Инструмент Jasmine обеспечивает простой способ определения собственных методов сопоставления.

Создайте в каталоге `spec/javascripts/helpers directory` файл с именем `to_be_scientific.coffee`.

> **Совет.** Выбор имен для файлов в каталоге `spec/javascripts/helpers` не имеет большого значения, но я предпочитаю давать описательные имена. Если создавать файлы с именами, соответствующими именам методов сопоставления, реализованных в этих файлах, их проще будет отыскать позднее, когда потребуется внести какие-нибудь изменения.

Добавьте следующий код в этот файл:

Пример:
(исходный файл: `calc.10/spec/javascripts/helpers/to_be_scientific.coffee`)

```
beforeEach ->
    @addMatchers
        toBeScientific: ->
                @actual.scientific is true
```

Пример:
(исходный файл: `calc.10/spec/javascripts/helpers/to_be_scientific.js`)

```
(function() {
    beforeEach(function() {
        return this.addMatchers({
            toBeScientific: function() {
                return this.actual.scientific === true;
            }
        });
    });
}).call(this);
```

> **Совет.** Методы сопоставления необязательно помещать в отдельные файлы. Их можно определить в единственном вспомогательном файле. Лично я предпочитаю определять методы в отдельных файлах. На мой взгляд, это делает программный код проще и понятнее. При желании, одноразовые методы сопоставления можно также определять непосредственно в блоках `describe`.

Чтобы определить собственный метод сопоставления, прежде всего необходимо определить функцию `beforeEach`, которая будет вызываться перед выполнением всего набора тестов. Внутри этой функции следует вызвать встроенную функцию `addMatchers`, которая делает именно то, что следует из ее имени. Она принимает объект, содержащий имена добавляемых методов сопоставления и функции, реализующие их. Собственный метод сопоставления должен возвращать либо `true`, либо `false`. Помните, как выше в этой главе я говорил, что если утверждение возвращает `true`, инструмент Jasmin считает, что тест пройден, в противном случае тест считается не пройденным? Это как раз то место, где определяется данное поведение.

В нашем методе сопоставления `toBeScientific` будет проверяться утверждение, что в тестируемом экземпляре класса `Calculator` включен флаг режима научных вычислений, и исходя из значения этого флага будет возвращаться значение `true` или `false`.

Определив реализацию собственного метода сопоставления, можно задействовать его в наших тестах, как показано ниже:

Пример: (исходный файл: `calc.10/spec/javascripts/calculator_spec.coffee`)

```
describe "Calculator", ->
    beforeEach ->
        @calculator = new Calculator()
    it "is not in scientific mode by default", ->
        expect(@calculator).not.toBeScientific()
    describe "scientific mode", ->
        beforeEach ->
            @calculator = new Calculator(true)
        it "is in scientific mode when set", ->
            expect(@calculator).toBeScientific()
    describe "#add", ->
        it "adds two numbers", ->
            expect(@calculator.add(1, 1)).toEqual 2
    describe "#subtract", ->
        it "subtracts two numbers", ->
            expect(@calculator.subtract(10, 1)).toEqual 9
    describe "#multiply", ->
        it "multiplies two numbers", ->
            expect(@calculator.multiply(5, 4)).toEqual 20
    describe "#divide", ->
        it "divides two numbers", ->
            expect(@calculator.divide(20, 5)).toEqual 4
```

Пример: (исходный файл: `calc.10/spec/javascripts/calculator_spec.js`)

```
(function() {
    describe("Calculator", function() {
        beforeEach(function() {
            return this.calculator = new Calculator();
        });
        it("is not in scientific mode by default", function() {
            return expect(this.calculator).not.toBeScientific();
        });
```

```
        describe("scientific mode", function() {
            beforeEach(function() {
                return this.calculator = new Calculator(true);
            });
            return it("is in scientific mode when set", function() {
                return expect(this.calculator).toBeScientific();
            });
        });
        describe("#add", function() {
            return it("adds two numbers", function() {
                return expect(this.calculator.add(1, 1)).toEqual(2);
            });
        });
        describe("#subtract", function() {
            return it("subtracts two numbers", function() {
                return expect(this.calculator.subtract(10, 1)).toEqual(9);
            });
        });
        describe("#multiply", function() {
            return it("multiplies two numbers", function() {
                return expect(this.calculator.multiply(5, 4)).toEqual(20);
            });
        });
        return describe("#divide", function() {
            return it("divides two numbers", function() {
                return expect(this.calculator.divide(20, 5)).toEqual(4);
            });
        });
    });
}).call(this);
```

Видите, насколько прозрачнее выглядят тесты? Метод сопоставления получился очень простым, но мы легко могли бы добавить в него еще больше логики, чтобы убрать еще несколько строк кода. Например, если бы класс `Calculator` имел графический интерфейс, в методе `toBeScientific` можно было бы проверить сам флаг режима и переключение изображения клавиатуры калькулятора в режим научных вычислений.

В заключение

Вот и все, наш короткий тур по инструменту тестирования Jasmine закончился. В этой главе мы немного поговорили о том, почему

тестирование играет важную роль и как соблюдение принципа разработки через тестирование может улучшить вашу жизнь. Надеюсь, я смог показать вам, что следовать этому принципу совсем не трудно и оно стоит того.

Мы коротко коснулись разных способов установки и настройки инструмента Jasmine, в частности, для нужд тестирования программного кода на языке CoffeeScript. После того, как мы подготовили Jasmine к работе, мы посмотрели, как выглядят испытательные тесты.

Затем мы определили свои тесты для класса `Calculator` и убедились, что все они проваливаются из-за отсутствия реализации класса. После того, как мы реализовали класс, мы убедились, что все тесты проходят успешно.

Наконец, мы выполнили несколько итераций разработки тестов и узнали, как пользоваться функцией `beforeEach` и как определять собственные методы сопоставления, чтобы обеспечить большую выразительность тестов.

Надеюсь вам понравилось это короткое путешествие по инструменту Jasmine. В Интернете можно найти множество сторонних библиотек, позволяющих писать более сложные тесты, включая проверку элементов графического интерфейса. Быстрый поиск на сайте GitHub [12] поможет вам отыскать замечательные библиотеки для использования в своих тестах.

И последнее: я хочу, чтобы прямо сейчас вы поклялись, что будете писать тесты для *всего* вашего программного кода, независимо от того, на каком языке он будет написан: CoffeeScript, JavaScript, Java, ColdFusion или Cobalt. Поднимите правую руку и произнесите следующую клятву:

```
"
    Торжественно клянусь тестировать весь свой программный код.
    Я буду тестировать весь код, а не только отдельные его части.
    Я понимаю, что отказ от тестирования
    вынудит Марка отыскать и отшлепать меня моими же тапками.
    Я делаю это не только для себя,
    но для всех разработчиков, кому придется работать с моим кодом.
    Я также обязуюсь привести других разработчиков к этой присяге.
    Если они откажутся, я сообщу Марку
    и он отшлепает их, их же тапками.
    Да здравствует тестирование!
"
```

Поздравляю! Теперь идите вперед и тестируйте!

Примечания

1. http://www.metabates.com/2010/07/01/testing-is-not-an-option/.
2. http://ru.wikipedia.org/wiki/Разработка_через_тестирование.
3. http://www.metabates.com/2010/10/12/how-to-become-a-test-driven-developer/.
4. http://pivotal.github.com/jasmine/.
5. https://github.com/rspec/rspec.
6. http://docs.jquery.com/QUnit.
7. http://code.google.com/p/js-test-driver/.
8. http://yuilibrary.com/yui/docs/test/.
9. http://johnbintz.github.com/jasmine-headless-webkit/.
10. http://nodejs.org.
11. https://github.com/pivotal/jasmine/wiki/Matchers.
12. http://github.com.

9. Введение в Node.js

Год тому назад или что-то около того в мир веб-разработки вихрем ворвалась платформа, разработанная компанией Joyent [1]. Эта платформа называется *Node.js*, [2] но чаще ее называют просто Node (и далее в книге будет использоваться это, более краткое название).

> **Совет.** Если кому-то не верится, что платформа Node «вихрем ворвалась в мир веб-разработки», можете поинтересоваться этим вопросом в таких компаниях, как LinkedIn, [3] которая в 2011 году объявила, что полностью переписала свой API с Ruby on Rails [4] на Node. А это очень веское подтверждение.

Так что же такое платформа Node, и почему разговор о ней зашел в книге, посвященной языку CoffeeScript? Об этом вы узнаете в данной главе.

Что такое Node.js?

Node – это реализация языка JavaScript для использования на стороне сервера, основанная на движке JavaScript V8 [5], разработанном в компании Google. Node одновременно является фреймворком, средой выполнения и языком. По этой причине многие иногда затрудняются дать точное определение платформе Node и объяснить, где и почему ее следует использовать.

Основное назначение платформы Node – помочь разработчикам в создании управляемых событиями, асинхронных приложений. В платформе Node все запросы обрабатываются асинхронно, и практически все операции ввода/вывода выполняются в неблокирующем режиме. По этой причине приложения на платформе Node получаются весьма эффективными, в терминах использования памяти, и способны параллельно обрабатывать огромное количество соединений.

Знакомство с платформой Node.js было добавлено в книгу о языке CoffeeScript по трем причинам. Во-первых, приложения для

платформы Node пишутся на JavaScript, поэтому те же самые приложения можно писать на CoffeeScript и пользоваться всеми преимуществами этого языка. Во-вторых, команда `coffee`, повсеместно используемая в этой книге, творит свое волшебство с помощью Node, поэтому CoffeeScript и Node уже являются родственниками. В-третьих, в состав платформы Node входит система управления пакетами *NPM*. [6] Она позволяет разработчикам упаковывать и распространять свои модули, которые затем могут импортироваться другими приложениями для Node. Эти модули могут пригодиться даже вне приложений для Node. Например, их можно импортировать в файлах Cakefile и с их помощью реализовывать некоторые задания.

В этой главе будет создан простой веб-сервер для обслуживания статических файлов в каталоге. Он также будет компилировать ресурсы на языке CoffeeScript в масштабе реального времени и возвращать скомпилированный код на JavaScript браузеру. И, разумеется, этот сервер будет написан на CoffeeScript.

Установка Node

Вы уже наверняка знаете о моей склонности посылать за инструкциями по установке на сайт производителя. Поэтому, чтобы узнать, как установить платформу Node в своей операционной системе, я рекомендую обратиться на страницу установки Node: http://nodejs.org/#download. Открою вам небольшой секрет: если в процессе чтения этой книги вы опробовали примеры программного кода, велика вероятность, что вы уже установили платформу Node. Но даже в этом случае у вас есть шанс обновить платформу до последней и лучшей версии.

После установки Node попробуем покрутить ее немного. С помощью следующей команды можно запустить *интерактивную консоль*:

```
> node
```

Затем в консоли Node можно выполнить любой код на JavaScript:

```
node> 1 + 1
```

В дополнение к интерактивной консоли Node имеется возможность выполнять файлы с программным кодом на JavaScript, почти

так же, как мы делали это с файлами CoffeeScript. Возьмем для примера следующий файл на JavaScript:

Пример: (исходный файл: `example.js`)

```
(function() {
    var sayHi;
    sayHi = function(name) {
        if (name == null) name = 'World';
        return console.log("Hello, " + name);
    };
    sayHi('Mark');
}).call(this);
```

Этот файл можно запустить командой:

```
> node example.js
```

И получить в результате такой вывод:

Вывод: (исходный файл: `example.js`)

```
Hello, Mark
```

Введение

Теперь, когда платформа Node установлена и мы знаем, как запускать файлы JavaScript, напишем небольшой *сервер «Hello, World»*. Сначала я покажу программный код с реализацией сервера, а потом расскажу о нем подробнее.

Пример: (исходный файл: `server.1/server.coffee`)

```
http = require('http')
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    data = "Hello World!"
    res.writeHead 200,
        "Content-Type": "text/plain"
        "Content-Length": Buffer.byteLength(data, "utf-8")
    res.write(data, "utf-8")
    res.end()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Пример: (исходный файл: server.1/server.js)

```
(function() {
    var http, ip, port, server;
    http = require('http');
    port = 3000;
    ip = "127.0.0.1";
    server = http.createServer(function(req, res) {
        var data;
        data = "Hello World!";
        res.writeHead(200, {
            "Content-Type": "text/plain",
            "Content-Length": Buffer.byteLength(data, "utf-8")
        });
        res.write(data, "utf-8");
        return res.end();
    });
    server.listen(port, ip);
    console.log("Server running at http://" + ip + ":" + port + "/");
}).call(this);
```

Прежде чем приступать к исследованию программного кода, запустим наш маленький сервер, чтобы посмотреть на него в действии:

```
> coffee server.1/server.coffee
```

Если теперь открыть в браузере страницу http://127.0.0.1:3000, должно появиться приветствие.

Итак, начнем с самого очевидного вопроса: «Почему файл запускался командой coffee, а не командой node, как было показано выше?». Причина в том, что команда node не знает, как скомпилировать и выполнить программный код на CoffeeScript. Если попробовать запустить файл CoffeeScript с помощью команды node, она взорвалась бы самыми невероятными ошибками. К счастью, команда coffee реализована поверх платформы Node, поэтому, чтобы получить желаемый результат, следует действовать, как было показано выше.

> **Совет.** При желании можно было бы предварительно скомпилировать файл CoffeeScript в файл JavaScript и затем выполнить его с помощью команды node, но зачем этот лишний труд, если можно просто воспользоваться командой coffee?

Исследуем наш сервер и посмотрим, как он действует.

Прежде всего, необходимо импортировать модуль `http` из платформы Node. Он содержит все необходимое для приема и обработки HTTP-запросов. Для этого используется функция `require`, предоставляемая платформой Node. Затем создаются несколько переменных, представляющих порт и IP-адрес, которые будут прослушиваться сервером.

Далее следует основная реализация сервера, использующая функцию с говорящим именем `createServer` из модуля `http`. Функции `createServer` передается функция обратного вызова, которая будет вызываться всякий раз, при поступлении нового запроса. Самое интересное, что наша функция обратного вызова вызывается только когда поступает новый запрос, все остальное время сервер простаивает и почти не использует вычислительные ресурсы компьютера. Оставим на минутку функцию обратного вызова и посмотрим на остальную часть файла. Все, что остается сделать – сообщить серверу порт и IP-адрес, где он должен принимать запросы.

Функция обратного вызова ожидает получить два аргумента. Первый аргумент представляет запрос, а второй – ответ, который будет отправлен обратно. Объект запроса, аргумент `req`, содержит огромный объем данных, которые могут пригодиться, – запрошенная страница, тип браузера, строка запроса и так далее.

> **Совет.** Я советую добавить в функцию обратного вызова вывод содержимого запроса с помощью функции `console.log`, чтобы иметь возможность видеть на экране, какие данные содержатся в нем. Это самый простой способ узнать, какая информация доступна в каждом запросе.

Когда запрос поступает серверу, он вызывает нашу функцию обратного вызова, передавая ей соответствующий запрос и объект, который можно использовать для создания ответа. Первое, что мы сделали, – определили переменную с именем `data` для хранения ответа. В данном примере в качестве ответа используется простая строка «Hello World!».

Теперь, когда мы определились, что сказать, можно приступать к отправке ответа клиенту. Сначала необходимо отправить код ответа и заголовки – делается это с помощью функции `writeHead`. В первом аргументе ей передается код HTTP-ответа. В данном случае – код 200, который сообщает, что все идет хорошо. Во втором

аргументе передается объект, представляющий HTTP-заголовки, отправляемые обратно. В данном случае отправляются два заголовка. Первый заголовок, `Content-Type`, определяет тип ответа и имеет значение `text/plain`, потому что ответ не содержит ни двоичных данных, ни разметки HTML. Второй заголовок, который должен быть отправлен обратно, – `Content-Length`. Он сообщает клиенту размер ответа в байтах. Платформа Node предоставляет замечательную вспомогательную утилиту `Buffer.byteLength`. Достаточно передать ей данные, объем которых требуется узнать, и она позаботится обо всем остальном.

После отправки клиенту кода и заголовков, можно отправить фактические данные, воспользовавшись функцией `write`.

После отправки данных в заключение следует вызвать функцию `end`, которая делает именно то, что следует из ее имени – она завершает отправку ответа. Итак, мы создали наш первый сервер на платформе Node. Поздравляю!

Потоковые ответы

При создании приложений для платформы Node важно помнить, что для отправки ответа функция `write` может вызываться неоднократно. Ее можно вызывать столько раз, сколько потребуется, перед заключительным вызовом функции `end`. Это отличный способ реализовать *потоковый API* для своего приложения. Посмотрим, как это можно сделать.

Напишем простой сервер, выводящий классическую строку из фильма «The Shining» («Сияние») [7]: «All work and no play makes Jack a dull boy» («Сплошная работа и отсутствие развлечений превращают Джека в зануду»). Передача этой строки браузеру будет повторяться в течение 30 секунд и затем прекращаться.

Пример: (исходный файл: `streaming/server.coffee`)

```
http = require('http')
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    res.writeHead 200,
        "Content-Type": "text/plain"
    setInterval ->
        res.write("All work and no play makes Jack a dull boy. ")
    , 10
```

```
    setTimeout ->
        res.end()
    , 30000
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Пример: (исходный файл: `streaming/server.js`)

```
(function() {
    var http, ip, port, server;
    http = require('http');
    port = 3000;
    ip = "127.0.0.1";
    server = http.createServer(function(req, res) {
        res.writeHead(200, {
            "Content-Type": "text/plain"
        });
        setInterval(function() {
           return res.write("All work and no play makes Jack a dull boy. ");
        }, 10);
        return setTimeout(function() {
            return res.end();
        }, 30000);
    });
    server.listen(port, ip);
    console.log("Server running at http://" + ip + ":" + port + "/");
}).call(this);
> coffee streaming/server.coffee
```

Если теперь открыть в браузере страницу http://127.0.0.1:3000, вы должны увидеть, как вывод этого жуткого сообщения несколько раз повторяется в течение 30 секунд и затем прекращается. Если бы наше приложение не было потоковым, мы в течение 30 секунд смотрели бы на пустой экран, задаваясь вопросом: «работает ли оно?», – а затем увидели бы мгновенное появление всех строк одновременно.

Внутри функции обратного вызова выполняется цикл `setInterval`, который отправляет сообщение каждые 10 миллисекунд. Далее следует блок `setTimeout`, завершающий отправку ответа через 30 секунд.

С помощью платформы Node совсем несложно реализовать потоковый сервер. Если хотите увидеть нечто по-настоящему волнующее, откройте несколько окон браузера и обратитесь в них к нашему

серверу. Вы увидите, что платформа Node с легкостью обслуживает все потоковые запросы, не требуя от нас добавлять в реализацию сервера что-то экстраординарное.

Создание сервера CoffeeScript

Создадим сервер, который будет полезен нам – *сервер приложений*. Требования к серверу будут очень просты. Он должен просматривать входящие запросы и выполнять одну из трех операций. Если клиент запрашивает файл JavaScript, сервер должен отыскать соответствующий файл CoffeeScript в каталоге `src`, скомпилировать его в файл JavaScript и вернуть результат компиляции клиенту. Если запрошен файл, не являющийся файлом JavaScript, сервер должен отыскать его в каталоге `public` и отправить клиенту. Наконец, если искомый файл отсутствует в каталоге `src` или `public`, сервер должен вернуть ответ с кодом `404`.

Теперь мы знаем, что будем создавать, но прежде чем приступить к делу, хочу предупредить, что программный код, который будет создан в этом разделе, имеет довольно большой объем, и будет повторяться неоднократно. По этой причине скомпилированный программный код на JavaScript будет показан только после окончания разработки сервера. Будьте уверены, вы еще поблагодарите меня за это. А теперь перейдем к разработке.

Прежде всего, я знаю, что программный код получится слишком объемным, чтобы уместить его в одной функции обратного вызова. Поэтому я помещу его в класс и буду работать с этим классом. Функцию обратного вызова для сервера мы оформим в виде класса, выполняющего все необходимые операции, поэтому нам сначала необходимо получить представление о том, как будет выглядеть этот класс.

Пример: (исходный файл: `server.2/server.coffee`)

```
http = require('http')
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Совет. Отказавшись от идеи поместить весь программный код в единственную функцию обратного вызова, мы получаем еще одно большое преимущество – простота тестирования. Если создать пару классов для обработки запросов, мы легко сможем использовать знания, полученные в главе 8, «Тестирование с помощью Jasmine», для тестирования обработки всех типов файлов, чтобы убедиться в безупречной работе, без необходимости запускать для этого сервер.

Если попытаться запустить этот программный код, он завершится с ошибкой, потому что у нас пока нет класса `Application`, поэтому напишем заготовку класса, опираясь на программный код, который только что написали:

Пример: (исходный файл: `server.3/server.coffee`)

```
http = require('http')
class Application
    constructor: (@req, @res) ->
    process: ->

port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Получилась отличная заготовка для класса `Application`. Теперь нарастим чуть больше плоти на его кости.

Пример: (исходный файл: `server.4/server.coffee`)

```
http = require('http')
url = require('url')
class Application
    constructor: (@req, @res) ->
        @pathInfo = url.parse(@req.url, true)
    process: ->
        if /^\/javascripts\//.test @pathInfo.pathname
            new JavaScriptProcessor(@req, @res, @pathInfo).process()
        else
            new PublicProcessor(@req, @res, @pathInfo).process()
port = 3000
ip = "127.0.0.1"
```

```
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Прежде всего, необходимо импортировать модуль `url`, входящий в состав платформы Node. Он содержит несколько вспомогательных функций, с помощью которых можно разбить строку URL на составляющие. Именно это и делается в конструкторе класса `Application`.

В функции `process` начинает появляться нечто по-настоящему интересное. Сначала необходимо определить тип запроса, а затем обработать его соответствующим образом. Из наших требований известно, что сервер будет обрабатывать запросы двух типов: файлы с программным кодом на JavaScript и все остальное. Поэтому создадим еще два класса. Один будет обрабатывать все файлы JavaScript, а другой – файлы из каталога `public`. Определить тип запроса нам поможет очень простое регулярное выражение. Затем можно обратиться к соответствующему классу для обработки запроса данного типа.

> **Совет.** Следует заметить, что если бы это было действующее приложение, я рекомендовал бы поместить определения классов в отдельные файлы, но в нашем учебном примере мы не будем усложнять себе жизнь.

Далее, создадим заготовки функций, которые потребуются для обработки файлов разных типов. Можно с достаточной степенью уверенности предположить, что оба класса, `JavascriptProcessor` и `PublicProcessor`, будут иметь общую функциональность, поэтому наверняка лучше будет создать родительский класс, а затем наследовать его в специализированных классах, чтобы обеспечить совместное использование общей функциональности. Учитывая сказанное выше, создадим заготовку родительского класса `Processor`, а также заготовки классов `JavascriptProcessor` и `PublicProcessor`:

Пример: (исходный файл: `server.5/server.coffee`)

```
http = require('http')
url = require('url')
class Application
    constructor: (@req, @res) ->
        @pathInfo = url.parse(@req.url, true)
    process: ->
```

```
        if /^\/javascripts\//.test @pathInfo.pathname
            new JavaScriptProcessor(@req, @res, @pathInfo).process()
        else
            new PublicProcessor(@req, @res, @pathInfo).process()
class Processor
    constructor: (@req, @res, @pathInfo) ->
    contentType: ->
        throw new Error("must be implemented!")
    process: ->
        throw new Error("must be implemented!")
    pathname: ->
    write: (data, status = 200, headers = {}) ->
class JavaScriptProcessor extends Processor
    contentType: ->
    process: ->
class PublicProcessor extends Processor
    contentType: ->
    process: ->
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Рассмотрим сначала класс `Processor`, он содержит пять функций, которые нам потребуются. Первая – функция `constructor`. Она принимает запрос, ответ и объект, представляющий информацию о пути к файлу.

Следующие две функции, `contentType` и `process`, должны быть реализованы в дочерних классах. Функция `contentType` будет возвращать соответствующий тип файла для включения в заголовки при отправке ответа. Функция `process` будет выполнять всю тяжелую работу по чтению файлов из каталога `public` или компиляции файлов с программный кодом на CoffeeScript.

Следующий метод, `pathname`, будет использоваться для определения имени файла на диске, который требуется отыскать. Дочерние классы не обязательно должны будут переопределять этот метод, потому что, мы предусмотрели получение простого значения по умолчанию, соответствующего пути в запросе, в чем вы убедитесь через несколько минут.

Последний метод, write, будет выполнять все операции, необходимые для отправки кода ответа, заголовков и тела ответа. Как видно в примере выше, по умолчанию возвращается код ответа 200 и пустой объект headers с заголовками. Через несколько минут мы определим достаточно разумные значения по умолчанию для этого объекта headers, когда будем наполнять этот метод.

Прежде чем перейти к классам JavascriptProcessor и PublicProcessor, закончим реализацию родительского класса Processor.

Пример: (исходный файл: server.6/server.coffee)

```
http = require('http')
url = require('url')
class Application
    constructor: (@req, @res) ->
        @pathInfo = url.parse(@req.url, true)
    process: ->
        if /^\/javascripts\//.test @pathInfo.pathname
            new JavaScriptProcessor(@req, @res, @pathInfo).process()
        else
            new PublicProcessor(@req, @res, @pathInfo).process()
class Processor
    constructor: (@req, @res, @pathInfo) ->
    contentType: ->
        throw new Error("must be implemented!")
    process: ->
        throw new Error("must be implemented!")
    pathname: ->
        @pathInfo.pathname
    write: (data, status = 200, headers = {}) ->
        headers["Content-Type"] ||= @contentType()
        headers["Content-Length"] ||= Buffer.byteLength(data, "utf-8")
        @res.writeHead(status, headers)
        @res.write(data, "utf-8")
        @res.end()
class JavaScriptProcessor extends Processor
    contentType: ->
    process: ->
class PublicProcessor extends Processor
    contentType: ->
    process: ->
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
```

```
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Здесь мы добавили возврат значения по умолчанию из функции pathname, соответствующего пути в запросе клиента. Реализация функции write так же оказалась достаточно прямолинейной и не слишком отличается от того, о чем говорилось выше в этой главе.

Теперь перейдем к реализации наиболее интересного из двух классов-обработчиков, класса JavascriptProcessor. В реализации этого класса необходимо переопределить три функции родительского класса и подключить еще два модуля из платформы Node. Взгляните:

Пример: (исходный файл: server.7/server.coffee)

```
http = require('http')
url = require('url')
fs = require('fs')
CoffeeScript = require('coffee-script')
class Application
    constructor: (@req, @res) ->
        @pathInfo = url.parse(@req.url, true)
    process: ->
        if /^\/javascripts\//.test @pathInfo.pathname
            new JavaScriptProcessor(@req, @res, @pathInfo).process()
        else
            new PublicProcessor(@req, @res, @pathInfo).process()
class Processor
    constructor: (@req, @res, @pathInfo) ->
    contentType: ->
        throw new Error("must be implemented!")
    process: ->
        throw new Error("must be implemented!")
    pathname: ->
        @pathInfo.pathname
    write: (data, status = 200, headers = {}) ->
        headers["Content-Type"] ||= @contentType()
        headers["Content-Length"] ||= Buffer.byteLength(data, "utf-8")
        @res.writeHead(status, headers)
        @res.write(data, "utf-8")
        @res.end()
class JavaScriptProcessor extends Processor
```

```
    contentType: ->
        "application/x-javascript"
    pathname: ->
        file = (/\/javascripts\/(.+)\.js/.exec(@pathInfo.pathname))[1]
        return "#{file}.coffee"
    process: ->
        fs.readFile "src/#{@pathname()}", "utf-8", (err, data) =>
            if err?
                @write("", 404)
            else
                @write(CoffeeScript.compile(data))
class PublicProcessor extends Processor
    contentType: ->
    process: ->
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Сначала необходимо подключить модуль `fs`. Этот модуль с невыразительным именем содержит функции доступа к файловой системе. Доступ к файловой системе необходим, чтобы можно было прочитать файлы с программным кодом на CoffeeScript перед компиляцией. Позднее, когда мы перейдем к реализации класса `PublicProcessor`, нам также потребуется читать файлы из каталога `public`.

Затем мы подключили модуль `coffee-script`. Я думаю, вы уже догадались, зачем он нужен, не так ли? Этот модуль потребуется нам для компиляции файлов с исходным программным кодом на CoffeeScript в программный код на JavaScript.

> **Совет.** Модуль `coffee-script` содержит не только функцию `compile`, но и многое другое. Я настоятельно рекомендую заглянуть в этот модуль и посмотреть, что еще он может предложить.

Подключив необходимые модули, можно приступать к реализации функций в классе `JavascriptProcessor`. Реализация функции `contentType` получилась очень простой, потому что известно, что программный код на JavaScript имеет тип содержимого `application/x-javascript`.

Совет. В некоторых версиях Internet Explorer могут возникать проблемы с данным типом содержимого. Попробуйте изменить тип на text/javascript, это должно помочь.

Реализация функции `pathname` выглядит не так просто, как реализация функции `contentType`, но и в ней нет ничего особенно сложного. Чтобы отбросить путь к запрошенному файлу `javascripts/` и расширение `.js`, в ней используется регулярное выражение. Например, если запросить файл `javascripts/foo.js`, регулярное выражение оставит только имя `foo`. После этого остается лишь вернуть полученное имя файла, добавив к нему расширение `.coffee`.

Наконец, необходимо реализовать метод `process`. Я считаю, что реализация этой функции до определенной степени выглядит более просто и понятно, чем реализация только что реализованной функции `pathname`. С помощью функции `readFile` из модуля `fs` мы пытаемся прочитать файл с программным кодом на CoffeeScript из каталога `src`. Затем функция `readFile` вызывает нашу функцию обратного вызова, переданную ей.

Эта функция обратного вызова принимает два аргумента. Первый – объект, представляющий ошибки, которые могли возникнуть при чтении файла, такие как отсутствие файла. Второй аргумент – фактические данные, прочитанные из файла.

Внутри функции обратного вызова, в случае ошибки, наличие которой проверяется с помощью оператора существования `?`, описанном в главе 3, «Управляющие конструкции», вызывается функция `write` родительского класса и ей передается пустое тело ответа и код 404. Если ошибки не были обнаружены, вызывается функция `compile` из модуля `coffee-script` и ей передается содержимое, прочитанное из файла на диске. Эта функция возвращает скомпилированный программный код на JavaScript. Затем скомпилированный код передается функции `write` для отправки обратно клиенту. Весьма изящно, правда?

Все, что осталось сделать, – реализовать те же три функции в классе `PublicProcessor` и наш сервер будет готов.

Пример: (исходный файл: `final/server.coffee`)

```
http = require('http')
url = require('url')
fs = require('fs')
CoffeeScript = require('coffee-script')
```

```
class Application
    constructor: (@req, @res) ->
        @pathInfo = url.parse(@req.url, true)
    process: ->
        if /^\/javascripts\//.test @pathInfo.pathname
            new JavaScriptProcessor(@req, @res, @pathInfo).process()
        else
            new PublicProcessor(@req, @res, @pathInfo).process()
class Processor
    constructor: (@req, @res, @pathInfo) ->
    contentType: ->
        throw new Error("must be implemented!")
    process: ->
        throw new Error("must be implemented!")
    pathname: ->
        @pathInfo.pathname
    write: (data, status = 200, headers = {}) ->
        headers["Content-Type"] ||= @contentType()
        headers["Content-Length"] ||= Buffer.byteLength(data, "utf-8")
        @res.writeHead(status, headers)
        @res.write(data, "utf-8")
        @res.end()
class JavaScriptProcessor extends Processor
    contentType: ->
        "application/x-javascript"
    pathname: ->
        file = (/\/javascripts\/(.+)\.js/.exec(@pathInfo.pathname))[1]
        return "#{file}.coffee"
    process: ->
        fs.readFile "src/#{@pathname()}", "utf-8", (err, data) =>
            if err?
                @write("", 404)
            else
                @write(CoffeeScript.compile(data))
class PublicProcessor extends Processor
    contentType: ->
        ext = (/\.(.+)$/.exec(@pathname()))[1].toLowerCase()
        switch ext
            when "png", "jpg", "jpeg", "gif"
                "image/#{ext}"
            when "css"
                "text/css"
            else
                "text/html"
```

```
    process: ->
        fs.readFile "public/#{@pathname()}", "utf-8", (err, data) =>
            if err?
                @write("Oops! We couldn't find the page you were looking
↪for.", 404)
            else
                @write(data)
    pathname: ->
        unless @_pathname
            if @pathInfo.pathname is "/" or @pathInfo.pathname is ""
                @pathInfo.pathname = "index"
            unless /\..+$/.test @pathInfo.pathname
                @pathInfo.pathname += ".html"
            @_pathname = @pathInfo.pathname
        return @_pathname
port = 3000
ip = "127.0.0.1"
server = http.createServer (req, res) ->
    app = new Application(req, res)
    app.process()
server.listen(port, ip)
console.log "Server running at http://#{ip}:#{port}/"
```

Пример: (исходный файл: `final/server.js`)

```
(function() {
    var Application, CoffeeScript, JavaScriptProcessor, Processor,
        PublicProcessor, fs, http, ip, port, server, url,
        __hasProp = Object.prototype.hasOwnProperty,
        __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function ctor()
{ this.constructor = child; } ctor.prototype = parent.prototype;
child.prototype = new ctor; child.__super__ = parent.prototype; return
child; };
    http = require('http');
    url = require('url');
    fs = require('fs');
    CoffeeScript = require('coffee-script');
    Application = (function() {
        function Application(req, res) {
            this.req = req;
            this.res = res;
            this.pathInfo = url.parse(this.req.url, true);
        }
```

```
        Application.prototype.process = function() {
            if (/^\/javascripts\//.test(this.pathInfo.pathname)) {
                return new JavaScriptProcessor(this.req, this.res,
↪this.pathInfo).process();
            } else {
                return new PublicProcessor(this.req, this.res,
↪this.pathInfo).process();
            }
        };
        return Application;
    })();
    Processor = (function() {
        function Processor(req, res, pathInfo) {
            this.req = req;
            this.res = res;
            this.pathInfo = pathInfo;
        }
        Processor.prototype.contentType = function() {
            throw new Error("must be implemented!");
        };
        Processor.prototype.process = function() {
            throw new Error("must be implemented!");
        };
        Processor.prototype.pathname = function() {
            return this.pathInfo.pathname;
        };
        Processor.prototype.write = function(data, status, headers) {
            if (status == null) status = 200;
            if (headers == null) headers = {};
            headers["Content-Type"] || (headers["Content-Type"] =
↪this.contentType());
            headers["Content-Length"] || (headers["Content-Length"] =
↪Buffer.byteLength(data, "utf-8"));
            this.res.writeHead(status, headers);
            this.res.write(data, "utf-8");
            return this.res.end();
        };
        return Processor;
    })();
    JavaScriptProcessor = (function(_super) {
        __extends(JavaScriptProcessor, _super);
        function JavaScriptProcessor() {
            JavaScriptProcessor.__super__.constructor.apply(this,
                                                            arguments);
```

```
        }
        JavaScriptProcessor.prototype.contentType = function() {
            return "application/x-javascript";
        };
        JavaScriptProcessor.prototype.pathname = function() {
            var file;
            file = (/\/javascripts\/(.+)\.js/.exec(this.pathInfo.pathname))
↪[1];
            return "" + file + ".coffee";
        };
        JavaScriptProcessor.prototype.process = function() {
            var _this = this;
            return fs.readFile("src/" + (this.pathname()), "utf-8",
↪function(err, data) {
                if (err != null) {
                    return _this.write("", 404);
                } else {
                    return _this.write(CoffeeScript.compile(data));
                }
            });
        };
        return JavaScriptProcessor;
    })(Processor);
    PublicProcessor = (function(_super) {
        __extends(PublicProcessor, _super);
        function PublicProcessor() {
            PublicProcessor.__super__.constructor.apply(this, arguments);
        }
        PublicProcessor.prototype.contentType = function() {
            var ext;
            ext = (/\.(.+)$/.exec(this.pathname()))[1].toLowerCase();
            switch (ext) {
                case "png":
                case "jpg":
                case "jpeg":
                case "gif":
                    return "image/" + ext;
                case "css":
                    return "text/css";
                default:
                    return "text/html";
            }
        };
        PublicProcessor.prototype.process = function() {
```

```
            var _this = this;
            return fs.readFile("public/" + (this.pathname()), "utf-8",
↪function(err, data) {
                if (err != null) {
                    return _this.write("Oops! We couldn't find the page you
↪were looking for.", 404);
                } else {
                    return _this.write(data);
                }
            });
        };
        PublicProcessor.prototype.pathname = function() {
            if (!this._pathname) {
                if (this.pathInfo.pathname === "/" || this.pathInfo.pathname
↪=== "") {
                    this.pathInfo.pathname = "index";
                }
                if (!/\..+$/.test(this.pathInfo.pathname)) {
                    this.pathInfo.pathname += ".html";
                }
                this._pathname = this.pathInfo.pathname;
            }
            return this._pathname;
        };
        return PublicProcessor;
    })(Processor);
    port = 3000;
    ip = "127.0.0.1";
    server = http.createServer(function(req, res) {
        var app;
        app = new Application(req, res);
        return app.process();
    });
    server.listen(port, ip);
    console.log("Server running at http://" + ip + ":" + port + "/");
}).call(this);
```

Реализация функции `process` в классе `PublicProcessor` мало чем отличается от реализации функции `process` в классе `JavascriptProcessor`. Два основных отличия заключаются в том, что в случае ошибки вместо пустой строки она отправляет сообщение, и в ней не требуется компилировать данные перед отправкой.

Метод `contentType` получился немного более замысловатым, потому что ему приходится обрабатывать HTML-файлы и изображения

некоторых типов. Сначала, с помощью регулярного выражения, он получает расширение запрошенного файла. Затем с помощью инструкции `switch`, представленной в главе 3, по расширению файла определяет его тип.

> **Совет.** Хочу заметить, если кому-то непонятно, что данный сервер не предназначен для использования в действующих приложениях. Он создавался исключительно как учебный пример. В его реализации имеется множество недостатков, о которых я умолчал, таких как ненадлежащая обработка ошибок и ненадежный способ определения типов файлов. Не надо слать мне гневные письма о своих неудачных попытках использовать этот сервер в работе.

На этом реализация функции `pathname` закончена. Как это ни странно, но данная функция является едва ли не самой сложной в примере, потому что в ней пришлось предусмотреть обработку трех ситуаций. Первая – обработка полностью сформированных имен файлов, таких как `index.html` или `images/foo.png`. Это обычные файлы и должны возвращаться клиенту в неизменном виде.

Второй случай – имена файлов без расширений, такие как `index` или `users/1`. Поскольку в таких именах отсутствуют расширения, к каждому из них необходимо добавить расширение по умолчанию `.html`. В результате они превращаются в имена `index.html` и `users/1.html`.

Последний случай, путь к файлу по умолчанию /. Его необходимо преобразовать в `index.html`, чтобы потом его можно было отыскать в каталоге `public`.

Обратите внимание, что все эти результаты сохраняются в переменной `@_pathname`, благодаря чему эту работу достаточно выполнить всего один раз, при первом вызове функции.

Опробование сервера

Что еще нужно сделать, чтобы запустить сервер? Сначала необходимо создать каталог `src` с единственным файлом `application.coffee`, содержащим следующий код:

Пример: (исходный файл: `final/src/application.coffee`)

```
$ ->
    $("body").html("Hello from jQuery and Node!!")
```

Пример: (исходный файл: `final/src/application.js`)

```
(function() {
    $(function() {
        return $("body").html("Hello from jQuery and Node!!");
    });
}).call(this);
```

Для тех, кто не знаком с библиотекой jQuery [8], скажу, что этот программный код замещает содержимое тега `body` в HTML-файле строкой «Hello from jQuery and Node!!» после загрузки дерева DOM [9].

Теперь добавьте в каталог `public` файл `index.html`:

Пример: (исходный файл: `final/public/index.html`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Node.js</title>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
↪jquery.min.js" type="text/javascript"></script>
        <script src="/javascripts/application.js"
↪type="text/javascript"></script>
    </head>
    <body>
        Hello from Node!!
    </body>
</html>
```

Этот HTML-файл очень прост. В заголовке он загружает два JavaScript-файла. Первый – библиотека jQuery, а второй – файл `/javascripts/application.js`, который должен отобразиться в хранящийся на нашем сервере файл `src/application.coffee`, если все пойдет как надо. В теле HTML-файла выводится строка «Hello from Node!!».

Запустите сервер и откройте в браузере страницу http://127.0.0.1:3000.

```
> coffee finale/server.coffee
```

Вы должны увидеть приветствие «Hello from jQuery and Node!!», вместо «Hello from Node!!».

В заключение

Здесь мы совершили молниеносный тур по платформе Node.js. Мы узнали, что такое платформа Node и познакомились с основными ее возможностями. Мы также узнали насколько просто с помощью Node реализовать потоковый сервер.

Кроме того, мы создали забавный сервер, автоматически компилирующий программный код на CoffeeScript.

Node.js – замечательная платформа, позволяющая создавать масштабируемые, управляемые событиями приложения. С ее помощью можно создавать не только HTTP-серверы. Кандидатами могут быть любые приложения, выполняющие операции с сокетами, от Telnet-серверов до приложений обмена мгновенными сообщениями. На этой платформе можно также реализовать потоковую парадигму, обратную той, что описывалась выше. Почему бы вместо потоковой передачи данных *от сервера клиенту* не организовать аналогичную передачу данных *от клиента серверу*? Представьте себе реализацию обработки файлов в процессе их *выгрузки на сервер*! Это вполне возможно с помощью Node.js.

В Интернете можно найти множество подробных описаний платформы Node и демонстрационных роликов, и я настоятельно рекомендую ознакомиться с ними, чтобы побольше узнать о возможностях этой платформы.

Примечания

1. http://www.joyentcloud.com/.
2. http://nodejs.org/.
3. http://www.linkedin.com.
4. http://www.rubyonrails.org.
5. http://code.google.com/p/v8/.
6. http://npmjs.org/.
7. http://www.imdb.com/title/tt0081505/[1].
8. http://jquery.com/.
9. http://ru.wikipedia.org/wiki/Document_Object_Model.

[1] Описание фильма в Википедии: http://ru.wikipedia.org/wiki/Сияние_(фильм). – *Прим. перев.*

10. Пример: список задач, часть 1 (серверная)

В первой части книги мы познакомились с достоинствами и недостатками языка CoffeeScript. Мы охватили все основные его особенности: как он работает, как отображается в JavaScript и как может помочь сделать наши приложения более ясными и надежными. В предыдущих нескольких главах мы рассмотрели несколько проектов, включенных в экосистему CoffeeScript. Теперь, в последних трех главах, я хотел бы продемонстрировать практическое применение полученных знаний.

Лично мне проще учиться, когда я наблюдаю за предметом изучения в действии. Знакомство с надуманными примерами использования тех или иных возможностей библиотеки позволяет лишь узнать, что эта библиотека может предложить, а наблюдение за ней в действии – совсем другое дело. Практические примеры позволяют понять, как эта библиотека впишется в мою повседневную работу. Проще говоря, она оживает для меня. Это как раз и есть то, что мы будем делать в последних нескольких главах.

Приложение, которое будет создано в этих главах, является классическим *приложением списка задач*. Всем нам приходилось писать их, и все мы видели примеры этих приложений в действии. То есть, все мы знаем, что такое приложение списка задач и как оно действует. Это достаточное большое приложение, в том смысле, что на его примере можно продемонстрировать применение библиотек и инструментов, но не настолько большое, чтобы его нельзя было написать за относительно короткий промежуток времени.

В этой главе мы создадим серверную часть приложения. Для этого нам потребуется сервер, который будет обслуживать страницу со списком задач, файлы ресурсов и самое важное – сам список задач, а также обеспечивать возможность сохранения этого списка в постоянном хранилище данных.

В двух последующих главах обсуждение темы будет продолжено с момента, где закончится данная глава. Там мы создадим пользовательский интерфейс приложения. С его помощью можно будет выполнять такие операции, как отображение списка задач на странице, создание новых задач, изменение существующих задач и их уничтожение.

В настоящий момент я считаю, что вы видели уже достаточно много программного кода на JavaScript, поэтому с данного момента я больше не буду приводить код на JavaScript, сгенерированный компилятором CoffeeScript. При желании увидеть этот код, вы можете скомпилировать его самостоятельно или посетить страницу этой книги на сайте GitHub. [1]

Установка и настройка фреймворка Express

Первым пунктом на пути к созданию лучшего в мире приложения списка задач является получение веб-сервера/веб-фреймворка, который будет обрабатывать наши запросы. Для своих нужд мы будем использовать самый популярный фреймворк на платформе Node – Express. [2] *Фреймворк Express* прост в установке, его можно очень быстро настроить и запустить. К тому же его окружает активное сообщество разработчиков, что делает данный фреймворк отличным выбором.

В новом каталоге создайте файл `app.coffee`, который будет играть роль отправной точки для серверной части приложения. Прямо сейчас, только оставьте этот файл пустым, пока не завершите установку фреймворка Express.

Фреймворк Express распространяется в виде модуля через *систему управления пакетами Node* (Node Package Manager, NPM), [3]. Система NPM коротко упоминалась в главе 9, «Введение в Node.js». NPM-модули представляют собой библиотеки, подготовленные к распространению и установке. Они могут непосредственно подключаться приложениями на платформе Node. Если у вас платформа Node уже установлена, система NPM так же должна быть установлена.

Установка фреймворка Express выполняется одной простой командой:

```
> npm install express
```

Она должна вывести строки, напоминающие следующее:

```
express@2.5.2 ./node_modules/express
--- mime@1.2.4
--- mkdirp@0.0.7
--- qs@0.4.0
--- con 2nect@1.8.3
```

После этого в каталоге приложения должна появиться новая папка `node_modules`. В ней система NPM будет сохранять устанавливаемые модули. Позднее мы установим в этот каталог еще несколько модулей.

> **Совет.** NPM-модули можно устанавливать глобально, используя ключ `-g`: `npm install -g express`. Я предпочитаю устанавливать модули в проект, чтобы их можно было сохранять в системе управления исходными текстами (SCM). Это существенно упрощает развертывание приложений и избавляет от вероятных конфликтов между разными версиями.

После установки фреймворка Express добавим плоти на кости в файл `app.coffee`, созданный ранее.

Пример: (исходный файл: `app.1/app.coffee`)

```
# Подготовка Express.js:
global.express = require('express')
global.app = app = express.createServer()
require("#{__dirname}/src/configuration")
# Настройка маршрута для домашней страницы:
require("#{__dirname}/src/controllers/home_controller")
# Запуск сервера:
app.listen(3000)
console.log("Express server listening on port %d in %s mode",
↪app.address().port, app.settings.env)
```

Коротко рассмотрим, что здесь делается. Во-первых, мы подключили модуль `express`. Затем вызвали функцию `createServer`. Она делает именно то, что следует из ее имени. Ссылка на вновь созданный сервер была присвоена двум переменным: локальной переменной `app` и глобальной переменной с тем же именем `app`.

> **Совет.** В платформе Node объект `global` представляет глобальную область видимости приложения, подобно объекту `window` в браузерах. Определяя атрибуты в объекте `global`, мы получаем возможность ссылаться на них из любой точки приложения.

Далее необходимо определить настройки фреймворка Express. Мы поместим их в отдельный файл src/configuration.coffee:

Пример: (исходный файл: app.1/src/configuration.coffee)

```
# Настройка фреймворка Express.js:
app.configure ->
    app.use(express.bodyParser())
    app.use(express.methodOverride())
    app.use(express.cookieParser())
    app.use(express.session(secret: 'd19e19fd62f62a216ecf7d7b1de434ad'))
    app.use(app.router)
    app.use(express.static(__dirname + '../public'))
     app.use(express.errorHandler(dumpExceptions: true, showStack: true))
```

Мы сообщили приложению о необходимости поддержки таких функций, как сеансы, обработка данных cookie, маршрутизация, обработка ошибок и парсинг содержимого HTTP-запросов. Мы также определили каталоги, где приложение сможет найти статические ресурсы, такие как файлы изображений и HTML-файлы.

> **Совет.** В платформе Node при обращении к переменной __dirname возвращается путь к каталогу, в котором находится ссылающийся на нее файл. Эта переменная с успехом может использоваться для конструирования путей к другим файлам.

Затем необходимо настроить маршрут для обработки домашней страницы. Реализована она будет в файле с именем src/controllers/home_controller.coffee. Файл home_controller.coffee будет подключаться в файле app.coffee с помощью функции require, дающей возможность загружать в приложение другие файлы.

Пример: (исходный файл: app.1/src/controllers/home_controller.coffee)

```
# Настройка маршрута для домашней страницы:
app.get '/', (req, res) ->
      res.send "Hello, World!"
```

В объекте app, связанном с глобальной областью видимости приложения, необходимо отобразить путь к домашней странице, когда кто-то попытается открыть ее. Для этого используется функция get объекта app. Существуют функции для отображения всех четырех

типов HTTP-запросов: `get`, `post`, `put`, `delete`. Предполагается, что при открытии домашней страницы будет выполняться HTTP-запрос `GET`, поэтому именно эта функция используется для отображения пути /.

Помимо отображаемого адреса URL (/), функция `get` принимает функцию обратного вызова, которая будет вызываться при обращении по этому адресу. Функции обратного вызова передается объект запроса `req` и объект ответа `res`, для выполнения операций с ними. В данном случае функция отправляет в ответе строку «Hello, World!».

После определения маршрута к домашней странице, остается лишь в файле `app.coffee` сообщить серверу, какой порт должен использоваться, и запустить его, когда мы запустим файл на выполнение.

Теперь запустите его!

```
> coffee app.coffee
```

После перехода по адресу http://localhost:3000 вы должны увидеть знакомое приветствие «Hello, World!».

Поздравляю, теперь у вас есть действующее приложение на платформе Express!

Получив действующее приложение, добавим в него механизм обработки шаблонов, чтобы не приходилось вставлять всю разметку HTML в файл `home_controller.coffee`. Необходимый механизм можно получить в виде единственного файла. Для этого следует установить NPM-пакет модуля `ejs`, позволяющего встраивать сценарии на JavaScript в шаблоны:

```
> npm install ejs
```

Эта команда должна вывести строку, напоминающую следующую:

```
ejs@0.6.1 ./node_modules/ejs
```

> **Совет.** Мы легко могли бы реализовать обслуживание простых HTML-файлов с тем, что уже имеется, однако механизм шаблонов `ejs` позволяет легко встраивать динамическое содержимое в HTML-страницы. Это может оказаться и, как правило, оказывается очень удобно.

Далее, необходимо сообщить фреймворку Express, где будут находиться шаблоны, и что эти шаблоны должны обрабатываться с помощью модуля `ejs`. Для этого нужно добавить две строки в конец файла `src/configuration.coffee`:

Пример: (исходный файл: app.2/src/configuration.coffee)

```
# Настройка фреймворка Express.js:
app.configure ->
    app.use(express.bodyParser())
    app.use(express.methodOverride())
    app.use(express.cookieParser())
    app.use(express.session(secret: 'd19e19fd62f62a216ecf7d7b1de434ad'))
    app.use(app.router)
    app.use(express.static(__dirname + '../public'))
    app.use(express.errorHandler(dumpExceptions: true, showStack: true))
    app.set('views', "#{__dirname}/views")
    app.set('view engine', 'ejs')
```

Создайте новый файл src/views/index.ejs. Этот файл будет играть роль домашней страницы приложения, когда кто-то будет обращаться по адресу, для которого определен путь к домашней странице. Сделаем эту страницу максимально простой – в ней будет выводиться то же приветствие «Hello, World!», но добавим в нее немного динамического содержимого, чтобы убедиться, что такая возможность существует:

Пример: (исходный файл: app.2/src/views/index.ejs)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todos</title>
    </head>
    <body>
        <h1>Hello, World!</h1>
        <h2>The date/time is: <%= new Date() %></h2>
    </body>
</html>
```

Наконец, необходимо отредактировать файл home_controller.coffee, чтобы задействовать новый файл index.ejs.

Пример: (исходный файл: app.2/src/controllers/home_controller.coffee)

```
# Настройка маршрута для домашней страницы:
app.get '/', (req, res) ->
      res.render 'index', layout: false
```

Здесь мы убрали вызов функции `send` объекта ответа, заменив его вызовом функции `render`, которой передается имя используемого шаблона. Мы также сообщили механизму шаблонов, что он не должен искать шаблон `layout` компоновки страницы.

> **Совет.** Важно отметить, что любое динамическое содержимое, помещаемое в шаблон `ejs`, должно быть написано на языке JavaScript, а не на CoffeeScript. Вы не раз будете обжигаться на этом, как это случалось со мной! Конечно, существуют неплохие механизмы обработки шаблонов, основанные на языке CoffeeScript, но они чуть более сложны в установке. Я рекомендую обратить внимание на такие механизмы, как Eco [4] и CoffeeKup. [5]

Запустите сервер и вы должны увидеть в браузере новую домашнюю страницу с текущей датой и временем.

Перейдем к следующему шагу – настройке базы данных.

Настройка MongoDB с помощью Mongoose

Список задач необходимо где-то хранить и для этой цели мы будем использовать MongoDB. [6] *MongoDB*, или просто Mongo, – это популярное NoSQL-хранилище объектов документов. Оно напоминает реляционные базы данных, в которых можно хранить данные, но обеспечивает большую гибкость, не требуя описания схемы таблиц. То есть, можно сразу приступить к использованию этого хранилища, не разрабатывая сценарии создания таблиц. Это делает его отличным выбором, потому что нам не придется продираться через дебри создания таблиц и других элементов базы данных. Можно просто сохранять информацию о задачах в хранилище.

Далее я буду предполагать, что вы уже установили MongoDB. В противном случае, сделайте это [7] прямо сейчас и возвращайтесь, когда закончите.

Что необходимо, чтобы задействовать Mongo в приложении? Во-первых, нужно установить Mongoose [8] – популярный фреймворк объектно-реляционного отображения (Object Relational Mapping, ORM) для платформы Node, который работает с хранилищем Mongo. Фреймворк Mongoose доступен в виде модуля NPM:

```
> npm install mongoose
mongoose@2.4.7 ./node_modules/mongoose
```

```
--- hooks@0.1.9
--- colors@0.5.1
---  mongodb@0.9.7-2-1
```

> **Совет.** Технически Mongoose не является механизмом ORM, потому что MongoDB является не реляционной базой данных, а хранилищем документов. Однако термин ORM получил настолько широкое толкование, что под ним подразумевают и такие механизмы, как Mongoose, поэтому я буду продолжать использовать его.

После установки Mongoose необходимо настроить его использование в приложении. Сделать это очень легко. Сначала создайте файл `src/models/database.coffee`. В каталоге `src/models` будет находиться весь программный код, реализующий операции с базой данных.

Содержимое этого файла показано ниже:

Пример: (исходный файл: `app.3/src/models/database.coffee`)

```
# Настройка Mongoose (MongoDB):
global.mongoose = require('mongoose')
global.Schema = mongoose.Schema
global.ObjectId = Schema.ObjectId
mongoose.connect("mongodb://localhost:27017/csbook-todos")
```

В этом файле сначала подключается модуль `mongoose` и затем он присваивается свойству объекта `global`, чтобы проще было обращаться к нему из других участков программы.

В последней строке в этом файле мы сообщаем фреймворку Mongoose, как подключиться к серверу Mongo и какую базу данных использовать. Содержимое этой строки может изменяться в зависимости от настроек системы.

Все, что осталось сделать для завершения настройки Mongoose – подключить в файле `app.coffee` только что созданный файл `src/models/database.coffee`, как показано ниже:

Пример: (исходный файл: `app.3/app.coffee`)

```
# Настройка фреймворка Express.js:
global.express = require('express')
global.app = app = express.createServer()
require("#{__dirname}/src/configuration")
# Настройка базы данных:
require("#{__dirname}/src/models/database")
```

```
# Настройка маршрута для домашней страницы:
require("#{__dirname}/src/controllers/home_controller")
# Запуск сервера:
app.listen(3000)
console.log("Express server listening on port %d in %s mode",
↪app.address().port,  app.settings.env)
```

После настройки Mongoose, завершим этот раздел созданием модели `Todo` представления данных. Модель `Todo` не содержит ничего необычного. В ней должны быть: заголовок, идентификатор, информация о состоянии (чтобы знать, выполнена ли задача или еще ожидает выполнения) и дата создания задачи.

Новая модель в фреймворке Mongoose создается с помощью функции `model`, которой передается новый объект `Schema`. Объект `Schema` сообщает фреймворку Mongoose и хранилищу Mongo типы данных для модели `Todo`.

Пример: (исходный файл: `app.3/src/models/todo.coffee`)

```
# Модель Todo для фреймворка Mongoose:
global.Todo = mongoose.model 'Todo', new Schema
    id: ObjectId
    title:
        type: String
        validate: /.+/
    state:
        type: String
        default: 'pending'
    created_at:
        type: Date
        default:  Date.now
```

Объект `Schema` для модели `Todo` содержит все, что необходимо. Его определение достаточно очевидно. Для свойств `state` и `created_at` определены полезные атрибуты `default` со значениями по умолчанию. Для свойства `title` предусматривается простейшая проверка его содержимого, требующая наличия в нем хотя бы одного символа перед сохранением.

> **Совет.** Фреймворк Mongoose позволяет определять достаточно сложные проверки. За дополнительной информацией по этой теме обращайтесь к исчерпывающей документации [9].

Наконец, необходимо добавить в файл `app.coffee` подключение только что созданной модели `Todo`.

Пример: (исходный файл: `app.3/app.coffee`)

```
# Настройка фреймворка Express.js:
global.express = require('express')
global.app = app = express.createServer()
require("#{__dirname}/src/configuration")
# Настройка базы данных:
require("#{__dirname}/src/models/database")
# Подключить модель Todo:
require("#{__dirname}/src/models/todo")
# Настройка маршрута для домашней страницы:
require("#{__dirname}/src/controllers/home_controller")
# Запуск сервера:
app.listen(3000)
console.log("Express server listening on port %d in %s mode",
➥app.address().port,  app.settings.env)
```

Теперь база данных и модели готовы к использованию. Как взаимодействовать с моделью `Todo` будет показано в следующем разделе, где мы приступим к созданию контроллера, обрабатывающего запросы к списку задач.

Создание Todo API

В серверной части нашего приложения списка задач осталось лишь реализовать *прикладной интерфейс доступа к задачам* из клиентского программного кода, который будет написан в следующей главе. Для начала создадим файл, где будет сосредоточена обработка всех запросов:

Пример: (исходный файл: `app.4/src/controllers/todos_controller.coffee`)

```
# Этот 'контроллер' будет обрабатывать запросы
# к серверу списка задач
# Получение списка задач:
app.get '/api/todos', (req, res) ->
    res.json [{}]
# Создание новой задачи:
app.post '/api/todos', (req, res) ->
    res.json {}
```

```
# Получение информации о конкретной задаче:
app.get '/api/todos/:id', (req, res) ->
    res.json {}
# Изменение конкретной задачи:
app.put "/api/todos/:id", (req, res) ->
    res.json {}
# Удаление конкретной задачи:
app.delete '/api/todos/:id', (req, res) ->
      res.json {}
```

Первое, на что необходимо обратить внимание прямо сейчас, – мы создали заготовки для обработки всех запросов, которые возвращают пустые объекты JSON [10]. Мы добавим реализацию в эти заготовки чуть ниже, а пока я хотел бы немного поговорить об имеющихся заготовках и как они действуют.

Выше в этой главе упоминалось, что фреймворк Express предоставляет функции для обработки всех четырех типов HTTP-запросов: `get`, `post`, `put` и `delete`. Все эти функции используются в файле `todos_controller.coffee`, поэтому можно утверждать, что наш API соответствует подходу REST [11].

> **Совет.** Можно было бы использовать только запросы `GET` и `POST`, но, на мой взгляд, это сделало бы наш API более запутанным. Подход REST позволяет более четко и понятно отразить наши намерения в API.

Нам требуется реализовать пять операций со списком задач.

Первая – получение списка всех задач, хранящихся в базе данных. Этой операции соответствует первый маршрут, объявленный выше.

Вторая операция – создание новой задачи. Ее реализация находится в функции `post`.

Третья операция обеспечивает возможность извлечения конкретной задачи из базы данных. Ей соответствует третий маршрут. Чтобы извлечь задачу из базы данных, необходимо знать ее идентификатор. Фреймворк Express позволяет определять отображения маршрутов с параметрами в них. Внутри функции обратного вызова имеется возможность извлекать значения этих параметров. Здесь роль параметра играет часть `:id` отображаемого пути. Получить значение параметра `:id` можно с помощью функции `param` доступного нам объекта `req`. Тот же прием будет использоваться в операциях изменения и удаления задачи.

Четвертая операция, которую необходимо обеспечить, – изменение задачи. Для реализации этой операции используется функция `put`, соответствующая HTTP-запросу `PUT`.

Последний маршрут и операция – удаление конкретной задачи.

Все реализации операций будут возвращать ответ в виде объекта JSON. В этом нам поможет метод `json` объекта ответа, предоставляемый фреймворком Express. Без метода `json` нам пришлось бы отправлять ответы вручную, как показано ниже:

```
res.send JSON.stringify({})
```

Создав необходимые заготовки, можно убедиться, что все действует, как ожидается, запустив приложение и открыв страницу http://localhost:3000/api/todos. На экране должна появиться строка:

```
[{}]
```

Она означает, что запрос успешно достиг функции, возвращающей список всех задач.

Выполнение запросов с помощью Mongoose

Прежде чем завершить главу и перейти к облачению этого негодника в красивые одежки, необходимо добавить реализацию каждой операции, поддерживаемой нашим API.

Извлечение всех задач

Начнем с извлечения всех задач, хранящихся в базе данных:

Пример: (исходный файл: `app.5/src/controllers/todos_controller.coffee`)

```
# Этот 'контроллер' будет обрабатывать запросы
# к серверу списка задач
# Получение списка задач:
app.get '/api/todos', (req, res) ->
    Todo.find {}, [], {sort: [["created_at", -1]]}, (err, @todos) =>
        if err?
            res.json(err, 500)
        else
            res.json @todos
# Создание новой задачи:
```

```
app.post '/api/todos', (req, res) ->
    res.json {}
# Получение информации о конкретной задаче:
app.get '/api/todos/:id', (req, res) ->
    res.json {}
# Изменение конкретной задачи:
app.put "/api/todos/:id", (req, res) ->
    res.json {}
# Удаление конкретной задачи:
app.delete '/api/todos/:id', (req, res) ->
    res.json {}
```

Механизм Mongoose предоставляет множество удобных функций класса, помогающих отыскивать записи в базе данных. В данном случае необходимо получить все записи.

> **Совет.** Фреймворк Mongoose снабжен отличной документацией. Я настоятельно рекомендую заглянуть в ее разделы, посвященные поиску документов [12] и построению сложных запросов. [13]

Чтобы отыскать все записи, достаточно просто обратиться к функции `find` и передать ей функцию, которая должна быть вызвана в случае успешного или неудачного выполнения запроса. Нам также требуется отсортировать список задач в обратном хронологическом порядке, чтобы первыми выводились последние задачи, а последними – самые старые. Для этого необходимо определить объект, описывающий порядок сортировки. Здесь мы сталкиваемся с некоторыми хитростями. Получить отсортированный список можно двумя способами: либо разбить реализацию операции поиска на несколько строк, сконструировать запрос и затем выполнить его, либо сделать так, как мы поступили здесь, – передать несколько пустых аргументов перед аргументом, определяющим порядок сортировки. В первом аргументе передается объект, представляющий условия, которые нужно вставить в запрос. В нашем случае не требуется ограничивать выборку данных. Второй аргумент, массив, позволяет указать поля, значения которых должны извлекаться. Нам необходимы все поля, поэтому мы оставили массив пустым.

> **Совет.** Почему при определении параметров сортировки в вызове функции `find` используются вложенные массивы? В действительности это обусловлено особенностями MongoDB. Если бы потребовалось отсортировать результаты по нескольким полям, мы

могли бы, например, определить такой порядок сортировки: {sort: [["created_at", -1], ["updated_at", 1]]}. Вложенные массивы представляют группу полей, по которым требуется выполнить сортировку. Первый элемент каждого массива – имя атрибута, а второй – направление сортировки, -1 – для сортировки по убыванию, и 1 – для сортировки по возрастанию. Как мне кажется, все это можно было реализовать проще, но, что есть, то есть.

Как уже упоминалось, функция обратного вызова, которая передается функции find, выполняется независимо от успешности выполнения запроса. Это означает, что в ней необходимо предусмотреть обработку ошибок.

Функция обратного вызова получает два аргумента. В случае ошибки первый аргумент будет содержать объект Error, или значение null в противном случае. Второй аргумент представляет результаты запроса, в данном случае – массив объектов Todo. Если при выполнении запроса возникнет ошибка, во втором аргументе будет передано значение null.

Внутри функции обратного вызова в первую очередь необходимо проверить, наличие ошибки. Сделать это можно с помощью оператора существования ?. Оператор существования проверяет, не содержит ли аргумент err значение undefined или null. В случае ошибки вызывается функция json, которой передается объект err и код 500. Это позволит пользовательскому интерфейсу вывести соответствующее сообщение об ошибке.

Если ошибка не обнаружена, функции json передается список задач @todos, полученный в результате выполнения запроса, а обо всем остальном позаботится фреймворк Express.

Совет. Обратите внимание, что функция обратного вызова определена с помощью оператора =>. Это необходимо, чтобы гарантировать доступность объектов req и res. Фреймворк Mongoose вызывает все функции асинхронно, а это означает, что сразу после вызова функции программа может продолжить дальнейшее выполнение, и объекты req и res окажутся недоступны функции.

Создание новых задач

Двинемся дальше и реализуем *создание новой задачи*.

Пример: (исходный файл: app.6/src/controllers/todos_controller.coffee)

```
# Этот 'контроллер' будет обрабатывать запросы
# к серверу списка задач
# Получение списка задач:
app.get '/api/todos', (req, res) ->
    Todo.find {}, [], {sort: [["created_at", -1]]}, (err, @todos) =>
        if err?
            res.json(err, 500)
        else
            res.json @todos
# Создание новой задачи:
app.post '/api/todos', (req, res) ->
    @todo = new Todo(req.param('todo'))
    @todo.save (err) =>
        if err?
            res.json(err, 500)
        else
            res.json @todo
# Получение информации о конкретной задаче:
app.get '/api/todos/:id', (req, res) ->
    res.json {}
# Изменение конкретной задачи:
app.put "/api/todos/:id", (req, res) ->
    res.json {}
# Удаление конкретной задачи:
app.delete '/api/todos/:id', (req, res) ->
     res.json  {}
```

Создание новой задачи выглядит достаточно просто. Сначала создается новый экземпляр класса `Todo`, при этом конструктору передается объект, представляющий значения атрибутов новой задачи. Откуда берутся эти значения? Они извлекаются из параметров HTTP-запроса `POST`, обработка которого отображается на эту операцию.

Фреймворк Express позволяет извлекать параметры HTTP-запроса вызовом функции `param` объекта `req`. Ей передается имя желаемого параметра, а она возвращает значение этого параметра или `null`, если параметр с таким именем отсутствует.

В нашей функции выполняется поиск параметра `todo`, который, как предполагается, является объектом, содержащим пары ключ/значение, необходимые для создания новой задачи, например:

```
{
    todo: {
        title: "My Todo Title",
```

```
        state: "pending"
    }
}
```

Чтобы сохранить новую задачу, вызывается функция save, предоставляемая фреймворком Mongoose, которой передается функция обратного вызова. Эта функция принимает один аргумент, представляющий ошибку, если такая возникнет. Наша функция обратного вызова следует той же логике, что и аналогичная функция в операции получения списка всех задач. Сначала она проверяет наличие ошибки. В случае ошибки клиенту отправляется объект err и код 500, в противном случае – объект JSON со вновь созданной задачей.

> **Совет.** Технически, при точном соблюдении протокола REST, операция создания новой задачи должна вернуть код 201 («успешное создание»). Однако по умолчанию в случае успеха фреймворк Express возвращает код 200 («успех»), что, впрочем, вполне подходит для наших нужд. Я оставляю за вами реализацию отправки кода 201.

Получение, изменение и удаление задачи

Три операции, которые осталось реализовать в нашем API, очень похожи друг на друга, поэтому мы быстро рассмотрим их все сразу.

Пример: (исходный файл: app.7/src/controllers/todos_controller.coffee)

```
# Этот 'контроллер' будет обрабатывать запросы
# к серверу списка задач
# Получение списка задач:
app.get '/api/todos', (req, res) ->
    Todo.find {}, [], {sort: [["created_at", -1]]}, (err, @todos) =>
        if err?
            res.json(err, 500)
        else
            res.json @todos
# Создание новой задачи:
app.post '/api/todos', (req, res) ->
    @todo = new Todo(req.param('todo'))
    @todo.save (err) =>
        if err?
            res.json(err, 500)
        else
            res.json @todo
```

```
# Получение информации о конкретной задаче:
app.get '/api/todos/:id', (req, res) ->
    Todo.findById req.param('id'), (err, @todo) =>
        if err?
            res.json(err, 500)
        else
            res.json @todo
# Изменение конкретной задачи:
app.put "/api/todos/:id", (req, res) ->
    Todo.findById req.param('id'), (err, @todo) =>
        if err?
            res.json(err, 500)
        else
            @todo.set(req.param('todo'))
            @todo.save (err) =>
                if err?
                    res.json(err, 500)
                else
                    res.json @todo
# Удаление конкретной задачи:
app.delete '/api/todos/:id', (req, res) ->
    Todo.findById req.param('id'), (err, @todo) =>
        if err?
            res.json(err, 500)
        else
            @todo.remove()
            res.json  @todo
```

Первое, что необходимо сделать в реализациях всех этих операций, – отыскать задачу по ее идентификатору `id`, которому в пути соответствует ключевое слово `:id`. Выполнить такой поиск можно, вызвав функцию класса `findById`, предоставляемую фреймворком Mongoose, и передав ей параметр `id`.

Функция `findById` принимает функцию обратного вызова, которой передает два аргумента. Как обычно, в первом аргументе передается объект ошибки, если она возникла, а во втором – объект задачи, если она была найдена.

Дальнейшие действия с найденной задачей зависят от типа операции, в которой выполняется поиск. Операция, которая должна лишь отобразить задачу, просто отправляет объект JSON с задачей.

Операция, изменяющая задачу, вызывает функцию `set`, которой передаются атрибуты, подлежащие изменению. В этом данная

операция мало чем отличается от операции создания новой задачи, реализованной ранее.

Наконец, операция удаления задачи просто вызывает функцию `remove` задачи, чтобы удалить ее из базы данных.

Реорганизация контроллера

Возможно вы уже заметили, что наш контроллер изобилует повторяющимися фрагментами программного кода. Поскольку это всего лишь учебный пример, мы могли бы оставить его в таком виде, но почему бы не попробовать *реорганизовать* его? Это даст вам возможность еще немного поиграть с классами.

> **Совет.** В действительности реорганизация программного кода, которую я собираюсь продемонстрировать, может показаться вам излишней или даже неполной, но я думаю, что вам будет интересно увидеть, как при желании можно реорганизовать программный код.

Для реорганизации программного кода создадим несколько классов, по одному для каждой операции. Мы также будем использовать наследование, чтобы избежать повторения программного кода, особенно при обработке ошибок, где наблюдается больше всего повторяющихся фрагментов.

Начнем с создания базового класса `Responder`, который будет наследоваться всеми остальными классами. Этот класс будет содержать реализацию функциональности по умолчанию и общей для всех классов.

Пример: (исходный файл: `app.8/src/controllers/responders/responder.coffee`)

```
class global.Responder
    respond: (@req, @res) =>
        @complete(null, {})
    complete: (err, result = {}) =>
        if err?
            @res.json(err, 500)
        else
            @res.json  result
```

В первую очередь обратите внимание, что класс `Responder` определен, как атрибут объекта `global`. Это упрощает доступ к нему из любых частей приложения.

Далее в классе определена функция по умолчанию с именем respond. Эта функция будет передаваться операциям, объявленным в файле todos_controller.coffee. Функция respond принимает два аргумента, объект запроса и объект ответа. Мы автоматически включаем их в область видимости класса, добавив перед именами символ @, и обеспечиваем их доступность для наших функций.

Функция respond почти всегда будет переопределяться в дочерних классах, но на всякий случай мы предусмотрели реализацию по умолчанию – вызов функции complete.

Функция complete является оберткой для всех повторяющихся фрагментов программного кода, выполняющих обработку ошибок и отправляющих ответ клиенту.

Закончив создание родительского класса, приступим к созданию дочерних классов, выполняющих операции с задачами. Начнем с класса, реализующего операцию получения списка всех задач.

Пример:

(исходный файл: app.8/src/controllers/responders/index_responder.coffee)

```
require "#{__dirname}/responder"
class Responder.Index extends Responder
    respond: (@req, @res) =>
          Todo.find {}, [], {sort: [["created_at", -1]]}, @complete
```

В первую очередь необходимо подключить класс Responder, воспользовавшись функцией require, которая предоставляется платформой Node.

При определении нового класса, его можно присвоить атрибуту объекта global, как это было сделано с классом Responder. Однако лучше стараться организовать программные компоненты в пространства имен, чтобы не пихать все подряд непосредственно в объект global. Именно так мы и поступили здесь. Кроме того, класс Responder.Index наследует класс Responder, о чем свидетельствует ключевое слово extend.

> **Совет.** В настоящем приложении можно было бы выделить классы в пространство имен, ясно свидетельствующее, что они предназначены для обслуживания ресурса Todo. При большом количестве ресурсов реализовать такой подход было бы проблематично.

Нам осталось лишь написать собственный метод respond. Этот метод выполняет знакомый уже запрос, отыскивающий все задачи.

Главное отличие, в сравнение с предыдущей реализацией, состоит в том, что теперь вместо функции обратного вызова мы обращаемся к функции `complete` из родительского класса `Responder`.

Теперь определим класс, реализующий операцию создания новой задачи.

Пример:

(исходный файл: `app.8/src/controllers/responders/create_responder.coffee`)

```
require "#{__dirname}/responder"
class Responder.Create extends Responder
    respond: (@req, @res) =>
        todo = new Todo(@req.param('todo'))
        todo.save(@complete)
```

Класс `Responder.Create` очень похож на только что созданный класс `Responder.Index`. Функция `respond` создает новую задачу и вызывает функцию `save`, передавая ей метод `complete` в виде функции обратного вызова.

Пример:

(исходный файл: `app.8/src/controllers/responders/show_responder.coffee`)

```
require "#{__dirname}/responder"
class Responder.Show extends Responder
    respond: (@req, @res) =>
        Todo.findById @req.param('id'), @complete
```

Класс `Responder.Show` также очень похож на первые два класса. Изменилась лишь реализация функции `respond`. Класс `Responder.Show` будет использоваться как родительский для остальных двух классов.

Пример:

(исходный файл: `app.8/src/controllers/responders/update_responder.coffee`)

```
require "#{__dirname}/show_responder"
class Responder.Update extends Responder.Show
    complete: (err, result = {}) =>
        if err?
            super
```

```
        else
            result.set(@req.param('todo'))
            result.save(super)
```

Поскольку при выполнении операции изменения так же требуется отыскать задачу, вместо класса `Responder` можно унаследовать класс `Responder.Show`, потому что класс `Responder.Show` уже содержит функциональность, необходимую для поиска задачи.

В классе `Responder.Update` нет необходимости создавать собственную функцию `respond`, потому что будет использоваться функция, унаследованная от класса `Responder.Show`. Однако нам придется переопределить функцию `complete`.

Первое, что необходимо сделать в новой функции `complete`, – проверить наличие ошибки. Если ошибка существует, вызывается функция `super`, которая вызовет оригинальную функцию `complete` из класса `Responder` и ошибка будет обработана.

Если ошибка отсутствует, выполняется установка значений требуемых атрибутов и затем вызывается функция `save`. При вызове функции `save` ей передается функция `super`, которая в свою очередь опять же вызовет оригинальную функцию `complete` из класса `Responder`. Она обработает ошибку и отправит соответствующий объект JSON в виде ответа.

Класс, реализующий удаление задачи, очень похож на только что созданный класс изменения задачи.

Пример:

(исходный файл: `app.8/src/controllers/responders/destroy_responder.coffee`)

```
require "#{__dirname}/show_responder"
class Responder.Destroy extends Responder.Show
    complete: (err, result = {}) =>
        unless err?
            result.remove()
        super
```

Он также наследует класс `Responder.Show` и имеет собственную реализацию функции `complete`, подобно классу `Responder.Update`.

Первое, что следует сделать в функции `complete`, – проверить наличие ошибки, которая может возникнуть при попытке отыскать задачу. Если ошибка отсутствует, вызывается функция `remove` задачи, которая удалеит ее из базы данных.

В заключение вызывается функция super, которая обработает возможную ошибку и отправит соответствующий ответ.

Теперь осталось лишь отредактировать файл todos_controller.coffee, чтобы задействовать новые классы:

Пример: (исходный файл: app.8/src/controllers/todos_controller.coffee)

```
# подключить все классы реализации операций:
for name in ["index", "create", "show", "update", "destroy"]
    require("#{__dirname}/responders/#{name}_responder")
# Этот 'контроллер' будет обрабатывать запросы
# к серверу списка задач
# Получение списка задач:
app.get '/api/todos', new Responder.Index().respond
# Создание новой задачи:
app.post '/api/todos', new Responder.Create().respond
# Получение информации о конкретной задаче:
app.get '/api/todos/:id', new Responder.Show().respond
# Изменение конкретной задачи:
app.put "/api/todos/:id", new Responder.Update().respond
# Удаление конкретной задачи:
app.delete '/api/todos/:id', new Responder.Destroy().respond
```

В начале файла необходимо подключить все созданные нами классы.

> **Совет.** Мы могли бы просто вызвать функцию require для каждого файла, так как их у нас совсем немного, но я предпочитаю создавать массив и выполнять итерации по нему. При такой организации программный код выглядит яснее и чтобы подключить еще один файл достаточно просто ввести несколько символов в массив, вместо того, чтобы копировать длинную строку кода.

Подключив классы, можно выбросить прежние функции обратного вызова, реализующие операции, заменить их инструкциями создания новых экземпляров соответствующих классов и вызвать функцию respond этих классов.

Теперь наш файл todos_controller.coffee выглядит намного проще, а общая функциональность, используемая всеми операциями, сосредоточена в одном месте. Это означает, что если в будущем нам потребуется изменить логику обработки ошибок или другую общую функциональность, сделать это придется лишь в одном файле.

В заключение

В этой главе мы написали серверную часть приложения, управляющего списком задач. Мы создали приложение на основе фреймворка Express, с помощью фреймворка Mongoose добавили в него поддержку хранилища MongoDB и реорганизовали программный код.

В следующей главе мы возьмем все, что было создано здесь, и покроем соблазнительными [14] одеждами. Для взаимодействия со своим замечательным серверным кодом мы будем использовать библиотеку jQuery.

Примечания

1. https://github.com/markbates/Programming-In-CoffeeScript.
2. http://expressjs.com/.
3. http://npmjs.org/.
4. https://github.com/sstephenson/eco.
5. http://coffeekup.org/.
6. http://www.mongodb.org/.
7. http://www.mongodb.org/.
8. http://mongoosejs.com/.
9. http://mongoosejs.com/docs/validation.html.
10. http://www.json.org/.
11. http://ru.wikipedia.org/wiki/REST.
12. http://mongoosejs.com/docs/finding-documents.html.
13. http://mongoosejs.com/docs/query.html.
14. Это была шутка. Любой, кто знает меня, скажет вам, что «соблазнение» не мой конек.

11. Пример: список задач, часть 2 (клиент на основе jQuery)

В главе 10, «Пример: список задач, часть 1 (серверная)», мы только что завершили разработку серверных компонентов приложения управления списком задач. В данной главе мы создадим пользовательский интерфейс этого приложения, который будет работать в браузере. Реализация интерфейса будет выполнена с применением весьма интересных инструментов, таких как Bootstrap [1] от компании Twitter [2] и jQuery.

Подготовка HTML с помощью Twitter Bootstrap

Начнем с подготовки некоторых основных стилей и разметки HTML для нашего приложения. Чтобы упростить работу со стилями CSS и оформление страниц, мы будем использовать проект Bootstrap компании Twitter. *Bootstrap* – это простой набор файлов со стилями CSS и сценариями JavaScript, упрощающий разработку пользовательского интерфейса приложения. Он реализует простую сетку, помогающую разместить элементы интерфейса на странице. Кроме того, в этот набор входят несколько привлекательных стилей оформления форм, кнопок, списков и многих других визуальных компонентов. Я настоятельно рекомендую ознакомиться с проектом поближе, чтобы понять, что он может предложить, потому что в этом приложении мы задействуем лишь малую его часть.

Первое, что необходимо сделать, – отредактировать файл `src/views/index.ejs`, чтобы задействовать в нем библиотеку Bootstrap, а также наши стили CSS. Благодаря этому позднее можно будет выполнить необходимые настройки для приложения.

Пример: (исходный файл: `app.1/src/views/index.ejs`)

```
<!DOCTYPE html>
<html>
```

```
    <head>
        <title>Todos</title>
        <link rel="stylesheet"
href="http://twitter.github.com/bootstrap/1.4.0/bootstrap.min.css">
        <%- css('/application') %>
    </head>
    <body>
    </body>
</html>
```

Мы добавили ссылку на файл CSS, входящий в состав библиотеки Bootstrap, а также довольно необычный фрагмент программного кода – вызов функции с именем `css`, которой передается значение `'/application'`. Именно так будут добавляться в приложение наши собственные стили CSS, а потом и программный код на CoffeeScript.

Работу необычного фрагмента обеспечивает модуль `connect-assets` из системы NPM, созданный Тревором Барнхамом (Trevor Burnham). [3] Этот модуль предоставляет пару функций, позволяющих автоматически отыскивать файлы CSS и CoffeeScript, и добавлять их в разметку HTML. В случае с программным кодом на CoffeeScript, он автоматически компилирует его в код на JavaScript, избавляя нас от необходимости делать это вручную.

> **Совет.** Те, кто знаком с модулем `asset-pipeline` из фреймворка Ruby on Rails, обнаружат, что модуль `connect-assets`, предназначенный для использования в приложениях на основе фреймворка Express, довольно близко имитирует его.

Чтобы модуль `connect-assets` смог отыскивать файлы ресурсов, их необходимо поместить в папку `assets`, находящуюся в корневом каталоге приложения. Создайте эту папку прямо сейчас и заодно создайте в ней файл со стилями CSS для украшения HTML-страниц, которые мы будем создавать.

Пример: (исходный файл: `app.1/assets/application.css`)

```
#todos li {
    margin-bottom: 20px;
}
#todos li .todo_title {
    width: 800px;
}
#todos li .completed {
```

```
    text-decoration: line-through;
}
#todos #new_todo .todo_title{
    width: 758px;
}
```

Поскольку эта книга не о стилях CSS, я не буду объяснять, как они действуют. Если вам действительно интересно, попробуйте позднее закомментировать их и посмотреть, какой эффект это окажет на приложение.

Теперь необходимо установить модуль `connect-assets`, чтобы все, что мы напишем, заработало.

```
> npm install connect-assets
connect-assets@2.1.6 ./node_modules/connect-assets
--- connect-file-cache@0.2.4
--- underscore@1.1.7
--- mime@1.2.2
---  snockets@1.3.3
```

В заключение нужно сообщить фреймворку Express, что для обслуживания наших файлов ресурсов он должен использовать модуль `connect-assets`. Сделать это можно, добавив строку в конец файла `configuration.coffee`:

Пример: (исходный файл: `app.1/src/configuration.coffee`)

```
# Настройка фреймворка Express.js:
app.configure ->
    app.use(express.bodyParser())
    app.use(express.methodOverride())
    app.use(express.cookieParser())
    app.use(express.session(secret: 'd19e19fd62f62a216ecf7d7b1de434ad'))
    app.use(app.router)
    app.use(express.static(__dirname + '../public'))
    app.use(express.errorHandler(dumpExceptions: true, showStack: true))
    app.set('views', "#{__dirname}/views")
    app.set('view engine', 'ejs')
    app.use(require('connect-assets')())
```

Если запустить приложение прямо сейчас и открыть в браузере адрес http://localhost:3000, мы увидим унылую, белую, пустую страницу. Если вы увидели что-то другое, значит где-то вы допустили ошибку.

Закончим этот раздел добавлением формы для создания новой задачи. Она нам очень пригодится, потому что сейчас в базе данных нет ни одной задачи, и эта форма даст нам возможность создать для пробы несколько задач.

Пример: (исходный файл: `app.2/src/views/index.ejs`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todos</title>
        <link rel="stylesheet" href="http://twitter.github.com/bootstrap/
↪1.4.0/bootstrap.min.css">
        <%- css('/application') %>
    </head>
    <body>
        <div class="container">
            <h1>Todo List</h1>
            <ul id='todos' class='unstyled'>
                <li id='new_todo'>
                    <div class="clearfix">
                        <div class="input">
                            <div class="input-prepend">
                                <span class='add-on'>New Todo</span>
                                <input class="xlarge todo_title" size="50"
↪type="text" placeholder="Enter your new Todo here..." />
                            </div>
                        </div>
                    </div>
                </li>
            </ul>
        </div>
    </body>
</html>
```

Если теперь запустить приложение и открыть его в браузере, должна появиться привлекательно оформленная форма, куда можно ввести данные для новой задачи. Эта форма пока не действует, но мы позаботимся об этом чуть ниже. Если вас интересует, откуда берутся все эти CSS-классы, используемые в разметке HTML – они определены в библиотеке Bootstrap, которую мы подключили буквально минуту назад.

Организация взаимодействий с помощью jQuery

Теперь, когда у нас есть форма, включим ее в работу и посмотрим, что из этого получится. В этом нам поможет jQuery – замечательная библиотека, в которую, похоже, влюблены все, имеющие доступ в Интернет. Для меня не существует более мощного набора инструментов в экосистеме JavaScript, чем jQuery. Первоначально она была создана в 2006 году Джоном Резигом (John Resig), [4] сейчас библиотека jQuery используется на 49% из 10000 [5] самых крупных веб-сайтов и является самой популярной библиотекой на языке JavaScript. Библиотека jQuery позволяет писать краткий и выразительный программный код, реализующий операции с деревом DOM, выполняющий AJAX-запросы, обрабатывающий события и воспроизводящий простейшие анимационные эффекты. Кроме этого она является кросс-платформенной, в том смысле, что поддерживает все основные браузеры и операционные системы.

> **Совет.** В древнюю мрачную эпоху развития Веб, разработчикам приходилось писать несколько версий одного и того же программного кода на JavaScript. Одна версия предназначалась для Internet Explorer, другая – для Netscape, и так далее. Современная библиотека jQuery позволяет писать одну версию кода и пребывать в уверенности, что она будет работать одинаково во всех основных браузерах.

Добавить библиотеку jQuery в приложение очень просто – достаточно лишь подключить ее в файле `index.ejs`:

Пример: (исходный файл: `app.3/src/views/index.ejs`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todos</title>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
➥jquery.min.js" type="text/javascript"></script>
        <%- js('/application') %>
        <link rel="stylesheet" href="http://twitter.github.com/bootstrap/
➥1.4.0/bootstrap.min.css">
        <%- css('/application') %>
    </head>
    <body>
        <div class="container">
```

```
            <h1>Todo List</h1>
            <ul id='todos' class='unstyled'>
                <li id='new_todo'>
                    <div class="clearfix">
                        <div class="input">
                            <div class="input-prepend">
                                <span class='add-on'>New Todo</span>
                                <input class="xlarge todo_title" size="50"
➥type="text" placeholder="Enter your new Todo here..." />
                            </div>
                        </div>
                    </div>
                </li>
            </ul>
        </div>
    </body>
</html>
```

Совет. В настоящих приложениях я стараюсь избегать подключения внешних библиотек, как это было сделано выше. Может так случиться, что из-за изменений в библиотеке в приложении появятся ошибки или, хуже того, внешняя ссылка окажется недействительной и тогда приложение перестанет работать. Я предпочитаю создавать локальные копии библиотек. Однако для наших целей такой способ подключения вполне пригоден.

Добавление формы создания новой задачи

В добавок к подключению библиотеки jQuery, я вызвал функцию `js` из модуля `connect-assets`, чтобы подключить файл `assets/application.coffee`.

Пример: (исходный файл: `app.3/assets/application.coffee`)

```
#= require_tree "jquery"
```

Файл `application.coffee` очень маленький, таким он и останется, а поможет нам в этом модуль `connect-assets`, позволяющий подключать группы файлов с программным кодом на CoffeeScript или JavaScript. В данном случае мы подключили каталог `jquery`. Это означает, что все файлы, находящиеся в этом каталоге, будут автоматически подключены к приложению.

В папке assets создайте новую папку jquery, а внутри нее создайте файл с именем new_todo_form.coffee. В этот файл мы поместим весь программный код, обслуживающий HTML-форму создания новой задачи. А теперь перейдем к созданию кода.

Прежде чем писать код обслуживания формы создания новой задачи, поговорим о том, что он должен делать. Когда пользователь введет в форму информацию о задаче и нажмет клавишу **Enter**, необходимо сначала проверить корректность заполнения формы, чтобы она содержала хотя бы один непробельный символ. В случае ошибки должно выводиться предупреждение, чтобы пользователь увидел возникшую проблему. Если форма заполнена правильно, необходимо отправить данные серверному API. Если в ответ от API будет получен признак успешного добавления задачи, необходимо включить новую задачу в список на странице и очистить форму. Если в ответ сервер сообщит об ошибке, необходимо вывести соответствующее сообщение. А теперь сам код:

Пример: (исходный файл: app.3/assets/jquery/new_todo_form.coffee)

```
$ ->
    # Передать форме фокус ввода сразу после загрузки страницы:
    $('#new_todo .todo_title').focus()
    # Обработка событий keypress в форме создания новой задачи:
    $('#new_todo .todo_title').keypress (e) ->
        # обрабатывается только нажатие клавиши 'enter':
        if e.keyCode is 13
            todo = {title: $(e.target).val()}
            if !todo.title? or todo.title.trim() is ""
                alert "Title can't be blank"
            else
                request = $.post "/api/todos", todo: todo
                request.fail (response) =>
                    message = JSON.parse(response.responseText).message
                    alert message
                request.done (todo) =>
                    $('#new_todo').after("<li>#{JSON.stringify(todo)}</li>")
                    $(e.target).val("")
```

Сразу после загрузки страницы форме передается фокус ввода. Благодаря этому пользователь сразу же сможет начать ввод данных, без необходимости щелкать мышью на форме.

Совет. При использовании библиотеки jQuery можно вызвать функцию с именем `$`, которое является псевдонимом имени `jQuery`, и передать ей свою функцию, которая должна быть вызвана сразу после загрузки страницы. Эту операцию можно выполнить столько раз, сколько потребуется. Удобно.

Далее к форме подключается функция, которая будет вызываться при нажатии клавиш, пока фокус ввода находится в форме. Реализация обработки нажатий клавиш может оказаться немного утомительной, особенно когда требуется обработать одну конкретную клавишу, клавишу **Enter**. Клавише **Enter** соответствует числовой код `13`, поэтому мы сравниваем атрибут `keyCode` объекта события, полученного от библиотеки jQuery, с этим числом. Если код клавиши равен `13`, обработчик продолжает работу, если было получено какое-то другое значение, отличное от 13, событие просто игнорируется.

Убедившись, что это именно то событие, которое ожидалось, можно продолжать его обработку. Далее необходимо получить значение формы. Оно будет использоваться для установки значения атрибута `title` создаваемой задачи. Затем создается объект, представляющий все данные, которые необходимо отправить на сервер.

Получив желаемое значение для атрибута `title` можно выполнить его проверку локально. Локальная проверка производится быстрее и потому не вызовет неприятных переживаний у пользователя, как удаленная проверка, когда приходится ждать пока она будет выполнена на стороне сервера и клиенту вернутся ее результаты.

Совет. Существует множество способов распределения проверок между сервером и клиентом, избавляющих от необходимости дублировать их. Если вы найдете подходящую для вас комбинацию, это будет замечательно. Однако не всегда возможно реализовать проверку, которая с равной степенью надежности может быть выполнена как на стороне клиента, так и на стороне сервера. Примером может служить проверка имени пользователя, которое обычно необходимо передать серверу для проверки уникальности. Выполнить такую проверку можно с помощью технологии AJAX и специализированного API на стороне сервера, либо убедиться на стороне клиента, что имя пользователя не является пустой строкой и позволить серверу выполнить проверку позднее.

Допустим, что локальная проверка увенчалась успехом. Теперь можно передать данные серверной части приложения. Для этого вызовом функции `post` из библиотеки jQuery создается новый AJAX-запрос. Функции передается URL приложения, `/api/todos`, и

объект, представляющий отправляемые данные. Значение, возвращаемое функцией post, присваивается переменой request, которая будет использоваться для подключения функции обратного вызова, обрабатывающей события, возникающие в ходе обработки запроса.

> **Совет.** В jQuery 1.5 появились так называемые отложенные объекты. До этого все функции обратного вызова приходилось определять внутри вызовов функций post и ajax, при первом обращении к ним. Это накладывало определенные ограничения. С появлением отложенных объектов функции обратного вызова стало возможным подключать в любой момент времени, даже после того, как обработка запроса уже завершится. Это позволяет писать более изолированный программный код, подключаемый к запросу.

В нашем случае необходимо подключить к запросу две функции обратного вызова. Первая из них будет вызвана, если по каким-то причинам запрос завершится неудачей. Эта функция получит объект response. Из этого объекта response нам нужно будет извлечь значение атрибута responseText, которое представляет собой текст в формате JSON, выполнить его парсинг, извлечь текст сообщения об ошибке и отобразить это сообщение на экране.

Вторая функция будет вызвана в случае успешного выполнения запроса. В нашем случае – успешного добавления новой задачи в базу данных. Поскольку у нас пока нет шаблона для вывода задач, мы просто выводим ответ в формате JSON внутри тега li и добавляем этот тег в список, непосредственно под формой создания новой задачи, благодаря чему самая последняя задача всегда будет отображаться первой в списке.

Если теперь запустить приложение и попробовать создать новую задачу, сразу под формой должен появиться текст, напоминающий следующий:

```
{"title":"My New Todo","_id":"4efa82bdf65049000000001a","created_at":"2011-
↪12-28T02:45:17.992Z","state":"pending"}
```

Отображение списка задач с помощью шаблонов Underscore.js

Для приложений на JavaScript существует масса систем управления шаблонами, и выбор той или иной из них во многом определяется личными предпочтениями. У нас нет каких-то особых требований,

поэтому я решил выбрать простейшую систему управления шаблонами, которая распространяется в составе библиотеки underscore.js. [6] Почему я выбрал эту систему, а не какую-то другую? Все просто, библиотека underscore.js входит в список зависимостей библиотеки Backbone.js, которая будет использоваться в следующей главе, а так как ее все равно придется подключать, то почему бы не воспользоваться ею уже сейчас.

> **Совет.** Лично я предпочитаю систему управления шаблонами eco, [7] позволяющую встраивать в шаблоны программный код на CoffeeScript. Из других популярных систем управления шаблонами могу порекомендовать Handlebars, [8] Mustache, [9] и Jade. [10] Одно время большой популярностью пользовалось расширение [11] поддержки шаблонов для jQuery, но оно было признано устаревшим, и его дальнейшая разработка была прекращена. Поэтому, если у вас зародилась идея использовать его, я настоятельно рекомендую подыскать что-нибудь другое.

Отредактируйте файл `index.ejs` и добавьте в него подключение библиотеки underscore.js:

Пример: (исходный файл: `app.4/src/views/index.ejs`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todos</title>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
↪jquery.min.js" type="text/javascript"></script>
        <script src="http://documentcloud.github.com/underscore/underscore-
↪min.js" type="text/javascript"></script>
        <%- js('/application') %>
        <link rel="stylesheet" href="http://twitter.github.com/bootstrap/
↪1.4.0/bootstrap.min.css">
        <%- css('/application') %>
    </head>
    <body>
        <div class="container">
            <h1>Todo List</h1>
            <ul id='todos' class='unstyled'>
                <li id='new_todo'>
                    <div class="clearfix">
                        <div class="input">
                            <div class="input-prepend">
```

```
                                <span class='add-on'>New Todo</span>
                                    <input class="xlarge todo_title"
↪size="50" type="text" placeholder="Enter your new Todo here..." />
                                </div>
                            </div>
                        </div>
                    </li>
                </ul>
            </div>
        </body>
</html>
```

Теперь нам придется делать на одну операцию меньше, когда мы будем говорить о библиотеке Backbone.

Далее, создайте файл `assets/templates.coffee`, где будут храниться шаблоны для нашего приложения.

Пример: (исходный файл: `app.4/assets/templates.coffee`)

```
# Изменить синтаксис для шаблонов underscore.js.
# Теперь вместо конструкций вида <%= some_var %> нужно будет
# использовать конструкции {{some_var}}
_.templateSettings =
    interpolate : /\{\{(.+?)\}\}/g
@Templates = {}
Templates.list_item_template = """
<div class="clearfix">
    <div class="input">
        <div class="input-prepend">
            <label class="add-on active"><input type="checkbox"
↪class="todo_state" /></label>
            <input class="xlarge todo_title" size="50" type="text"
↪value="{{title}}" />
            <button class='btn danger'>X</button>
        </div>
    </div>
</div>
"""
```

В самом начале файла изменяется порядок по умолчанию интерполяции динамических данных библиотекой underscore.js. Причина в том, что на мой взгляд вводить комбинации символов {{ }} проще, чем <%= %>. Кроме того, это обычное соглашение, которому следуют многие другие системы управления шаблонами. Поэтому, если

позднее у вас появится желание перейти на использование другой системы, сделать это будет намного проще.

Теперь остается лишь определить шаблон. Для этого создается новый объект `Templates` и его свойству `list_item_template` присваивается требуемый HTML-шаблон, оформленный в виде встроенного документа на CoffeeScript, о которых рассказывалось в главе 2, «Основы».

Сам шаблон чрезвычайно прост. Динамически в шаблон вставляется только заголовок задачи. Изменение флажка, отражающего состояние задачи, будет выполняться за пределами шаблона. Создайте в папке `assets/jquery` новый файл `todo_item.coffee`. В этом файле определите функцию, которая будет добавлять задачу в список, используя только что созданный шаблон.

Пример: (исходный файл: `app.4/assets/jquery/todo_item.coffee`)

```
@TodoApp ||= {}
TodoApp.appendTodo = (todo) ->
    li = $("<li>#{_.template(Templates.list_item_template)(todo)}</li>")
    $('#new_todo').after(li)
```

Здесь в первую очередь создается новая переменная `TodoApp`, которая будет доступна за пределами файла. Добавив символ `@` в объявление переменной `TodoApp`, мы сообщили компилятору CoffeeScript, что переменная должна быть включена в объект `this`, каковым в данном случае является объект `window` в браузере. После включения переменной `TodoApp` в объект `window`, она будет доступна из любой точки программы, где доступен объект `window`.

Далее создается функция, принимающая задачу и выводящая ее на экран с использованием шаблона. Подключив библиотеку underscore.js, мы получили доступ к переменной с именем `_`, а через нее – к функции с именем `template`. Здесь мы вызываем эту функцию и передаем ей HTML-шаблон, а также задачу в формате JSON, которая должна быть доступна в этом шаблоне. Библиотека underscore.js выполнит необходимые операции и вернет требуемый фрагмент разметки HTML. После чего этот фрагмент будет добавлен в список задач, сразу за формой создания новой задачи.

Теперь отредактируйте файл `new_todo_form.coffee`, чтобы задействовать в нем функцию добавления новой задачи в список.

Пример: (исходный файл: `app.4/assets/jquery/new_todo_form.coffee`)

```
$ ->
    # Передать форме фокус ввода сразу после загрузки страницы:
    $('#new_todo .todo_title').focus()
    # Обработка событий keypress в форме создания новой задачи:
    $('#new_todo .todo_title').keypress (e) ->
        # обрабатывается только нажатие клавиши 'enter':
        if e.keyCode is 13
            todo = {title: $(e.target).val()}
            if !todo.title? or todo.title.trim() is ""
                alert "Title can't be blank"
            else
                request = $.post "/api/todos", todo: todo
                request.fail (response) =>
                    message = JSON.parse(response.responseText).message
                    alert message
                request.done (todo) =>
                    TodoApp.appendTodo(todo)
                     $(e.target).val("")
```

В функции `request.done` мы заменили строку, выводившую текст с описанием задачи в формате JSON, получаемую от сервера, на вызов новой функции `TodoApp.appendTodo`.

Перезапустите приложение и добавьте новую задачу. На этот раз задача должна быть добавлена в список с соблюдением оформления.

Вывод списка имеющихся задач

Теперь, когда появилась возможность создавать новые задачи, и реализован их вывод при создании, необходимо обеспечить вывод задач, уже имеющихся в базе данных, при загрузке страницы. Если прямо сейчас добавить несколько задач и затем перезагрузить страницу, создастся впечатление, что эти задачи исчезли. В действительности они живы и здоровы, и находятся в нашей базе данных, но пока не видны, потому что мы еще не реализовали их извлечение и вывод. К счастью, для этого достаточно написать относительно короткий фрагмент кода.

Пример: (исходный файл: `app.5/assets/jquery/retrieve_existing_todos.coffee`)

```
$ ->
    request = $.get('/api/todos')
    request.done (todos) ->
```

```
    for todo in todos.reverse()
      TodoApp.appendTodo(todo)
```

Прежде всего, необходимо дождаться, пока страница будет полностью загружена, и только потом обращаться к серверному API за списком задач. После загрузки страницы создается новый запрос к серверной функции, возвращающей список задач, и к нему подключается функция обратного вызова, которая будет выполнена при получении ответа. В этой функции мы изменяем порядок следования задач на обратный, и затем по одной передаем их функции `appendTodo`. Зачем нужно изменять порядок следования задач? Ответ на этот вопрос настолько же прост, насколько неприятен. В действительности задачи поступают от сервера в правильном порядке. Однако наша функция `appendTodo` добавляет каждую задачу в начало списка, что равносильно изменению порядка следования задач на обратный. Можно было бы изменить принцип действия серверного API, но он как раз работает правильно – неправильно работает клиент. Можно было бы написать другую функцию, добавляющую задачи в правильном порядке, или добавить в функцию `appendTodo` параметр, определяющий порядок добавления, но в конечном итоге текущее решение является самым простым и маловероятно, что впоследствии оно вызовет какие-либо проблемы.

Если теперь перезагрузить страницу, на экране должен появиться список задач, хранящихся в базе данных.

Изменение задач

Получив возможность создавать новые задачи и выводить список имеющихся задач, мы тут же оказываемся перед проблемой отсутствия возможности внесения некоторых изменений в задачи. Наше приложение должно позволять изменять значение атрибута `title` задачи, а также переключать состояние задачи.

Для этого мы добавим две функции. Первая будет следить за флажком и текстовым полем в каждой задаче. При обнаружении изменений в этих элементах интерфейса, она будет вызывать вторую функцию, которая будет отправлять изменения серверному API.

Пример: (исходный файл: `app.6/assets/jquery/watch_todo_for_changes.coffee`)

```
@TodoApp ||= {}
# Слежение за изменениями в задаче:
```

```
TodoApp.watchForChanges = (li, todo) ->
    # Если изменилось состояние флажка:
    $('.todo_state', li).click (e) =>
        TodoApp.updateTodo(li, todo)
    # Если в поле ввода заголовка была нажата клавиша "enter":
    $('.todo_title', li).keypress (e) =>
        if e.keyCode is 13
            TodoApp.updateTodo(li, todo)
```

Функция watchForChanges, как можно заключить из ее имени, следит за элементом li указанной задачи, и при изменении состояния флажка или нажатии на клавишу **Enter** в текстовом поле ввода заголовка задачи вызывает функцию updateTodo. Чтобы обеспечить вызов функции watchForChanges, добавим в функцию appendTodo вызов этой функции при добавлении новой задачи:

Пример: (исходный файл: app.6/assets/jquery/todo_item.coffee)

```
@TodoApp ||= {}
TodoApp.appendTodo = (todo) ->
    li = $("<li>#{_.template(Templates.list_item_template)(todo)}</li>")
    $('#new_todo').after(li)
     TodoApp.watchForChanges(li, todo)
```

Теперь все задачи будут под пристальным наблюдением. Напишем функцию updateTodo, сохраняющую изменения.

Пример: (исходный файл: app.6/assets/jquery/update_todo.coffee)

```
@TodoApp ||= {}
# Изменение задачи:
TodoApp.updateTodo = (li, todo) ->
    todo.title = $('.todo_title', li).val()
    if !todo.title? or todo.title.trim() is ""
        alert "Title can't be blank"
    else
        if $('.todo_state', li).attr('checked')?
            todo.state = 'completed'
        else
            todo.state = 'pending'
        request = $.post "/api/todos/#{todo._id}",
            todo: todo
            _method: 'put'
        request.fail (response) =>
```

```
        message = JSON.parse(response.responseText).message
        alert message
```

Функция `updateTodo` напоминает реализацию добавления новой задачи. Однако в ней есть несколько важных отличий. Во-первых, получив содержимое текстового поля, мы проверяем, не является ли оно пустым. Если проверка прошла успешно, мы добавляем остальные данные, которые требуется отправить обратно на сервер. В данном случае необходимо изменить значение атрибута `state`, исходя из состояния флажка.

Далее создается запрос к серверу. Так как требуется изменить конкретную задачу, в URL запроса требуется включить идентификатор задачи. Наиболее наблюдательные из вас могут заметить, что в действительности данные отправляются на сервер посредством запроса `POST`, тогда как серверный API требует, чтобы отправка выполнялась методом `PUT`. На то есть две причины. Во-первых, исторически не все браузеры поддерживают HTTP-запросы, кроме типов `GET` и `POST`. Во-вторых, то же самое относится и к библиотеке jQuery. Чтобы обойти это ограничение, многие фреймворки, такие как Express и Ruby on Rails, особое внимание уделяют параметру с именем `_method` и устанавливают тип запроса в соответствии со значением этого параметра. В нашем случае мы устанавливаем в параметре `_method` значение `PUT`, поэтому фреймворк Express будет рассматривать этот запрос не как запрос типа `POST`, а как запрос типа `PUT`.

В заключение определяется наличие ошибки, возникшей в ходе изменения задачи. Если ошибка присутствует, она отображается, а противном случае мы просто позволяем пользователю продолжить работу со своими задачами.

У нас осталась еще одна маленькая проблема. Если пометить задачу, как выполненную и обновить страницу, она отобразится, как ожидающая выполнения, потому что флажок состояния пока не проверяется, и при отображении задачи к ней не будет применен соответствующий стиль, помогающий пользователю увидеть, что задача завершена. Напишем функцию, которая будет применять соответствующие стили:

Пример: (исходный файл: `app.7/assets/jquery/style_by_state.coffee`)

```
@TodoApp ||= {}
# Изменяет стиль отображения в соответствии с состоянием задачи:
TodoApp.styleByState = (li, todo) ->
```

```
    if todo.state is "completed"
         $('.todo_state', li).attr('checked', true)
         $('label.active', li).removeClass('active')
         $('.todo_title', li).addClass('completed').attr('disabled', true)
     else
         $('.todo_state', li).attr('checked', false)
         $('label', li).addClass('active')
         $('.todo_title', li).removeClass('completed').attr('disabled',false)
```

Эта функция просто проверяет состояние задачи, и если она помечена как выполненная, к элементам разметки будут применены соответствующие CSS-классы. В противном случае эти стили будут удалены. Одна из прелестей библиотеки jQuery состоит в том, что она позволяет писать подобный программный код, не вынуждая проверять наличие в элементе того или иного CSS-класса.

Если класс уже присутствует в элементе, попытка добавить его просто игнорируется. То же относится и к операции удаления класса – если класс отсутствует в элементе, попытка удалить его игнорируется. Это именно то, что нам нужно.

Теперь, чтобы обеспечить корректное применение стилей, осталось лишь добавить вызов функции `styleByState` в нескольких местах. В первую очередь вызов этой функции необходимо добавить в функцию `appendTodo`. Если существующая задача будет передана функции `appendTodo`, к ней гарантированно будут применены соответствующие стили.

Пример: (исходный файл: `app.7/assets/jquery/todo_item.coffee`)

```
@TodoApp ||= {}
TodoApp.appendTodo = (todo) ->
    li = $("<li>#{_.template(Templates.list_item_template)(todo)}</li>")
    $('#new_todo').after(li)
    TodoApp.watchForChanges(li, todo)
    TodoApp.styleByState(li, todo)
```

Нам также необходимо гарантировать применение соответствующих стилей при изменении задачи. Сделать это можно, подключив функцию обратного вызова к объекту `request` в функции `updateTodo`:

Пример: (исходный файл: `app.7/assets/jquery/update_todo.coffee`)

```
@TodoApp ||= {}
# Изменение задачи:
```

```
TodoApp.updateTodo = (li, todo) ->
    todo.title = $('.todo_title', li).val()
    if !todo.title? or todo.title.trim() is ""
        alert "Title can't be blank"
    else
        if $('.todo_state', li).attr('checked')?
            todo.state = 'completed'
        else
            todo.state = 'pending'
        request = $.post "/api/todos/#{todo._id}",
            todo: todo
            _method: 'put'
        request.fail (response) =>
            message = JSON.parse(response.responseText).message
            alert message
        request.done (todo) ->
            TodoApp.styleByState(li, todo)
```

Удаление задач

Приложение практически закончено. Нам осталось лишь связать кнопку удаления с обработчиком, чтобы пользователи могли удалять ненужные больше задачи. Сейчас вы уже и сами без труда напишете необходимый программный код, тем не менее, коротко рассмотрим его.

Начнем с функции `deleteTodo`:

Пример: (исходный файл: `final/assets/jquery/delete_todo.coffee`)

```
@TodoApp ||= {}
# Удаление задачи:
TodoApp.deleteTodo = (li, todo) ->
    if confirm "Are you sure?"
        request = $.post "/api/todos/#{todo._id}", _method: 'delete'
        request.done =>
                li.remove()
```

Функция `deleteTodo` отправляет запрос серверному API, выполняя HTTP-запрос типа `DELETE`, посредством специального параметра `_method`, представленного выше. Если запрос на удаление задачи увенчался успехом, выбранная задача удаляется из списка на странице – все просто и понятно.

Теперь осталось подключить обработчик к кнопке удаления, и приложение будет закончено. Сделать это можно в функции `watchForChanges`, которую мы написали ранее:

Пример: (исходный файл: `final/assets/jquery/watch_todo_for_changes.coffee`)

```
@TodoApp ||= {}
# Слежение за изменениями в задаче:
TodoApp.watchForChanges = (li, todo) ->
    # Если изменилось состояние флажка:
    $('.todo_state', li).click (e) =>
        TodoApp.updateTodo(li, todo)
    # Если в поле ввода заголовка была нажата клавиша "enter":
    $('.todo_title', li).keypress (e) =>
        if e.keyCode is 13
            TodoApp.updateTodo(li, todo)
    $('button.danger', li).click (e) =>
        e.preventDefault()
        TodoApp.deleteTodo(li, todo)
```

Вот и все! Приложение готово! Поздравляю.

В заключение

Мы получили, что хотели. С помощью библиотеки jQuery мы создали интерактивного веб-клиента для приложения управления списком задач. Это оказалось совсем несложно, и в процессе разработки вы смогли увидеть, как на языке CoffeeScript можно писать очень выразительный программный код, использующий библиотеку jQuery.

Подход, предпринятый в этой главе, с точки зрения «архитектуры» программного кода, является, пожалуй, самым распространенным в среде разработчиков, использующих библиотеку jQuery. Мы написали пакет функций, обменивающихся некоторыми объектами и выполняющих необходимые операции. Для создания этого приложения можно было бы выбрать другой подход, основанный на создании классов, управляющих задачами и более аккуратно обертывающих задачи HTML-элементами и собственными обработчиками событий.

Почему я не продемонстрировал второй подход? На то есть две причины. Во-первых, как я уже говорил, выбранный здесь подход

является самым распространенным среди тех, кто пишет простые приложения на JavaScript с применением библиотеки jQuery, поэтому мне хотелось продемонстрировать, как будет выглядеть подобный программный код на языке CoffeeScript. Вторая причина, почему я не стал использовать стиль программирования, основанный на «классах», заключается в том, что я не хотел повторно изобретать колесо, то есть Backbone.js.

Backbone.js – это простой фреймворк, позволяющий создавать представления, подключенные к элементам страницы, связывать эти представления с моделями, такими как наши задачи, легко обеспечивать взаимодействия между ними и их реакцию на события. А действительно, почему бы не посмотреть на фреймворк Backbone в действии? Переверните страницу, перейдите к главе 12, «Пример: список задач, часть 3 (клиент на основе Backbone.js)», и приступим!

Кстати, для тех, кому любопытно, как выглядел бы программный код клиента на основе библиотеки jQuery, выбери мы подход с использованием классов, я уже написал его. Пользуйтесь! [12]

Примечания

1. http://twitter.github.com/bootstrap/.
2. http://twitter.com.
3. https://github.com/trevorBurnham/connect-assets.
4. http://ru.wikipedia.org/wiki/John_Resig.
5. http://ru.wikipedia.org/wiki/Jquery.
6. http://documentcloud.github.com/underscore.
7. https://github.com/sstephenson/eco.
8. http://www.handlebarsjs.com/.
9. https://github.com/janl/mustache.js/.
10. http://jade-lang.com/.
11. https://github.com/jquery/jquery-tmpl.
12. https://github.com/markbates/Programming-In-CoffeeScript/tree/master/todo2/alt-final.

12. Пример: список задач, часть 3 (клиент на основе Backbone.js)

В главе 11, «Пример: список задач, часть 2 (клиент на основе jQuery)» мы написали клиентскую часть приложения для управления списком задач, задействовав библиотеку jQuery. В конце главы я говорил, что клиентскую часть можно было бы написать иначе, но для этого пришлось бы практически повторить фреймворк Backbone.js [1]. Поскольку обычно я предпочитаю не изобретать колесо, я подумал, что вместо этого можно было бы посмотреть, как будет выглядеть приложение, если вместо библиотеки jQuery использовать в нем фреймворк Backbone.

Что такое Backbone.js?

Backbone – это клиентский MVC-фреймворк [2], написанный на языке JavaScript *Джереми Ашкенасом* (Jeremy Ashkenas), [3] создателем языка программирования, известного под названием CoffeeScript. [4] Backbone позволяет писать высокодинамичные клиентские приложения на JavaScript или, как в нашем случае, на CoffeeScript.

Фреймворк Backbone состоит из трех отдельных частей. Первая – слой представлений. Представления дают возможность отображать элементы на экране и следить за изменениями в этих элементах, реагируя соответствующим образом. Представления могут также получать события от других объектов и обновлять себя соответственно этим событиям.

Вторая часть фреймворка Backbone – модели и коллекции. Модель отображается на единственный экземпляр объекта. В нашем случае, такой моделью может быть модель задачи `Todo`. Объект модели может взаимодействовать с хранилищем данных, чтобы обеспечить собственное сохранение. Он может также содержать другие функции, упрощающие работу с объектом, такие как функция объединения имени и фамилии в единую строку. Коллекция – это именно то, что предполагается из названия – коллекция объектов модели.

То есть, в данном приложении это может быть коллекция объектов модели `Todo`. Коллекции в Backbone также способны взаимодействовать с хранилищем данных, посредством API, подобного нашему.

Кроме того, модели и коллекции могут генерировать самые разнообразные события, при выполнении различных операций с этими объектами. Например, когда в коллекцию добавляется новый экземпляр модели, коллекция возбуждает событие `add`. Эти события могут перехватываться другими объектами, такими как представления. Например, представление может следить за событием `add` в коллекции и по этому событию отображать экземпляр модели на экране. Чуть ниже будет показано, как действует весь этот механизм.

Последняя часть фреймворка Backbone – маршрутизатор. Маршрутизаторы в Backbone позволяют следить и откликаться на изменение URL в браузере. При изменении URL, когда в маршрутизаторе отыскивается соответствующий вариант маршрута, выполняется программный код, связанный с этим маршрутом. Это напоминает нашу серверную часть приложения на основе фреймворка Express. Мы не будем использовать маршрутизаторы в этой главе, но это не означает, что они бесполезны. Просто в нашем простейшем приложении в их использовании нет необходимости.

Вот и все. Конечно, это лишь очень краткий обзор фреймворка Backbone. На протяжении всей этой главы мы будем знакомиться с дополнительными подробностями об этом фреймворке, но она не является полноценным руководством по использованию Backbone, потому что мы будем касаться лишь тех частей фреймворка, которые используются в приложении.

Подготовка

Прежде чем приступить к перестройке приложения, необходимо сначала навести в нем порядок, и приготовить к внедрению фреймворка Backbone.

Первое, что следует сделать, – удалить папку `assets/jquery`.

Второе – удалить ссылки на этот каталог из файла `assets/application.coffee`:

Пример: (исходный файл: `app.1/assets/application.coffee`)

```
#= require "templates"
```

Вот и все! Если теперь перезапустить приложение, на экране останется только форма создания новой задачи, которая не будет выполнять никаких действий. Начнем перестройку приложения.

Настройка фреймворка Backbone.js

Внедрение фреймворка в приложение выполняется практически безболезненно. Для этого достаточно будет удовлетворить всего одну зависимость: underscore.js. [5] Несмотря всего на одну «тяжелую» зависимость, фреймворк не имеет большой практической пользы без библиотеки, реализующей операции с деревом DOM или обеспечивающей поддержку технологии AJAX, такой как библиотека jQuery (или Zepto [6]). К счастью обе библиотеки, jQuery и underscore.js, уже подключены в файле `index.ejs`, поэтому нам остается только добавить сам фреймворк Backbone:

Пример: (исходный файл: `app.1/src/views/index.ejs`)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todos</title>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
↪jquery.min.js" type="text/javascript"></script>
        <script src="http://documentcloud.github.com/underscore/
↪underscore-min.js" type="text/javascript"></script>
        <script src="http://documentcloud.github.com/backbone/
↪backbone-min.js" type="text/javascript"></script>
        <%- js('/application') %>
        <link rel="stylesheet" href="http://twitter.github.com/bootstrap/
↪1.4.0/bootstrap.min.css">
        <%- css('/application') %>
    </head>
    <body>
        <div class="container">
            <h1>Todo List</h1>
            <ul id='todos' class='unstyled'>
                <li id='new_todo'>
                    <div class="clearfix">
                        <div class="input">
                            <div class="input-prepend">
                                <span class='add-on'>New Todo</span>
                                    <input class="xlarge todo_title"
↪size="50" type="text" placeholder="Enter your new Todo here..." />
                            </div>
                        </div>
                    </div>
                </li>
```

```
            </ul>
        </div>
    </body>
</html>
```

Вот и все, что нужно было сделать для внедрения Backbone в приложение. Однако я добавлю еще один файл. Я не собирался добавлять его до того момента, когда мы начнем создавать модель `Todo`, но похоже, что лучше сразу выполнить всю подготовительную работу.

Когда Backbone будет взаимодействовать с сервером, по умолчанию он будет отправлять данные в таком виде:

```
{title: 'My New Todo'}
```

Но, если вы помните, наш API ожидает получить данные с указанием пространства имен, как показано ниже:

```
todo: {title: 'My New Todo'}
```

Чтобы реализовать эту особенность, мы «позаимствуем» фрагмент из файла `backbone-rails` [7], входящего в состав фреймворка Ruby on Rails, где уже реализовано необходимое нам исправление для Backbone. Этот файл приводится ниже:

Пример: (исходный файл: `app.2/assets/backbone_sync.js`)

```
// Взято из файла https://github.com/codebrew/backbone-rails.
// Включает данные в формате JSON в пространство имен модели
// перед отправкой данных на сервер.
// Например: {todo: {title: 'Foo'}}
(function() {
    var methodMap = {
        'create': 'POST',
        'update': 'PUT',
        'delete': 'DELETE',
        'read' : 'GET'
    };
    var getUrl = function(object) {
        if (!(object && object.url)) return null;
        return _.isFunction(object.url) ? object.url() : object.url;
    };
    var urlError = function() {
        throw new Error("A 'url' property or function must be specified");
```

```
    };
    Backbone.sync = function(method, model, options) {
        var type = methodMap[method];
        // Параметры по умолчанию для JSON-запроса.
        var params = _.extend({
            type: type,
            dataType: 'json',
            beforeSend: function( xhr ) {
                var token = $('meta[name="csrf-token"]').attr('content');
                if (token) xhr.setRequestHeader('X-CSRF-Token', token);
                model.trigger('sync:start');
            }
        }, options);
        if (!params.url) {
            params.url = getUrl(model) || urlError();
        }
        // Проверить наличие данных запроса.
        if (!params.data && model && (method == 'create' ||
↪method == 'update')){
            params.contentType = 'application/json';
            var data = {}
            if(model.paramRoot) {
                data[model.paramRoot] = model.toJSON();
            } else {
                data = model.toJSON();
            }
            params.data = JSON.stringify(data)
        }
        // Не обрабатывать данные в запросах, отличных от GET.
        if (params.type !== 'GET') {
            params.processData = false;
        }
        // Сгенерировать событие окончания синхронизации
        var complete = options.complete;
        options.complete = function(jqXHR, textStatus) {
            model.trigger('sync:end');
            if (complete) complete(jqXHR, textStatus);
        };
        // Выполнить запрос.
        return $.ajax(params);
    }
}).call(this);
```

Я не жду, что вы сразу поймете, что делает этот фрагмент, в особенности если учесть, что о моделях пока ничего не говорилось, но вы можете поверить мне на слово, что этот фрагмент сделает нашу жизнь проще и краше. Поэтому нам остается только принять факт его существования и быть благодарными за это. Чтобы добавить этот файл в приложение, необходимо сначала поместить его в каталог `assets` под именем `backbone_sync.js`, а затем подключить его в файле `assets/application.coffee`:

Пример: (исходный файл: `app.2/assets/application.coffee`)

```
#= require "backbone_sync"
#= require "templates"
#= require_tree "models"
```

Теперь все готово и можно начинать использовать Backbone!

Создание модели Todo и коллекции ее экземпляров

Первой частью этого приложения является модель `Todo`. Данная модель представляет отдельные задачи, которые мы будем получать с сервера.

Создайте в каталоге `assets` новую папку с именем `models`. В эту папку мы поместим файлы с определениями модели `Todo` и коллекции `Todos`.

Пример: (исходный файл: `app.2/assets/models/todo.coffee`)

```
# Модель Todo для клиента на основе Backbone:
class @Todo extends Backbone.Model
    # пространство имен 'todo' для данных в формате JSON,
    # см. backbone_sync.js
    paramRoot: 'todo'
    # Сконструировать URL, добавив в конец _id, если имеется:
     url: ->
          u = "/api/todos"
          u += "/#{@get("_id")}" unless @isNew()
          return u
     # По умолчанию функция isNew из фреймворка Backbone ищет строку 'id',
     # Mongoose возвращает "_id", поэтому необходимо исправить функцию:
     isNew: ->
            !@get("_id")?
```

При создании новых моделей для фреймворка Backbone, важно помнить о необходимости наследовать класс `Backbone.Model`, в противном случае модель лишится функциональности, присущей моделям Backbone.

Поскольку мы добавили в приложение файл `backbone_sync.js`, необходимо указать имя, под которым наши данные будут отправляться на сервер. Для этого мы присвоили атрибуту `paramRoot` значение `todo`.

Затем необходимо сообщить фреймворку URL, который должен использоваться для взаимодействия с сервером. Для этого мы создали функцию `url`. Backbone автоматически будет отыскивать эту функцию и с ее помощью определять, где находится наш API. Когда создается новый объект `Todo`, он не имеет идентификатора, поэтому идентификатор следует добавлять, только если объект не является новым. Функция `isNew`, встроенная в Backbone, возвращает `true` или `false`, в зависимости от того, является ли объект «новым» или нет.

> **Совет.** Чтобы получить значение атрибута модели Backbone, такого как `title` или `_id`, следует использовать функцию `get`. Это обусловлено тем, что все атрибуты моделей Backbone хранятся в переменной с именем `attributes`, с целью предотвратить любые конфликты между атрибутами и функциями, реализованными в Backbone, и вашими собственными.

Функция `isNew` в Backbone просто пытается отыскать атрибут `id` в объекте. Если ей это удается, объект считается не новым. К сожалению, MongoDB [8] вместо атрибута `id` возвращает атрибут `_id`. Поэтому нам потребовалось переписать функцию `isNew`, чтобы учесть эту особенность.

> **Совет.** Подобно функции `isNew`, нам пришлось написать собственную функцию `url`, потому что вместо атрибута `id` MongoDB использует атрибут `_id`. Если бы Backbone использовал атрибут `_id`, нам достаточно было бы определить в модели `Todo` атрибут `url` (не функцию) со значением `/api/todos`, а фреймворк Backbone автоматически добавлял бы в него значение атрибута `_id`. Но все есть так как есть.

Закончив определение модели `Todo`, можно приступать к определению коллекции `Todos`. Как упоминалось выше, коллекция – это список, напоминающий массив, хранящий множество экземпляров модели `Todo`. В данном приложении коллекция `Todos` будет использоваться для получения всех задач, имеющихся в базе данных.

Совет. Лично мне не очень нравится необходимость определять отдельный класс для управления коллекцией экземпляров модели, но это не слишком высокая цена за функциональность, которую несут в себе коллекции. Остается лишь привыкнуть жить с этим. Дополнительно замечу, что я обычно помещаю определение класса коллекции в один файл с определением модели. Это упрощает внесение возможных изменений в будущем.В данном случае я поместил их в отдельные файлы, чтобы проще было демонстрировать их в книге.

Определение класса коллекции `Todos` выглядит очень просто:

Пример: (исходный файл: `app.2/assets/models/todos.coffee`)

```
# Коллекция Todos для клиента на основе Backbone:
class @Todos extends Backbone.Collection
    model: Todo
    url: "/api/todos"
```

Класс `Todos` должен наследовать класс `Backbone.Collection`, чтобы получить всю необходимую функциональность. После этого в коллекции остается определить всего два атрибута.

Первому атрибуту, `model`, присваивается класс `Todo`. Этот атрибут сообщает коллекции, что при получении данных от сервера или из другого места, эти данные должны быть преобразованы в объекты `Todo`.

Второй атрибут – `url`. Поскольку это коллекция, нам не приходится беспокоиться о каких-либо идентификаторах в URL. Поэтому достаточно определить простой атрибут.

На этом определение коллекции `Todos` закончено.

Отредактируйте файл `assets/application.coffee` и добавьте в него подключение созданного нами каталога `models`:

Пример: (исходный файл: `app.2/assets/application.coffee`)

```
#= require "backbone_sync"
#= require "templates"
#= require_tree "models"
```

Если по характеру вы похожи на меня, значит вам тоже не терпится увидеть свою работу в действии. Хорошо, давайте быстренько задействуем коллекцию `Todos` и модель `Todo` для получения списка задач от сервера и его вывода с помощью шаблона, созданного в предыдущей главе.

Сделать это можно, добавив всего несколько строк в файл assets/application.coffee, как показано ниже:

Пример: (исходный файл: app.3/assets/application.coffee)

```
#= require "backbone_sync"
#= require "templates"
#= require_tree "models"
$ ->
    template = _.template(Templates.list_item_template)
    todos = new Todos()
    todos.fetch
        success: ->
            todos.forEach (todo) ->
                  $('#todos').append("<li>#{template(todo.toJSON())}</li>")
```

Новый экземпляр шаблона создается сразу после загрузки дерева DOM, поэтому его можно будет использовать для отображения задач после их получения.

Затем создается новый экземпляр коллекции Todos и ссылка на нее сохраняется в переменной todos.

После подготовки экземпляра коллекции Todos, можно вызвать функцию fetch. Для получения списка задач с сервера, эта функция использует атрибут url, созданный нами в коллекции Todos. Если попытка извлечения задач увенчалась успехом, функция fetch вызывает переданную ей функцию обратного вызова success.

Функция success выполняет итерации по полученным с сервера объектам задач в списке моделей Todo, используя функцию forEach, отображает каждую задачу с использованием шаблона и выводит их на экран.

В результате всех этих действий, после перезагрузки страницы вы должны увидеть на экране привлекательно оформленный список задач. Не надо пока пытаться изменять или удалять задачи. Мы сделаем это позже. В следующем разделе мы создадим первое представление Backbone взамен только что написанного программного кода для отображения задач.

Вывод списка задач с помощью представления

Программный код, написанный в предыдущем разделе, прекрасно справляется со своей задачей, но его можно сделать еще более простым и гибким. А поможет нам в этом класс Backbone.View. Заменим код, написанный выше, классом Backbone.View.

Сначала создайте каталог `views` в папке `assets`. Здесь будут находиться файлы с определениями представлений. Создайте в этой папке файл `todo_list_view.coffee` со следующим содержимым:

Пример: (исходный файл: `app.4/assets/views/todo_list_view.coffee`)

```
# 'Главное' представление для приложения на основе Backbone
class @TodoListView extends Backbone.View
    el: '#todos'
    initialize: ->
        @template = _.template(Templates.list_item_template)
        @collection.bind("reset", @render)
        @collection.fetch()
    render: =>
        @collection.forEach (todo) =>
                $(@el).append("<li>#{@template(todo.toJSON())}</li>")
```

Какие операции выполняет этот код? Хороший вопрос! Прежде всего здесь создается новый класс `TodoListView`, наследующий класс `Backbone.View`. Благодаря наследованию класса `Backbone.View` он получает доступ к множеству особенностей, которые будут использоваться на протяжении оставшейся части этой главы.

> **Совет.** Обратите внимание, что подобно классам `Todo` и `Todos`, имя определяемого класса `TodoListView` предваряется символом `@`. Символ `@` обеспечивает доступность класса за пределами функции-обертки, автоматически создаваемой компилятором CoffeeScript. Если этого не сделать, класс будет недоступен за пределами файла, в котором он определен.

Далее определяется, что с этим представлением будут связаны элементы страницы `#todos`. Делается это посредством присваивания значения атрибуту `el`. Если этого не сделать, фреймворк Backbone создаст новый объект `div` для атрибута `el`, и вам вручную придется вставлять этот элемент в страницу. Как действует этот механизм, мы увидим чуть ниже.

Теперь мы подошли к функции `initialize`. Функция `initialize` – это особая функция. Она вызывается фреймворком Backbone после инициализации объекта представления. Едва ли кто-то захочет писать функции `constructor` в своих классах представлений, так как при этом можно потерять все шоколадное богатство, которое Backbone старается создать для вас. Если вам потребуется выполнить

некоторые операции в момент создания экземпляра представления, используйте для этого функцию initialize.

Как это обычно происходит, нам необходимо предусмотреть выполнение нескольких операций при создании экземпляра класса TodoListView. В частности, требуется выполнить несколько операций с объектом коллекции @collection. Первым вашим вопросом должен быть: «Откуда берутся все эти переменные?». В Backbone имеется несколько «магических» переменных и атрибутов, в том числе переменные @collection и @model. Через минуту вы увидите, что при создании экземпляра класса TodoListView, ему передается объект, содержащий ключ collection, значением которого является новая коллекция Todos. Ссылка на эту коллекцию присваивается объекту @collection в TodoListView, обеспечивающему доступ к коллекции Todos. Вскоре вы увидите все изменения в файле application.coffee, и тогда многое для вас прояснится.

Что следует сделать с объектом @collection, также известным, как коллекция Todos? Прежде всего, нужно вызвать функцию bind и сообщить ей, что всякий раз, когда коллекция будет генерировать событие reset, в ответ должна вызываться функция @render экземпляра класса TodoListView.

Как объект коллекции будет генерировать событие reset? В Backbone обычно принято генерировать это событие в функции fetch. Как было показано выше, функция fetch получает полный список задач с сервера. Для получения этого списка мы вызываем функцию fetch в последней строке в функции initialize. Вызов функции fetch, в свою очередь, сгенерирует событие reset, для обработки которого будет вызвана функция @render.

Функция @render служит для вывода списка задач, хранящихся в коллекции. Она не слишком отличается от первоначального программного кода в файле application.coffee, отображающего задачи на экране. Основное отличие заключается в том, что на этот раз функция forEach применяется непосредственно к объекту @collection, минуя вызов функции обратного вызова success. Другое отличие – отпала необходимость ссылаться непосредственно на элемент #todos, вместо него используется атрибут @el, хранящий ссылку на требуемый элемент. Использование атрибута @el вместо непосредственного имени элемента избавит нас от утомительного труда, если позднее нам придется провести реорганизацию программного кода. Достаточно будет изменить значение одного атрибута @el и не копаться по всему программному коду.

Совет. В объявлении функции `@render` вместо оператора `->` использован оператор `=>`, чтобы эта функция смогла получить свой контекст после возбуждения события `reset` и возможность обращаться к остальным элементам класса. Если бы здесь использовался оператор `->`, этот код порождал бы ошибку, такую как `TypeError: 'undefined' is not an object` (TypeError: 'undefined' не является объектом), в инструкции `'this.collection.forEach'`, из-за недоступности объекта `@collection` в функции `@render`. Если бы мы были вынуждены использовать оператор `->`, мы могли бы вручную связать функцию с контекстом внутри функции `initialize`, вызвав функцию `bindAll` из библиотеки Underscore. `_.bindAll(@, "render")`. Вместо всего этого я просто использовал оператор `=>`.

Теперь осталось лишь отредактировать файл `application.coffee`, чтобы вместо старого программного кода задействовать в нем новый класс `TodoListView`:

Пример: (исходный файл: `app.4/assets/application.coffee`)

```
#= require "backbone_sync"
#= require "templates"
#= require_tree "models"
#= require_tree "views"
$ ->
    # Запуск приложения на основе Backbone.js:
    new TodoListView(collection: new Todos())
```

Как видно в примере, сначала необходимо подключить, каталог `views`, чтобы подхватить все представления, находящиеся там. Затем, после загрузки страницы, остается лишь создать новый экземпляр класса `TodoListView`, передав ему новый экземпляр коллекции `Todos`. На этом мы закончили работу с файлом `application.coffee` до конца главы.

Создание новых задач

Реализовав вывод существующих задач на экран, можно перейти к подключению формы, чтобы получить возможность *создания новых задач*. Для этого нам потребуется представление для управления формой и обработки события нажатия клавиши **Enter** при вводе новой задачи, чтобы сохранить ее на сервере и отобразить на экране.

Определим новый класс `NewTodoView`, как показано ниже:

Пример: (исходный файл: app.5/assets/views/new_todo_view.coffee)

```
# Представление, обрабатывающее создание новой задачи:
class @NewTodoView extends Backbone.View
    el: '#new_todo'
    events:
        'keypress .todo_title': 'handleKeypress'
    initialize: ->
        @collection.bind("add", @resetForm)
        @$('.todo_title').focus()
    handleKeypress: (e) =>
        if e.keyCode is 13
            @saveModel(e)
    resetForm: (todo) =>
        @$('.todo_title').val("")
    saveModel: (e) =>
        e?.preventDefault()
        model = new Todo()
        model.save {title: @$('.todo_title').val()},
            success: =>
                @collection.add(model)
            error: (model, error) =>
                if error.responseText?
                    error = JSON.parse(error.responseText)
                alert  error.message
```

Определение этого класса получилось немного длиннее, чем определение класса TodoListView выше, однако большую часть составляет функция saveModel, которая, к настоящему моменту, для вас должна быть простой и понятной. Тем не менее, мы обсудим ее чуть ниже.

Сам класс TodoListView требуется связать с элементом #new_todo страницы, для чего снова используется атрибут el.

Далее, необходимо сообщить классу NewTodoView, что он должен принимать некоторые события и откликаться на них. Фреймворк Backbone позволяет легко сделать это с помощью атрибута events объекта. Однако отображение событий с помощью атрибута events выглядит несколько замысловатым. Ключом в определении отображения события является составной ключ. Первая его часть – имя события, такое как click, submit, keypress и так далее, за которым следует CSS-селектор элемента, за событиями в котором требуется организовать наблюдение. Значением отображения является

функция, которая должна вызываться по событию. В данном случае реализуется наблюдение за событием keypress в элементе .todo_title, которое должно обрабатываться функцией handleKeypress.

> **Совет.** Необходимо сделать два важных замечания, касающихся отображения событий в Backbone. Первое: CSS-селектор действует в области видимости атрибута el. Второе: в качестве функции передается строка с ее именем, а не ссылка на функцию, как это делалось при привязывании обработчиков событий в коллекции. Я не знаю причину такого разногласия, но, как есть, так есть. Это первое, на что следует обратить внимание, если представление работает не так, как ожидается.

В функции initialize необходимо связать функцию resetForm с событием add, возникающим в объекте @collection, который передается функции-конструктору при создании экземпляра класса NewTodoView. Позднее, в функции saveModel, после получения от сервера подтверждения успешного создания новой задачи, она будет добавлена в коллекцию @collection. Это вызовет событие add, которое в свою очередь приведет к вызову функции resetForm. Функция resetForm, как видно в примере, сбрасывает форму в исходное состояние.

Кроме того, в функции initialize необходимо после загрузки страницы установить фокус ввода в элемент .todo_title формы. Для этого здесь используется специальная функция класса Backbone.View, @$. Она позволяет передавать ей CSS-селекторы, записанные в стиле библиотеки jQuery, которые будут ограничены областью элемента @el, находящегося под наблюдением представления. Не будь этой функции, эту операцию можно было бы реализовать, например, с помощью такой инструкции: $(#new_todo .todo_title'), дающей доступ к тому же элементу.

Функция handleKeypress должна выглядеть для вас знакомой. Она проверяет, была ли нажата клавиша **Enter** и вызывает функцию @saveModel, выполняющую всю тяжелую работу по сохранению экземпляра модели на сервере.

> **Совет.** Экземпляр модели можно было отправить на сервер непосредственно в функции handleKeypress, однако в текущей реализации, если позднее придется добавить кнопку сохранения новой задачи, нам достаточно будет просто связать кнопку с функцией saveModel, в атрибуте events, и избежать дублирования программного кода.

Функция saveModel мало чем отличается от функций, написанных ранее и выполняющих подобные операции. Сначала она создает новый экземпляр класса Todo. Затем создает список атрибутов, в данном случае в список включается один атрибут title, и передает его функции save вместе с функциями обратного вызова success и error.

> **Совет.** Почему я записал e?.preventDefault() вместо e.preventDefault()? Причина в том, что иногда может потребоваться вызвать эту функцию, не передавая ей объект события. Оператор существования гарантирует, что функция preventDefault будет вызвана, только если данная функция получит объект события. Отличный прием, достойный, чтобы взять его на вооружение.

Настоящее волшебство происходит в функции success, когда сервер присылает подтверждение, что задача была благополучно сохранена в базе данных. Она вызывает функцию add объекта @collection и передает ей вновь созданную задачу. Когда это происходит, вызывается функция resetForm, потому что мы сообщили объекту @collection о необходимости вызывать эту функцию, когда будет происходить событие add.

Если теперь запустить приложение и попробовать сделать это, ничего не произойдет, потому что мы еще не создали новый экземпляр класса NewTodoView. Чтобы сделать это, добавьте в класс TodoListView создание нового экземпляра класса NewTodoView, а также обработку новых задач, добавленных в коллекцию @collection, и их отображение на странице.

Пример: (исходный файл: app.5/assets/views/todo_list_view.coffee)

```
# 'Главное' представление для приложения на основе Backbone
class @TodoListView extends Backbone.View
    el: '#todos'
    initialize: ->
        @template = _.template(Templates.list_item_template)
        @collection.bind("reset", @render)
        @collection.fetch()
        @collection.bind("add", @renderAdded)
        new NewTodoView(collection: @collection)
    render: =>
        @collection.forEach (todo) =>
            $(@el).append("<li>#{@template(todo.toJSON())}</li>")
    renderAdded: (todo) =>
        $("#new_todo").after("<li>#{@template(todo.toJSON())}</li>")
```

В функцию `initialize` класса `TodoListView` мы добавили две строки. Первая из них добавляет еще один обработчик события `add`, возникающего в объекте `@collection`, на этот раз предписывается вызывать функцию `renderAdded` класса `TodoListView`.

Вторая строка, добавленная в функцию `initialize` класса `TodoListView`, создает новый экземпляр класса `NewTodoView` и передает ему объект `@collection`.

Функция `renderAdded` в классе `TodoListView` будет вызываться при добавлении новой задачи в объект `@collection` и получать вновь созданную задачу. Получив новую задачу, мы легко можем добавить ее в список задач на странице.

> **Совет.** Все это можно было бы реализовать внутри функции `render` и избавиться от функции `renderAdded`. Однако есть ряд причин, почему я не сделал этого. Во-первых, на отображение всего списка требуется намного больше времени, чем на добавление единственной задачи в список на экране. Во-вторых, функция `render` увеличилась бы в размерах, из-за необходимости очищать список задач, уже отображаемых на странице, чтобы избежать появления повторяющихся записей.

Представление для отображения отдельной задачи

Наше приложение проделало длинный путь, с того момента, как мы выбросили весь программный код, использующий библиотеку jQuery, написанный нами в главе 11, «Пример: список задач, часть 2 (клиент на основе jQuery)», но мы еще далеки от завершения. Приложение позволяет создавать новые задачи и отображает их на странице, но оно пока не позволяет редактировать и удалять задачи. Но, прежде чем приступать к реализации этих операций, следует немного почистить уже написанный программный код. Нам необходимо *представление, которое будет управлять отдельными задачами* на странице, чтобы можно было следить за изменениями в них, такими как изменение заголовка или признака завершенности, а также удалять задачи.

Прямо сейчас добавим в наш код использование нового представления `TodoListItemView`, предназначенного для отображения отдельных задач на странице. Но прежде создадим следующий класс:

Пример: (исходный файл: app.6/assets/views/todo_list_item_view.coffee)

```
# Представление для каждой задачи в списке:
class @TodoListItemView extends Backbone.View
    tagName: 'li'
    initialize: ->
        @template = _.template(Templates.list_item_template)
        @render()
    render: =>
        $(@el).html(@template(@model.toJSON()))
        if @model.get('state') is "completed"
            @$('.todo_state').attr('checked', true)
            @$('label.active').removeClass('active')
            @$('.todo_title').addClass('completed').attr('disabled', true)
        return @
```

Представление TodoListItemView не является отображением существующего элемента страницы, поэтому здесь не требуется устанавливать атрибут el. По умолчанию в классе Backbone.View атрибут el соответствует тегу div. Однако в нашем случае необходимо связать представление с тегом li. Для этого можно воспользоваться атрибутом tagName.

Функция initialize класса TodoListItemView присваивает переменной @template шаблон, который должен использоваться для отображения задачи на экране. Затем вызывается функция @render.

Функция записывает в разметку HTML элемента @el, в данном случае – элемента li, шаблон, в котором используется объект @model. Откуда взялся этот объект? Объект @model передается при создании нового экземпляра класса TodoListItemView, как будет показано ниже.

После отображения шаблона в функции render необходимо внести изменения в разметку HTML, чтобы обеспечить корректное оформление задач, помеченных как выполненные.

В заключение функция render возвращает экземпляр класса TodoListItemView. Это не является необходимостью, Это обычное соглашение о возвращаемых значениях в мире Backbone, позволяющее легко составлять цепочки вызовов методов объектов.

Закончив определение класса TodoListItemView, можно задействовать его в классе TodoListView и убрать из него реализацию отображения шаблона.

Пример: (исходный файл: app.6/assets/views/todo_list_view.coffee)

```
# 'Главное' представление для приложения на основе Backbone
class @TodoListView extends Backbone.View
    el: '#todos'
    initialize: ->
        @collection.bind("reset", @render)
        @collection.fetch()
        @collection.bind("add", @renderAdded)
        new NewTodoView(collection: @collection)
    render: =>
        @collection.forEach (todo) =>
            $(@el).append(new TodoListItemView(model: todo).el)
    renderAdded: (todo) =>
       $("#new_todo").after(new  TodoListItemView(model:  todo).el)
```

В обеих функциях класса TodoListView, render и renderAdded, удалось заменить разметку HTML с обернутым ею шаблоном, созданием нового экземпляра класса TodoListItemView и получением значения атрибута el этого экземпляра. Напомню, что атрибут el содержит разметку HTML, сконструированную в функции render класса TodoListItemView.

Если теперь перезапустить приложение, на экране должен появиться список задач, в котором все завершенные задачи должны иметь соответствующее оформление.

Изменение и проверка моделей в представлениях

Теперь, создав представление TodoListItemView для управления каждой задачей на странице, мы получили место, куда можно добавить логику проверки наличия изменений в задачах и реализацию соответствующей реакции на изменения. Начнем с проверки наличия изменений. В каждой задаче пользователь может изменить две ее характеристики. Он может изменить заголовок или изменить состояние флажка, тем самым изменяя состояние задачи. Об удалении задач будет рассказываться в следующем разделе.

Пример: (исходный файл: app.7/assets/views/todo_list_item_view.coffee)

```
# Представление для каждой задачи в списке:
class @TodoListItemView extends Backbone.View
    tagName: 'li'
    events:
        'keypress .todo_title': 'handleKeypress'
        'change .todo_state': 'saveModel'
    initialize: ->
        @template = _.template(Templates.list_item_template)
        @model.bind("change", @render)
        @model.bind("error", @modelSaveFailed)
        @render()
    render: =>
        $(@el).html(@template(@model.toJSON()))
        if @model.get('state') is "completed"
            @$('.todo_state').attr('checked', true)
            @$('label.active').removeClass('active')
            @$('.todo_title').addClass('completed').attr('disabled', true)
        return @
    handleKeypress: (e) =>
        if e.keyCode is 13
            @saveModel(e)
    saveModel: (e) =>
        e?.preventDefault()
        attrs = {title: @$('.todo_title').val()}
        if @$('.todo_state').attr('checked')?
            attrs.state = 'completed'
        else
            attrs.state = 'pending'
        @model.save attrs
    modelSaveFailed: (model, error) =>
        if error.responseText?
            error = JSON.parse(error.responseText)
        alert error.message
       @$('.todo_title').val(@model.get('title'))
```

Первое, что необходимо сделать, – добавить несколько обработчиков событий. Первый – для события `keypress`, возникающего в поле `.todo_title`. По аналогии с классом `NewTodoView`, по этому событию будет вызываться функция `handleKeypress`, проверяющая нажатие клавиши **Enter**, и вызывающая в этом случае функцию `saveModel`. Нам также необходимо обработать событие `change`, возникающее в элементе `.todo_state`. По любому изменению флажка будет непосредственно вызываться функция `saveModel`.

В функции `initialize` объекту `@model` сообщается, что нас интересуют два его события. Первое событие – `change`. В случае изменения экземпляра модели должна быть вызвана функция `render`, чтобы обеспечить отображение на экране самой последней версии задачи. Эта особенность пригодится, когда пользователь щелкнет на флажке, чтобы изменить состояние задачи и приложению потребуется добавить/удалить соответствующие CSS-классы.

Другое интересующее нас событие – `error`. Это событие может возникать во время попытки сохранить задачу на сервере. В случае появления ошибки должна вызываться функция `modelSaveFailed`, отображающая сообщения об ошибках на экране.

Наконец, нам необходима функция `saveModel`, которая, как уже сообщено фреймворку Backbone, должна вызываться при попытке изменить задачу. К настоящему моменту уже не нужно объяснять вам, как действует эта функция. Она просто извлекает значения атрибутов, доступных для изменения, и передает их функции `save`.

> **Совет.** В классе `NewTodoView` функции `saveModel` передавались функции обратного вызова success и error, а в классе TodoListItemView – нет. Причина в том, что фактически эта задача решается посредством обработчиков событий, возникающих в объекте @model. Этот подход не использовался в классе NewTodoView, потому что мы постоянно создавали новые экземпляры Todo, из-за чего нам пришлось бы постоянно привязывать обработчики событий. Добавить функции обратного вызова оказалось намного проще.

Теперь у нас есть возможность изменять заголовок и признак состояния задачи, сохранять их на сервере и соответственно менять стиль отображения задачи на экране.

Проверка

Прежде, чем завершить описание реализации изменения задач, добавим в класс `Todo` несложную проверку на стороне клиента, чтобы исключить необходимость выполнения дополнительных запросов для проверки на стороне сервера. В частности, нам нужно убедиться, что атрибут `title` не содержит пустое значение.

Благодаря тому, что класс `Todo` наследует класс `Backbone.Model`, он имеет доступ к простейшей системе проверок. Действует она следующим образом: когда вызывается функция `save` класса `Backbone.Model`, она проверяет наличие функции с именем `validate`. Если

такая функция существует, ей передается объект, содержащий все изменившиеся атрибуты. Если функция `validate` вернет значение отличное от `null`, выполнение функции `save` будет прекращено, и она вернет значение, полученное от функции `validate`.

Добавим функцию `validate` в модель `Todo`:

Пример: (исходный файл: `app.7/assets/models/todo.coffee`)

```
# Модель Todo для клиента на основе Backbone:
class @Todo extends Backbone.Model
    # пространство имен 'todo' для данных в формате JSON,
    # см. backbone_sync.js
    paramRoot: 'todo'
    # Сконструировать URL, добавив в конец _id, если имеется:
     url: ->
          u = "/api/todos"
          u += "/#{@get("_id")}" unless @isNew()
          return u
    # По умолчанию функция isNew из фреймворка Backbone ищет строку 'id',
    # Mongoose возвращает "_id", поэтому необходимо исправить функцию:
    isNew: ->
       !@get("_id")?
    # Проверка экземпляра модели перед сохранением:
    validate: (attrs) ->
        if !attrs.title? or attrs.title.trim() is ""
            return message: "Title can't be blank"
```

Как видите, функция получилась очень простой. Она проверяет наличие атрибута `title` и убеждается, что его значением не является пустая строка. Если такой атрибут отсутствует или содержит пустую строку, возвращается объект с ключом `message`, имеющим значение `"Title can't be blank"` (заголовок не может быть пустым).

Вот и все! Попробуйте. Если теперь стереть заголовок в существующей задаче или оставить его пустым в новой задаче, на экране должно появиться окно с текстом предупреждения «Title can't be blank». Все, что необходимо для автоматического вызова метода `validate`, уже имеется. От нас не требуется добавлять еще что-либо.

Удаление моделей из представлений

Сейчас осталось лишь привязать кнопку **Delete** (Удалить) и приложение будет закончено. Реализовать операцию удаления задачи чрезвычайно просто. Необходимо лишь добавить в класс

TodoListItemView обработчик события щелчка на кнопке, и вызвать в нем соответствующую функцию удаления задачи и удаления ее из списка на странице.

Пример: (исходный файл: final/assets/views/todo_list_item_view.coffee)

```
# Представление для каждой задачи в списке:
class @TodoListItemView extends Backbone.View
    tagName: 'li'
    events:
        'keypress .todo_title': 'handleKeypress'
        'change .todo_state': 'saveModel'
        'click .danger': 'destroy'
    initialize: ->
        @template = _.template(Templates.list_item_template)
        @model.bind("change", @render)
        @model.bind("error", @modelSaveFailed)
        @render()
    render: =>
        $(@el).html(@template(@model.toJSON()))
        if @model.get('state') is "completed"
            @$('.todo_state').attr('checked', true)
            @$('label.active').removeClass('active')
            @$('.todo_title').addClass('completed').attr('disabled', true)
        return @
    handleKeypress: (e) =>
        if e.keyCode is 13
            @saveModel(e)
    saveModel: (e) =>
        e?.preventDefault()
        attrs = {title: @$('.todo_title').val()}
        if @$('.todo_state').attr('checked')?
            attrs.state = 'completed'
        else
            attrs.state = 'pending'
        @model.save attrs
    modelSaveFailed: (model, error) =>
        if error.responseText?
            error = JSON.parse(error.responseText)
        alert error.message
        @$('.todo_title').val(@model.get('title'))
    destroy: (e) =>
        e?.preventDefault()
        if confirm "Are you sure you want to destroy this todo?"
```

```
        @model.destroy
            success: =>
                $(@el).remove()
```

После добавления еще одного определения в атрибут `events`, нам осталось лишь написать функцию `destroy`, которая будет удалять задачу обращением к серверному API и исключать ее из списка на странице. Именно эти операции выполняет функция `destroy`, представленная здесь. Прежде чем вызвать функцию `destroy` в модели `Todo`, она вызывает еще одну встроенную функцию фреймворка Backbone, спрашивающую у пользователя подтверждение на выполнение операции.

В случае успешного удаления задачи на сервере вызывается функция `remove` из библиотеки jQuery для удаления элемента `@el` с задачей из страницы. На этом разработка приложения завершена!

> **Совет.** В процессе удаления экземпляра модели возникает несколько событий, обработку которых можно было бы предусмотреть. Например, в коллекции можно было бы реализовать обработку события `destroy` и по нему повторно отображать список задач. Я не стал делать этого по тем же причинам, почему отказался повторно отображать список задач после добавления новой. Просто знайте, что в случае необходимости вы можете реализовать обработку таких событий.

В заключение

В этой главе мы избавились от библиотеки jQuery, использовавшейся в главе 11, «Пример: список задач, часть 2 (клиент на основе jQuery)», и заменили ее фреймворком Backbone.js. Сейчас самое время предложить вам загрузить программный код для этой главы и для главы 11 и сравнить их. Все примеры программного кода для этой книги можно найти на сайте Github.com. [9]

В этой главе мы познакомились с моделями и коллекциями из фреймворка Backbone. Затем узнали, как использовать представления и события для управления элементами страницы и как они могут взаимодействовать друг с другом.

Здесь мы лишь слегка коснулись того, что может предложить фреймворк Backbone.js для создания высокодинамичных, хорошо организованных пользовательских интерфейсов веб-приложений.

Я рекомендую вам ознакомиться с учебными руководствами, статьями в блоге и демонстрационными роликами, посвященными фреймворку Backbone, чтобы побольше узнать о нем.

Примечания

1. http://documentcloud.github.com/backbone/[1].
2. http://ru.wikipedia.org/wiki/Model-View-Controller.
3. https://github.com/jashkenas/.
4. Если говорить серьезно, фреймворк Backbone был выбран не только потому, что его написал Джереми, автор языка CoffeeScript. Мне действительно очень нравится этот фреймворк и я использую его в своей повседневной работе.
5. http://documentcloud.github.com/underscore.
6. http://zeptojs.com.
7. https://github.com/codebrew/backbone-rails.
8. http://www.mongodb.org/.
9. https://github.com/markbates/Programming-In-CoffeeScript.

[1] Документацию с описанием Backbone.js на русском языке можно найти по адресу: http://backbonejs.ru. – *Прим. перев.*

Предметный указатель

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

B

C

D

E

F

H

I

J

M

N

O

P

R

S

T

U

V

W

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по поч товому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Марк Бейтс

CoffeeScript. Второе дыхание JavaScript

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Татаринов А. Ю.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 01.08.2012. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 19,375. Тираж 300 экз.

Веб-сайт издательства: www.dmk-press.ru