



Dart for Hipsters

Fast, Flexible, Structured Code for the Modern Web

Dart语言程序设计

【美】Chris Strom 著

韩国恺 译



人民邮电出版社
POSTS & TELECOM PRESS

The
Pragmatic
Programmers



· 新 · 锐 · 编 · 程 · 语 · 言 · 集 · 萃 ·

Dart for Hipsters
Fast, Flexible, Structured Code for the Modern Web

Dart语言程序设计

【美】Chris Strom 著
韩国恺 译



人民邮电出版社

北 京

异步社区Kensuke(18319066421) 专享 请尊重版权

异步社区电子书

感谢您购买异步社区电子书！异步社区已上架电子书 500 余种，社区还会经常发布福利信息，对社区有贡献的读者赠送免费样书券、优惠码、积分等等，希望您在阅读过程中，把您的阅读体验传递给我们，让我们了解读者心声，有问题我们会及时修正。

社区网址：<http://www.epubit.com.cn/>

反馈邮箱：contact@epubit.com.cn

异步社区里有什么？

图书、电子书（[半价电子书](#)）、优秀作译者、访谈、技术会议播报、赠书活动、下载资源。

异步社区特色：

纸书、电子书同步上架、纸电捆绑超值优惠购买。

最新精品技术图书全网首发预售。

晒单有意外惊喜！

异步社区里可以做什么？

博客式写作发表文章，提交勘误赚取积分，积分兑换样书，写书评赢样书券等。

联系我们：

微博：

@ 人邮异步社区

@ 人民邮电出版社 - 信息技术分社

微信公众号：

人邮 IT 书坊

异步社区

QQ 群：368449889

图书在版编目 (C I P) 数据

Dart语言程序设计 / (美) 斯特罗姆 (Strom, C.) 著
; 韩国恺译. — 北京 : 人民邮电出版社, 2013. 1
(新锐编程语言集萃)
ISBN 978-7-115-29694-8

I. ①D… II. ①斯… ②韩… III. ①程序语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字 (2012) 第242619号

版权声明

Copyright © 2012 The Pragmatic Programmers, LLC. Original English language edition, entitled *Dart for Hipsters*.

Simplified Chinese-language edition Copyright © 2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 The Pragmatic Programmers, LLC 授权人民邮电出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

版权所有, 侵权必究。

新锐编程语言集萃

Dart 语言程序设计

-
- ◆ 著 [美] Chris Strom
 - 译 韩国恺
 - 责任编辑 杨海玲
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
 - ◆ 开本: 800×1000 1/16
印张: 8.75
字数: 165 千字 2013 年 1 月第 1 版
印数: 1—3 000 册 2013 年 1 月河北第 1 次印刷

著作权合同登记号 图字: 01-2012-6702 号

ISBN 978-7-115-29694-8

定价: 35.00 元

读者服务热线: (010) 67132692 印装质量热线: (010) 67129223

反盗版热线: (010) 67171154

广告经营许可证: 京崇工商广字第 0021 号

内容提要

本书是第一本关于 Dart 语言的中文书籍，介绍了当前 Dart 语言的最新内容。书中涵盖了 Dart 语言基础、并发编程、Web 编程和 HTML5 应用等方方面面的内容。

因为 Dart 语言让人感觉非常熟悉，与一般编程语言的书通常以“Hello World”开篇不同，本书一开始就带领读者编写一个 Ajax 功能的应用程序，然后详细讨论 Dart 的基本类型，把 Dart 编译为 JavaScript，面向对象的编程方法，并构建一个易于使用和维护的库，最后介绍在 Dart 中如何使用 HTML5 编程。

作者通过真实的项目，引领读者用 Dart 解决实际问题。每一个项目作为进一步深入讨论 Dart 语言特性的基础。为了增强对 Dart 语言的理解，项目会逐渐深入，并且越来越复杂。读完整本书后，读者不仅获得全面的 Dart 语言的知识，而且还从头构建了一个完整的 MVC 库。

本书适合编程语言爱好者和 Web 开发者阅读。

对本书的赞誉

对于任何想要了解什么是 Dart 语言，以及如何将它与当前浏览器结合起来使用的人，这都是一本有趣且容易阅读的书。对 Dart 语言计划发布特性的评论，给了你买这本书足够的理由。

——Matt Margolis

起初我有点儿怀疑 Dart 语言。这本书让我了解到 Dart 语言的前景和当前的状态，它将作为我可以依赖的一本可靠的参考书目。

——Juho Vepsäläinen

这是第一本介绍 Dart 这种令人兴奋且十分有前途的编程语言的书。清晰和平易近人的文字很吸引读者，它肯定会对 Dart 语言的成功有所贡献。我特别喜欢作者对这种语言函数式方面论述以及对 Isolate 的讨论。

——Dr. Ivo Balbaert

译者序

2011 年 10 月，Google 发布了一种新的编程语言——Dart，主要目标是将其作为一种结构化的 Web 开发语言。Dart 语言既让人感觉熟悉，又足够灵活；既适用于快速原型开发，又适用于组织大型的代码库；既可以用于桌面版和移动版的浏览器中，也可以在服务器端使用。

Dart 是一门很新的语言，自发布以来发展速度很快。这要感谢 Google 的大力支持和社区的积极反馈。说它很新，是因为在翻译此书时，Dart 发布还不满一年，甚至还没有发布第一个正式版（但是很快会发布第一个里程碑版 M1）。说它发展很快，是因为这一年来频繁地发布更新，很多方面都在不断地完善和改进，甚至我在翻译此书时都要不断跟进 Dart 语言的最新版本，并对原书的内容做更新和修订。

正是由于这两点，本书与其他编程语言书籍明显不同。我深知技术爱好者对最新内容的渴望，所以翻译此书时我给自己定的一个目标就是要保持书中的内容最新。通过对原书内容的大量修订和补充，以及对示例代码的实际验证和修改，尽可能保证本书在出版时能够与 Dart 语言的最新版本一致。

Dart 是一门很特别的语言。我觉得其最有特色的特性是可选类型，在动态语言的基础上，结合了静态语言的优点。另一个特色是，Dart 中的类和接口是统一的，每个类都有一个隐式接口。除此之外，还有很多非常不错的特性，比如工厂构造函数和命名构造函数、getter/setter 方法、语言级别的级联调用等。作为现代语言的基本功能，Dart 自然也包含了良好的面向对象和并发编程的支持，未来可能还支持 Mixin 和基于镜像的反射。

编程语言并不是孤立存在的，Dart 也是这样，它由语言规范、虚拟机、类库和工具组成。因为 Dart 要运行于其他浏览器中，所以 dart2js 编译工具也是语言核心的一部分。Dart Editor 是 Dart 的集成开发环境，包含了语言分析工具。Dart 的可选类型带来的一个好处就是，它可以像静态语言一样得到 IDE 的良好支持，而这就是由语言之外的独立分析工具提供的。

尽管本书篇幅不大，但内容比较全面。同时以实践性的项目章节贯穿全书。

这是我翻译的第一本书，翻译中出现错误在所难免，请读者见谅并反馈给我。另外，还要感谢出版社和编辑给予的大力支持。

韩国恺

2012 年 10 月 17 日

前言

为什么使用 Dart

当我问这个问题时，我并不想知道 Google 为什么正在致力于 Dart，我也不是想问这门语言的设计者希望实现什么。当然，在这本书中我们将谈及这些问题和更多的内容。

我也常常问自己：“为什么使用 Dart？”到底是什么让我认为这是一门值得学习的好语言，甚至愿意单独为这门语言写一整本书？特别是当前才发布了 0.08 版^①。

这个问题的答案与我的个人及职业发展轨迹有关——理解如何使互联网更快。回到那段日子，我还是一名简简单单的 Perl 程序员。我很喜欢 Perl 这门语言，并用它做些我想做的东西。但是，当接触了 Ruby 和 Ruby on Rails 之后，我震惊了。简单、清晰的代码和强约定的组合赢得了我的青睐，而且持续了相当长的一段时间。

之后，我尝试了小型框架，如 Sinatra^②，它保留了 Ruby 语言的美，但是却带来更小、更快的代码。这两种框架都满足了 Web 开发的连续性，我当时也很满意。

但是还能有更多的选择吗？这最终把我带到了 Node.js 以及在其上构建的各种 JavaScript 框架，而且看起来很难再对服务器端进行改进。

然后，我发现了 SPDY^③协议，它令我着迷，以至于写了 *The SPDY Book*^④这本书。这不仅是改进现有的东西，而且是尝试重新定义游戏规则。

在我接触 SPDY 的时候，我注意到了一件事，无论我利用多少这种协议所提供的优势，Web 应用的最终速度还是受限于处理网页和客户端脚本、CSS 等的速度。

① 翻译此书时为 0.10 版。——译者注

② <http://sinatrarb.com>，一种 Ruby 语言实现的用于快速创建 Web 应用的 DSL。——译者注

③ <http://www.chromium.org/spdy>。

④ <http://spdybook.com/>。

JavaScript 已经有 17 年的历史了。在它首次被引入时，还没有 Web 2.0、Ajax、CSS，而且根本没有多少客户端交互。当 JavaScript 首次出现时，主要的使用场景就是验证表单并用警告框提示！

在接下来的 17 年里，JavaScript 语言已经从网景公司拥有并缓慢开发的一种专有语言，演化为一个定期添加新特性的 Web 标准。但是与委员会添加新特性到标准中相比，Web 演化的速度明显更快。

然后 Dart 语言到来了。Dart 问：考虑到我们现今所知的 Web，我们如何从头开始构建 JavaScript？怎样才能尽可能快地加载和运行？如何编写才能使我们很容易地定义和加载外部库？

我们怎样才能使开发者轻松地写出漂亮的代码？

如果 Dart 语言是这许多问题的答案（并且我将实际尝试它们），那么 Dart 很可能是很长一段时间内最令人兴奋的技术。

这就是对“为什么使用 Dart？”的回答。

谁应该阅读本书（除了技术达人）

这本书主要是为那些想让自己的 JavaScript 技能保持最新的开发者们写的。提高 JavaScript 技能的最好途径就是亲身实践和阅读别人的代码。但是，有时看看竞争者正在忙什么也会有根本上的帮助。既然这样，随着我们探索 Dart 语言带来了什么，我们能够更好地理解这种优秀语言的与众不同之处。

我也希望这本书会证明这种新的转变是有用的。对于当今的浏览器，Dart 语言已经是一个可用于构建快速 Web 应用的有价值的平台。我希望你在读完这本书后，能够很好地武装起来去开发下一代 Web 应用。

这本书的目标读者应该是喜欢学习各种编程语言的开发者。我关注了 Dart 语言的很多方面，特别是那些令我兴奋和高兴的地方。

当然，技术达人也应该读这本书。Dart 语言相当与众不同，并且足够强大，这使它正吸引着经典的语言技术达人，值得那些想改变世界的人们花时间阅读。

本书的组织

我想做出一些不同的尝试。我没有每章介绍语言的一部分。每个部分都是从一个实际的 Dart 项目开始的，包括一些特意选择的说明。在这些部分中，我的目标是通过给出这种语言的真实感觉，展现出 Dart 语言所声称的重要改进。因为这些都是真实的项目，所以很适合指出 Dart 语言的强项，当然也包括一些弱项。

每一个这样的项目章的后面都会跟着几个更深入语言某个方面的特定主题的章。我使用这些章来涵盖项目章需要更详细解释的地方，以及不能在当前 Dart 语言参考资料中找到的东西。

所以，如果你需要快速的语言介绍，你完全可以单独地从项目章节开始。如果你想要更传统方式的书籍，那么跳过项目章而只阅读主题章，或者全部阅读——我保证它值得你花时间！

第一个项目在第 1 章。对于这个项目的补充在第 2 章、第 3 章、第 4 章和第 5 章。

下一个项目在第 9 章，这个项目源于第 1 章的这个简单 Ajax 应用程序，并把它提炼为一个羽翼丰满的 MVC 库。如果你想要领略一门语言，编写一个程序库，特别是 MVC 库，是最好的方式。与 MVC 库相关的还有第 7 章、第 10 章和第 8 章。

紧接着，我们将在第 11 章看一下 Dart 语言中的依赖注入。与 JavaScript 不同，Dart 语言并不是主要作为一种动态语言，尽管如在第 11 章所看到的那样，它仍然可能实现一些传统动态语言的技巧。在这个项目之后是关于 Dart 语言测试的介绍，这是一个重要的主题，尽管不是完全源自 Dart。

最后一个项目在第 13 章，在这里我们探索作为传统传递回调函数的高层次的替代——Dart “Future”。这引起了第 14 章对代码隔离和消息传递的讨论。

最后，我们以各种 HTML5 技术的简短探索来结束本书。

本书不包含的内容

本书不涵盖 Dart Editor 的内容。在某种程度上看，算是一点儿缺失。Dart 这样的强类型语言具有代码自动完成的能力，Dart Editor 便可提供这种功能。不过，本书的

重点是语言本身，而不是其相关的工具。此外，一些人（包括我自己）会坚持用他们自己选择的代码编辑器。

这本书的目的不是作为语言参考手册。为这种发展中的语言做参考手册还太早了。不过，希望本书能成为（对开发者不够友好的）语言规格说明书^①和（有些地方尚不完整的）API 文档^②的一个有力补充。

关于未来

因为 Dart 语言还会继续发展，根据 Dart 语言的变化速度，本书的内容可能一年有一次或两次的评审，然后做相应的修订、删除或补充。如果您发现任何错误或需要改进的地方，请提交到公共问题跟踪上：<https://github.com/dart4hipsters/dart4hipsters.github.com/issues>。关于新主题方面的建议也非常欢迎。

本书的约定

类名按骆驼式命名（camel-cased），例如 `HipsterModel`。类对应的文件名和类名一样，例如 `HipsterModel.dart`。变量名按蛇式命名（snake-cased^③），例如 `background_color`，而函数和方法名都是小写字母开头的骆驼式命名，例如 `routeToRegExp()`。

让我们开始吧

正如前文所说的，让我们开始编写没有历史包袱的 Web 代码吧。让我们编写 Dart 代码！

① Dart 语言规范在 <http://www.dartlang.org/docs/spec/>。它主要是给语言实现者看的，但必要时也对应用开发者有用。

② <http://api.dartlang.org>。

③ 一种以下划线连结单词的命名方式。——译者注

目 录

第一部分 入 门

第 1 章 项目：第一个 Dart 应用程序	2	4.1 dart:html	25
1.1 后端部分	2	4.2 查找元素	25
1.2 Dart 的 HTML 部分	3	4.3 添加元素	27
1.3 Dart 的 Ajax 部分	4	4.4 删除元素	28
1.4 这个应用程序还无法运行	9	4.5 更新元素	29
1.5 下一步做什么	9	4.6 DOM 就绪	30
第 2 章 基本类型	10	4.7 下一步做什么	30
2.1 数字类型	10	第 5 章 编译为 JavaScript	31
2.2 字符串类型	10	5.1 用 dart2js 编译为	
2.3 布尔类型	12	JavaScript	32
2.4 HashMap（也称为 Hash 或		5.2 维护 Dart 与 JavaScript 并存	34
关联数组）	12	5.3 下一步做什么	36
2.5 列表（也称为数组）	14	第二部分 有效的编程技术	
2.6 日期类型	16	第 6 章 项目：Dart 中的 MVC	38
2.7 类型	17	6.1 Dart 中的 MVC	38
2.8 下一步做什么	18	6.2 实现集合	40
第 3 章 Dart 中的函数式编程	19	6.3 实现模型	44
3.1 匿名函数	20	6.4 实现视图	47
3.2 一阶函数	23	6.5 实现删除	50
3.3 可选参数	23	6.6 下一步做什么	52
3.4 下一步做什么	24	第 7 章 类和对象	53
第 4 章 操作 DOM	25	7.1 类是顶级概念	53
		7.2 实例变量	54

7.3 方法.....	55	11.1 用 noSuchMethod() 改变 类行为	92
7.4 静态方法和静态变量（也称为 类方法和类变量）	58	11.2 通过依赖注入实现同步	97
7.5 接口.....	60	11.3 下一步做什么	101
7.6 子类.....	61	第 12 章 测试.....	102
7.7 构造函数.....	61	12.1 获得测试框架	102
7.8 下一步做什么	67	12.2 2+2=5 应该出错	103
第 8 章 事件.....	68	12.3 下一步做什么	108
8.1 普通事件.....	68	第五部分 Dart 的高级使用	
8.2 自定义事件系统.....	69	第 13 章 项目：终结回调函数的 地狱	110
8.3 下一步做什么	73	13.1 Future.....	110
第三部分 代码组织		13.2 Future 中的错误处理	113
第 9 章 项目：提炼库.....	76	13.3 下一步做什么	115
9.1 要提炼什么，要保留什么	76	第 14 章 Future 和 Isolate	116
9.2 真正的库	81	14.1 Completer 和 Future.....	116
9.3 下一步做什么	84	14.2 Isolate	118
第 10 章 库	85	14.3 小结	119
10.1 part 语句	85	第 15 章 HTML5 和 Dart.....	121
10.2 import 语句	87	15.1 动画	121
10.3 核心 Dart 库.....	89	15.2 本地存储	122
10.4 下一步做什么	89	15.3 WebSocket.....	123
第四部分 可维护性		15.4 Canvas	125
第 11 章 项目：变化的行为.....	92	15.5 小结	127

第一部分 入门

Dart 有很多特点，但也许最令人印象深刻的是，它让熟悉 JavaScript 的程序员感到非常熟悉。在前面的几章中，我们从头开始，编写一个 Dart 应用程序。

第 1 章

项目：第一个 Dart 应用程序

大部分编程书籍都以一个“Hello World!”示例开始。我要说，换种方式——我们这里都是编程达人。让我们开始写代码吧！

首先因为 Dart 语言写起来感觉比较熟悉，所以我们在深入之前不应该走得太远。让我们直接跳到更有趣的事情上：一个带 Ajax 的网站。任何一个真正的达人都会收集大量的漫画书（我说的对吗？不是就我一个人这样吧，哈），那么让我们来考虑一个简单的 Dart 应用程序，通过 REST 风格的接口来操作一组漫画书。

在某种程度上，这可能有点儿太激进了。别担心，我们会在后续章节中深入细节。

1.1 后端部分

本章的示例代码可以在 <https://github.com/eee-c/dart-comics> 的“your_first_dart_app”分支上找到。作为技术爱好者，我们已经在用 Node.js 了，所以后端需要 Node.js 和 npm。说明包含在项目的 README 中。

作为 REST 风格，这个应用程序应该支持下面这些操作：

- GET /comics（返回漫画书的列表）；
- GET /comics/42（返回一本漫画书）；
- PUT /comics/42（更新一本漫画书）；
- POST /comics（在集合中新建一本漫画书）；
- DELETE /comics/42（删除一本漫画书）。

我们不用太关心除此之外的后端部分的细节。

1.2 Dart 的 HTML 部分

我们的整个应用程序将遵循最近的客户端 MVC 框架的惯例。因此，我们只需一个 Web 页面。

`your_first_dart_app/index.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Dart Comics</title>
  <link rel="stylesheet" href="/stylesheets/style.css">

  <!-- 强制让 Dartium 启动脚本引擎 -->
  <script language="text/javascript">
    navigator.webkitStartDart();
  </script>

  <!-- 主程序脚本 -->
  <script src="/scripts/comics.dart"
    type="application/dart"></script>
</head>

<body>
  <h1>Dart Comics</h1>
  <p>Welcome to Dart Comics</p>

  <ul id="comics-list"></ul>

  <p id="add-comic">
    Add a sweet comic to the collection.
  </p>
</body>
</html>
```

对于大家而言，这个 Web 页面的大部分应该很熟悉。它包含了简单的 HTML、CSS 的链接和脚本。

1. HTML 头部

唯一要注意的有点儿奇怪的地方是第一个 `<script>` 标签。在这个标签中，用 JavaScript 来启动 Dart 脚本引擎。

your_first_dart_app/_index_force_dartium_script_engine.html

```
<!-- 强制让 Dartium 启动脚本引擎 -->
<script language="text/javascript">
  navigator.webkitStartDart();
</script>
```

要点: 在写这本书时, 在 Dartium 上必须用 `navigator.webkitStartDart()` 来启用 Dart VM, Dartium 是包含 Dart 语言支持的 Chrome 版本^①。这个要求可能会在不久的将来去除。

接下来, 我们加载了实际代码的内容。这里唯一的变化是 `<script>` 标签的 `type` 属性, 用来表示这是 Dart 代码。

your_first_dart_app/_index_src_dart.html

```
<!-- 主程序脚本 -->
<script src="/scripts/comics.dart"
type="application/dart"></script>
```

第 10 章还有更多关于 Dart 加载库和包含代码的内容。现在只要知道它会按照我们期望的方式加载 Dart 即可。

2. HTML 主体

作为 HTML 的主体部分, 没什么新东西, 不过我们应该注意一下将附加上行为的两个元素的 ID。

your_first_dart_app/_index_body.html

```
<h1>Dart Comics</h1>
<p>Welcome to Dart Comics</p>
<ul id="comics-list"></ul>
<p id="add-comic">
  Add a sweet comic to the collection.
</p>
```

对 `#comics-list` UL 元素, 我们将附加上后端数据存储中的漫画书列表。`#add-comic` 段落标签上将附加上一个表单处理程序。那么, 我们就开始吧。

1.3 Dart 的 Ajax 部分

在 `scripts/comics.dart` 中, 我们的 Dart 应用程序以两个 Dart 库的加载和一

^① <http://www.dartlang.org/dartium/>。

个 `main()` 函数开始。

`your_first_dart_app/comics_initial_main.dart`

```
import 'dart:html';
import 'dart:json';
main() {
  load_comics();
}
load_comics() {
  // 在这里具体实现
}
```

正如我们将在第 10 章看到的，`import` 语句有很多功能。现在我们只要简单地认为它导入了 Dart 核心行为之外的其他功能即可。

所有的 Dart 应用程序都是以 `main()` 作为执行的入口点。在 JavaScript 中，我们只要直接写代码然后就可以运行，但在 Dart 中这样不行。起初这可能看起来类似 C 系列的语言，但这确实是有意义的，难道要让遍布于各处的 JavaScript 源文件和 HTML 文件中的所有代码都立刻执行？^① 在 Dart 语言中，`main()` 入口点不只是约定，它是由这一语言强制的最佳实践。

至于 `load_comics()` 函数，我们一步步来。首先我们需要标识出列表需要附加的 DOM 元素（`#comics-list`），接着我们需要一个 Ajax 调用来填充这个 DOM 元素。为了实现这两件事，我们最初的 Dart 代码看上去大概像下面这样：

`your_first_dart_app/load_comics.dart`

```
load_comics() {
  var list_el = document.query('#comics-list');
  ajax_populate_list(list_el);
}
```

除了明显地忽略了 `function` 关键字之外，这个例子就像是 JavaScript 代码！在第 3 章，我们将介绍更多差异。Dart 语言仍然用我们熟悉和喜欢的分号和大括号——语言设计者有意让人对这门语言感到很熟悉，至少在表面上看起来很熟悉。

注意：和 JavaScript 不同，在 Dart 中分号不是可选的。

除了熟悉，这段代码一看就很容易阅读和理解。没有怪异的、遗留的 DOM 方法。我们用 `document.query()` 而不是用 `document.getElementById()` 来查找元素，并

① 在一个页面中，出现的所有独立 js 文件或嵌入在 HTML 文件中的 JavaScript 都会立即被解析执行。但我们往往需要一个开始执行的入口点，一般是在文档准备好时执行一个函数，如 jQuery 的 `$(document).ready(handler)`。——译者注

且用`#comics-list` 这种 CSS 选择器的方式，正如我们在 jQuery 中已经习惯的那样。

现在已经找到了需要填充的 UL 元素，让我们来看看如何产生一个 Ajax 请求。就像在 JavaScript 中一样，我们要创建一个新的 XHR 对象^①，并添加了一个请求加载完成的处理程序，然后打开并发送这个请求。

`your_first_dart_app/_ajax_populate_list.dart`

```
ajax_populate_list(container) {  
  var req = new HttpRequest();  
  
  req.on.load.add((event) {  
    var list = JSON.parse(req.responseText);  
    container.innerHTML = graphicNovelsTemplate(list);  
  });  
  
  // 动作, 资源, 异步模式  
  req.open('get', '/comics', true);  
  req.send();  
}
```

对于过去做过 Ajax 编程的人，对大部分代码应该立刻就能看懂。我们打开创建的 XHR 对象，指定要获取的资源，然后实际发送请求。

当添加事件处理程序时，我们看到与 JavaScript 的使用方式有所不同。XHR 对象有一个 `on` 属性，它列出了所有支持的事件处理程序。我们访问其中的一个 `load` 处理程序类型，这样我们就能用 `add()` 方法添加一个处理程序。在这种情况下，我们解析获得的 JSON 数据为一个散列值的列表。它大概就像这样：

`your_first_dart_app/comics.json`

```
[  
  {  
    "title": "Watchmen",  
    "author": "Alan Moore",  
    "id": 1,  
  },  
  {  
    "title": "V for Vendetta",  
    "author": "Alan Moore",  
    "id": 2,  
  },  
  {  
    "title": "Sandman",  
    "author": "Neil Gaiman",  
    "id": 3,  
  },  
]
```

^① 为了减少名称中的冗余信息，在新版的 Dart 中，过去的 `XMLHttpRequest` 已改为 `HttpRequest`，相关的名称也是如此，如 `XMLHttpRequestException` 已改为 `HttpRequestException`。——译者注

然后，我们要实现这个简单的 Dart 应用程序的最后一部分——一个填充漫画书列表的模板。

your_first_dart_app/_graphic_novels_template.dart

```
graphic_novels_template(list) {
  var html = new StringBuffer();
  list.forEach((graphic_novel) {
    html.add(_graphic_novel_template(graphic_novel));
  });
  return html.toString();
}

_graphic_novel_template(graphic_novel) {
  return """
    <li id="${graphic_novel['id']}">
      ${graphic_novel['title']}
      by
      ${graphic_novel['author']}
    </li>
    """;
}
```

第一个函数简单迭代这个漫画书列表（在心里，我把它们当做漫画小说），构建为一个 HTML 字符串^①。

第二个函数展示了两个 Dart 语言特性：多行字符串和字符串变量插值。多行字符串由 3 个引号表示（单引号或双引号）。在字符串中，我们可以用美元符号插入值（甚至一个简单表达式）。对于简单的变量插入，大括号是可以省略的：`$name` 和 `${name}` 是一样的。而对更复杂的插入，如散列查找，大括号就是必需的。

就是这样！我们准备好了一个功能完备的、带 Ajax 的 Web 应用程序。汇总的代码如下：

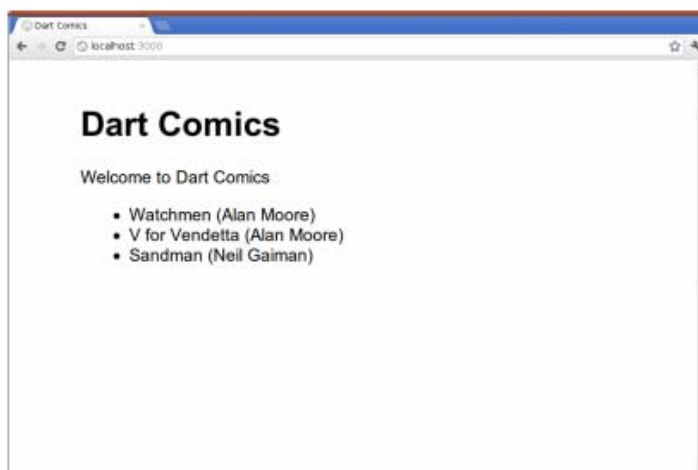
your_first_dart_app/comics.dart

```
import 'dart:html';
import 'dart:json';
main() {
  load_comics();
}
load_comics() {
```

① Dart 语言现在已不支持字符串拼接（+）操作，所以需要改用 `StringBuffer`。简单（非循环）的字符串拼接，如 `str1+str2`，可以用字符串内的变量插值实现，如 `"$a $b"`。或者使用相邻字符串的方式，在 Dart 语言中相邻的字符串会自动拼接，包括多行情况。——译者注

```
var list_el = document.query('#comics-list');
ajax_populate_list(list_el);
}
ajax_populate_list(container) {
var req = new HttpRequest();
req.on.load.add((event) {
var list = JSON.parse(req.responseText);
container.innerHTML = graphicNovelsTemplate(list);
});
// 动作, 资源, 异步模式
req.open('get', '/comics', true);
req.send();
}
graphic_novels_template(list) {
var html = new StringBuffer();
list.forEach((graphic_novel) {
html.add(_graphic_novel_template(graphic_novel));
});
return html.toString();
}
_graphic_novel_template(graphic_novel) {
return ""
<li id="${graphic_novel['id']}">
  ${graphic_novel['title']}
  by
  ${graphic_novel['author']}
</li>
"";
}
```

页面加载后看上去像这样:



这便是我们探索 Dart 语言的良好开端。的确, 我们这里掩饰了许多成就 Dart 语言

的地方。但是这样做，让我们有了一个使用 Ajax 的 Web 应用程序的良好开始。最棒的是，我们写的代码看起来似乎与 JavaScript 没有什么不同。一些语法比我们使用 JavaScript 更整洁一点（没人会抱怨整洁的代码），并且这些字符串特性也很不错。但是总的来说，可以肯定的是，使用相对更短的 Dart 语言是有生产力的。

1.4 这个应用程序还无法运行

在写作本书时，这个应用还不能在（几乎）任何地方实际运行。

任何浏览器（甚至 Chrome）都还不支持 Dart 语言。为了原生地运行这个 Web 应用程序，我们需要安装 Dartium——一个嵌入 Dart VM 的 Chrome 分支版本。Dartium 可以从 Dart 语言网站获得^①。

即使在 Chrome 支持 Dart 语言之后，我们仍然面临着只能被市场上个别浏览器支持的情况。这有点儿糟糕。

好在 Dart 能够编译为 JavaScript，这意味着你可以在利用 Dart 能力的同时对所有平台可用。为了更容易实现这一点，我们添加一个小的 JavaScript 库，用来探测浏览器是否支持 Dart 语言，如果不支持就加载编译成 JavaScript 的等价物。

`your_first_dart_app/_index_src_js_fallback.html`

```
<!-- 启用回退到 Javascript -->
<script src="/scripts/conditional-dart.js"></script>
```

第 5 章中将讨论这个帮助文件的细节。目前，只要知道我们的 Dart 代码不是锁定在一个浏览器厂商那里就足够了。我们肯定不会回到 VBScript 那样。

1.5 下一步做什么

不可否认，这是对 Dart 语言的快速入门。能够如此快速地搭建和运行真是太棒了。仿佛我们此刻就获得了生产力。

尽管如此，我们才刚刚开始学习 Dart 语言，别搞错了，我们的 Dart 代码还可以改进。因此，让我们用下面的几章来熟悉 Dart 语言中的一些重要概念。之后，在第 6 章中，我们将准备好把这个 Dart 应用转变成 MVC 的方式。

^① <http://www.dartlang.org/dartium/>。

第 2 章

基本类型

本书中一个反复出现的主题是，Dart 语言就是要让人觉得很熟悉。如果做到了这一点，那么对这一语言的基本组成部分的论述应该是相对简短的，并且确实是这样。尽管如此，对一些核心类型的介绍还是有幫助的。自然，其中也会有几处“陷阱”。

2.1 数字类型

整数和小数都是数字类型^①，这意味着它们支持很多相同的方法和运算符。Dart 语言中的数字类型与许多其他语言中的数字类型非常像。

```
2 + 2;           // 4
2.2 + 2;         // 4.2
2 + 2.2;         // 4.2
2.2 + 2.2;       // 4.4
```

可以看出，在二者混合运算时 Dart 语言的数字类型做了“正确的事”。

2.2 字符串类型

字符串是不可变的，换种直观的说法就是，字符串的操作会产生新的字符串而不是修改现有的字符串。字符串（像数字一样）是可散列的，这意味着唯一的对象有唯

^① 在 Dart 中，int 和 double 都继承自 num。——译者注

一的一个散列值来区分彼此^①。如果我们将一个字符串变量赋值给另一个变量，它们将有一样的散列值，因为它们是同一个对象。

```
var str1 = "foo",  
    str2 = str1;  
  
str1.hashCode; // 425588957  
str2.hashCode; // 425588957
```

但是，如果我们修改了第一个字符串变量，那么它将获得一个全新的对象，而另一个字符串对象还是指向原来的字符串。

```
str1 = "bar";  
  
str1.hashCode; // 617287945  
str2.hashCode; // 425588957
```

Dart 语言使字符串的处理更容易。例如，可以用三重引号括起来的方式新建多行字符串。

```
"""Line #1  
Line #2  
Line #3""";
```

Dart 语言也支持相邻字符串拼接。

```
'foo' ' ' 'bar'; // 'foo bar'
```

这种相邻字符串的便利用法甚至扩展到多行字符串上。

```
'foo'  
' '  
'bar'; // 'foo bar'
```

最后一个 Dart 字符串的便利用法是字符串内的变量插值。Dart 语言使用\$表示要插值的变量。

```
var name = "Bob";  
  
"Howdy, $name"; // "Howdy, Bob"
```

^① 严格地说，散列值并不要求一定唯一，但应该很好地分布。如果两个对象相等（equals），那么二者的散列值也应该相等，但反之不一定成立。也就是说，不同内容的字符串的散列值也有可能相同，只是概率很小。——译者注

如果在变量表达式结束和字符串开始的地方存在混淆，可以将大括号和\$一起使用。

```
var comic_book = new ComicBook("Sandman");

"The very excellent ${comic_book.title}!";
// "The very excellent Sandman"
```

2.3 布尔类型

在 Dart 语言中，布尔类型（bool）的值只允许是 true 和 false。在 Dart 语言中“真值”和它自身的概念一样简单：如果不是 true，那就是 false。考虑下面的代码：

```
var name, greeting;

greeting = name ? "Howdy $name" : "Howdy";
// "Howdy"

/** Name 仍然不是 true */
name = "Bob";
greeting = name ? "Howdy $name" : "Howdy";
// "Howdy"

greeting = (name != null) ? "Howdy $name" : "Howdy";
// "Howdy Bob"
```

如果你用过许多其他语言，那么不会奇怪在布尔上下文中 null、""和 0 的值为 false。你可能也习惯于"Bob"和 42 的值为 false。

“真值”的语义运行在“类型检查”模式下会稍有不同（见 2.7 节），但是最好不要依靠这种差异。如果我们总是用最后那种写法，那么我们就肯定不会犯错了。

在第 7 章中，我们将探讨操作符定义，这允许我们自定义特定类的 equals/== 方法的含义。这给 Dart 语言关于布尔类型带来了一定的灵活性。

2.4 HashMap（也称为 Hash 或关联数组）

在 Dart 语言中，键值对的数据结构是由 HashMap 对象实现的。下面的代码定义

了一个叫 options 的 HashMap 变量:

```
var options = {  
  'color': 'red',  
  'number': 2  
};
```

如你所料, 用方括号从 HashMap 中获取值:

```
var options = {  
  'color': 'red',  
  'number': 2  
};  
options['number']; // 2
```

HashMap 实现了 Map 接口, 它的大多数 API^①都定义在这里。这包括关于获取键 (keys)、获取值 (values) 以及用 forEach() 迭代整个对象的信息。

```
var options = {  
  'color': 'red',  
  'number': 2  
};  
  
options.forEach((k, v) {  
  print("$k: $v");  
});  
// number: 2  
// color: red
```

注意: 键值对的顺序没有保证。

HashMap 的一个非常有用的特性是 putIfAbsent() 方法。下面两段代码是等价的:

```
// 普通的实现  
if (!options.containsKey('age')) {  
  var dob = new Date.fromString('2000-01-01'),  
      now = new Date.now();  
  options['age'] = now.year - dob.year;  
}  
  
// 更好的实现  
options.putIfAbsent('age', findAge);
```

① http://api.dartlang.org/dart_core/Map.html。

```
findAge() {  
  var dob = new Date.fromString('2000-01-01'),  
      now = new Date.now();  
  return now.year - dob.year;  
}
```

在第一个例子中，条件语句和代码块都与 `options` 对象的实现细节相关。但在第二个例子中，`findAge()` 函数只与计算当前年龄相关，而 `options` 对象则只关心添加值。

要点：应该总是考虑使用 `putIfAbsent()` 方法，这样 Dart 程序会更清晰。

第一个例子也可以改成下面这样：

```
if (!options.containsKey('age')) {  
  options['age'] = findAge();  
}
```

但是，不用 `putIfAbsent()` 方法的话，似乎只能把 `findAge` 的实现放到条件语句中。无论怎样，条件语句的格式永远不会像等价的 `putIfAbsent()` 方法这样清晰。

```
options.putIfAbsent('age', findAge);
```

`putIfAbsent()`——学习它，喜欢它。它会节省你的生命（当然，可能不会，但至少会让生活更轻松）。

2.5 列表（也称为数组）

任何语言都需要表示一个事物的列表。为使开发者容易掌握，Dart 语言的列表基本符合你所期望的那样。

```
var muppets = ['Count', 'Bert', 'Snuffleupagus'];  
var primes = [1, 2, 3, 5, 7, 11];  
// 索引从 0 开始  
muppets[0]; // 'Count'  
primes.length; // 6
```

Dart 语言确实提供了一些不错的并且一致的方法来操作列表。

```
var muppets = ['Count', 'Bert', 'Ernie', 'Snuffleupagus'];

muppets.setRange(1, 2, ['Kermit', 'Oscar']);
// muppets => ['Count', 'Kermit', 'Oscar', 'Snuffleupagus']

muppets.removeRange(1, 2);
// muppets => ['Count', 'Snuffleupagus'];

muppets.addAll(['Elmo', 'Cookie Monster']);
// muppets => ['Count', 'Snuffleupagus', 'Elmo', 'Cookie Monster']
```

同时也有一些内建的迭代方法。

```
var muppets = ['Count', 'Bert', 'Ernie', 'Snuffleupagus'];

muppets.forEach((muppet) {
  print("$muppet is a muppet.");
});
// =>
// Count is a muppet.
// Bert is a muppet.
// Ernie is a muppet.
// Snuffleupagus is a muppet.

muppets.some((muppet) => muppet.startsWith('C'));// true

muppets.every((muppet) => muppet.startsWith('C'));// false

muppets.filter((muppet) => muppet.startsWith('C'));// ['Count']
```

要点：写作此书时，Dart 语言当前缺少 `reduce()` 和 `fold()` 这样的能够执行高阶操作的列表方法。^①

太好了，对于 Dart 语言的列表和数组没有太多需要介绍的了。它是 Dart 语言中众多“刚好够用”的事物之一。

集合（Collection）

上一节中使用的迭代方法其实不是定义在 `List` 类上的。而是来自 `List` 的超类：`Collection`。`Collection` 家族的另两个成员是 `Set` 和 `Queue`。

^① 当前 `List` 已支持 `map` 和 `reduce` 方法。——译者注

Set 是内部元素唯一的集合，并且拥有一些数学集合的操作方法。

```
var sesame = new Set.from(['Kermit', 'Bert', 'Ernie']);
var muppets = new Set.from(['Piggy', 'Kermit']);

// 因为 Ernie 已经在集合中了，所以不会插入
sesame.add('Ernie');           // => ['Kermit', 'Bert', 'Ernie']
sesame.intersection(muppets); // => ['Kermit']
sesame.isSubsetOf(muppets);    // => false
```

Queue 是一种可以在首尾两端操作的集合。

```
var muppets = new Queue.from(['Piggy', 'Rolf']);

muppets.addFirst('Kermit');
// muppets => ['Kermit', 'Piggy', 'Rolf']

muppets.removeFirst();
muppets.removeLast();
// muppets => ['Piggy']
```

根据现在的 Queue 推论，常规的列表不能操作开头的元素。也就是说，List 没有 shift 或 unshift 的方法^①。

2.6 日期类型

在浏览器上 Dart 语言对日期和时间带来了一些非常期盼的改进。关于日期方面有一个令许多 JavaScript 开发者感到痛苦的问题，而在 Dart 语言中第一个月是从 1 开始的^②。让我们开始庆祝吧。

Dart 语言中日期的改进还不止这些。例如，有多种创建日期对象的方法。

```
var mar = new Date.fromString('2012-03-01 14:31:12');③
// 2012-03-01 14:31:12.000

var now = new Date.now();
```

① 除了支持的方法不同外，当前文档中没有清楚地说明 List 和 Queue 的区别。但是，目前可以认为 Queue 默认是由双向链接的列表实现的，所以首尾操作的效率较高；而 List 默认是动态增长的数组，因此二者对于不同类型的操作性能表现是不同的。——译者注

② 在 JavaScript 和 Java 等语言中，第一个月是从 0 开始的，12 个月分别是 0~11。这种设定不直观，也容易犯错。——译者注

③ 目前 Dart API 文档没有准确说明支持哪些字符串格式用于新建 Date。——译者注

```
// 2012-03-10 01:02:24.149

var apr = new Date(2012, 4, 1, 0, 0, 0, 0);
// 2012-04-01 00:00:00.000
```

更好的是，不仅可以操作日期，而且相当好用。

```
var mar = new Date(2012, 3, 1, 0, 0, 0, 0);
var apr = new Date(2012, 4, 1, 0, 0, 0, 0);

var diff = apr.difference(mar);
diff.inDays; // => 31

apr.add(new Duration(days: 15));① // => 2012-04-16
```

`Date` 中的 `difference()` 方法返回一个封装了一段时间的 `Duration` 对象。`Duration` 对象支持从“天”到“毫秒”的各种时间单位的查询。正如在 `add()` 方法示例中所见，`Duration` 对象也很适合从一个特定的日期开始增加或减少时间。

在 Dart 语言中使用日期并不是一件可怕的事。正如我们所见，这很轻松。

2.7 类型

要点：强烈建议在类型检查模式^①下完成 Dart 代码的编写。默认情况下，Dart 运行在“生产”模式下^②，在生产模式下当出现表面上的类型定义冲突时程序不会崩溃。这样就只能依靠开发者自己在开发过程中捕获尽快多的此类问题^④。要启用类型检查模式，在命令行中用 `DART_FLAGS='--enable_type_checks --enable_asserts'/path/to/dartium` 参数启动 Dartium。

在转移到其他主题之前，让我们快速了解一下 Dart 语言中的变量声明。到目前为止，我们都是按照 JavaScript 中的惯例用 `var` 关键字来声明变量的。在 Dart 语言中，`var` 表示可变类型。换句话说，我们不仅没有指定类型，而且还告诉解释器这个类型可能会改变。

① `Date` 的 `add()` 方法返回一个新的 `Date` 对象，当前对象并不改变。——译者注

② Dart 语言有两种运行模式，即检查模式（checked mode）和生产模式（production mode），默认运行在生产模式下。本书原文采用的是“type-checked mode”表示检查模式，这是 Dart 语言早期的说法，现在一般采用“checked mode”的说法，这里还按照原文的“类型检查模式”翻译。——译者注

③ Dart VM 和 Dartium 本身默认都是在生产模式下运行的，而在 Dart Editor 里，所有（命令行和 Web）启动类型默认都是在检查模式下运行的，可以在启动选项中设置。——译者注

④ 官方建议在开发环境中使用检查模式运行，在生产环境下使用生产模式运行。检查模式可以帮你尽早发现问题，但会影响程序性能，所以正式的环境应该用生产模式运行。——译者注

```
var muppet = 'Piggy';

// Dart 类似 JavaScript，允许这样做，但是更进一步！
muppet = 1;
```

Dart 语言一定程度上是强类型语言^①，这意味着它更喜欢我们用实际的类型声明而不是 `var`。

```
String muppet = 'Piggy';

// 在类型检查模式下会失败
// 其他地方会抱怨（在 Dart Editor 中会给出警告）
muppet = 1;
```

Dart 语言也允许我们做一些傻事，就像前面那段代码中那样，但是它会给出警告。在命令行中可以启用类型检查模式，在类型检查模式下，前面那段代码会抛出异常。

尽管 `var` 关键字是可接受的，但是声明类型通常被认为是个好习惯。

```
int i = 0;
bool is_done = false;
String muppet = 'Piggy';
Date now = new Date.now();
```

对于包含其他类型的类型，也可以声明对象要持有的类型。

```
HashMap<String,bool> is_awesome = {
  'Scooter': false,
  'Bert': true,
  'Ernie': false
};

List<int> primes = [1,2,3,5,7,11];
```

声明类型使代码的意图更明确也更容易阅读，这对可维护性是很重要的。在编译代码时，类型也将有助于解释器，允许它运行得更快。

2.8 下一步做什么

本章中包含了很多方面的内容。此刻，Dart 语言中的很多东西应该仍是感到熟悉的，也有一些关键但（我希望是）受欢迎的差异。

^① Dart 是动态类型语言，也是强类型语言，类似于 Ruby。——译者注

第 3 章

Dart 中的函数式编程

JavaScript 的一个特点是它支持函数式编程。因为 Dart 的目标是让人感觉熟悉，现在让我们看看在 Dart 语言中函数式编程是什么样的。

我们先从一个传统的例子开始，计算斐波纳契数列。在 JavaScript 中，大概像下面这样写：

```
function fib(i) {  
  if (i < 2) return i;  
  return fib(i-2) + fib(i-1);  
}
```

探索一个语言的函数式编程特性，斐波纳契数列是个很好的例子。不仅因为它是一个函数，也因为它的递归性质可以展示函数的调用。

我不想纠缠于描述递归或者这个函数的细节^①。相反，让我们关注如何在 JavaScript 中使用这个函数。

```
fib(1) // => 1  
fib(3) // => 2  
fib(10) // => 55
```

看得出 JavaScript 函数足够简单。首先是 `function` 关键字，然后是函数名，跟着是圆括号中的参数列表，最后是描述函数体的代码块。

^① 斐波纳契数列在其他地方有很好的描述，如果你需要重温一下，请参考 http://en.wikipedia.org/wiki/Fibonacci_number。

那么，等价的 Dart 语言版本是什么样呢？

```
// Dart
fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

等一下，这和 JavaScript 的版本有什么不同吗？

```
function fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

细心的读者会注意到，Dart 版本缺少 `function` 关键字。除了这一点，两个函数是完全相同的，调用方式也一样。

```
fib(1); // => 1
fib(5); // => 5
fib(10); // => 55
```

如果没有其他区别，可以看出 Dart 语言的设计者确实让这门语言让人感觉很熟悉。

3.1 匿名函数

有经验的 JavaScript 程序员非常精通于使用匿名函数。因为在 JavaScript 中函数是顶级概念，函数可以在 JavaScript 中任意传递。甚至某些框架成了回调函数的地域，但是撇开审美不说，没有人会否认匿名函数是 JavaScript 中的一个重要部分。那么，在 Dart 中也一定是这样的，对吗？

在 JavaScript 中，匿名函数省略了函数名，仅使用 `function` 关键字。

```
function(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

我们已经看到 JavaScript 和 Dart 的函数仅有的差异是后者没有 `function` 关键字。事实证明，这也是二者在匿名函数上仅有的差异。

```
(i) {  
  if (i < 2) return i;  
  return fib(i-2) + fib(i-1);  
}
```

乍一看，这看起来很奇怪，感觉光秃秃的。但是，这仅仅是从 JavaScript 的角度来看。Ruby 中有比较类似的 lambda 和 Proc。

```
{ |i| $stderr.puts i }
```

认真地考虑一下，在 JavaScript 中 function 关键字真正起了什么作用？下意识的反应是，它有助于标识出匿名函数，但在实践中，这仅仅是一个干扰。

考虑下面这个显示斐波纳契数值的代码：

```
var list = [1, 5, 8, 10];  
list.forEach(function(i) {fib_printer(i)});  
  
function fib_printer(i) {  
  console.log("Fib(" + i + "): " + fib(i));  
}  
  
function fib(i) {  
  if (i < 2) return i;  
  return fib(i-2) + fib(i-1);  
}
```

function 关键字对代码的可读性有帮助还是有阻碍？显然，这使情况变得更糟，尤其是在 foreach() 调用的内部。

让我们考虑以下等价的 Dart 代码。

```
var list = [1, 5, 8, 10];  
list.forEach((i) {fib_printer(i)});  
  
fib_printer(i) {  
  print("Fib($i): ${fib(i)}");  
}  
  
fib(i) {  
  if (i < 2) return i;  
  return fib(i-2) + fib(i-1);  
}
```

注意：我们使用了最早在第 1 章中提到的字符串变量插值的技巧：`"Fib($i): ${fib(i)}"`。

我们所做的只是删除了 `function` 关键字，但是代码的意图更清晰了。如果这种效果贯穿于整个项目，那么将显著提升代码库的长期健康。

说到清晰，如果你嫌大括号麻烦，对于简单的函数还有一种更酷的语法。这个迭代语句中的匿名函数 `(i) { fib_printer(i); }` 可以写成 `(i) => fib_printer(i)`。这样，我们的代码就变成了下面这样：

```
var list = [1, 5, 8, 10];
list.forEach((i) => fib_printer(i));

fib_printer(i) {
  print("Fib($i): ${fib(i)}");
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

参数 `(i)` 在匿名函数的定义和 `fib_printer(i)` 调用中重复出现了。在 JavaScript 中，没有更清晰的做法了^①。然而在 Dart 中，函数 `(i) => fib_printer(i)` 可以进一步被简化为简单的 `fib_printer`^②。

```
var list = [1, 5, 8, 10];
list.forEach(fib_printer);

fib_printer(i) {
  print("Fib($i): ${fib(i)}");
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

在这段 Dart 代码中，这么用确实很简短。

① 在 JavaScript 中也是可以的：`list.forEach(fib_printer);`。——译者注

② Dart 中 `forEach()` 方法要求的参数是一个 `void f(E element)` 的函数，所以可以直接传递函数名。——译者注

3.2 一阶函数

像前面的 `forEach()` 方法那样，这种把匿名函数传递到迭代方法中的行为，已经展示出了函数作为头等公民的良好支持，也就是说，可以把函数像变量一样进行赋值和传递。

在写作本书时，Dart 语言还缺少一些功能（例如反射）来支持复杂的函数式概念，如 `curry` 化或组合（combinator）。不过，在 Dart 中已经可以执行偏函数应用（partial function application）了。^①

偏函数应用的典型示例是把一个对 3 个数字求和的函数转化为固定了其中的一个数字的函数。

```
add(x, y, z) {  
  return x + y + z;  
}  
  
makeAdder2(fn, arg1) {  
  return (y, z) {  
    return fn(arg1, y, z);  
  };  
}  
  
var add10 = makeAdder2(add, 10);
```

偏函数应用这个名字来自于返回一个已经应用了一个参数的函数。在这里，`makeAdder2` 这个函数返回另外一个接收两个参数的函数。调用这个新函数的结果与调用第一个参数固定为 `arg1` 的原函数的结果一样。

在这里，`add10()` 函数接收两个数字参数，对它们求和，再加上 10。

```
add10(1,1); // => 12
```

3.3 可选参数

在 JavaScript 应用中更繁琐的事情之一是提取可选参数。在 Dart 语言中使用内建的语法来封装这个概念，解决了这一问题。

^① Dart 语言已经明确表示要支持基于镜像的反射（Mirror-based Reflection），估计很快将会实现。另外，这里说的偏函数和组合应该是已经支持了，本节中的示例代码已经可以正常运行，但在作者写作时尚不支持。——译者注

像下面这样，把可选参数放在方括号中：

```
f(a, {b1: 'who', b2, b3, b4, b5, b6, b7}) {  
  // ...  
}
```

可以完全不用任何可选参数来调用这个函数：`f('foo')`。在这种情况下，函数体中的参数 `a` 将被赋值为 `'foo'`。

要指定可选参数，需要在函数调用中给它们加上参数名作为前缀。

```
f('foo', b6: 'bar', b3: 'baz');
```

调用前面这个函数的结果是，在 `f()` 方法中，变量 `a` 赋值为 `'foo'`，`b1` 是 `'who'`，`b3` 是 `'baz'`，`b6` 是 `'bar'`。所有其他可选参数 `b2`、`b4`、`b5` 和 `b7` 都是 `null`。

这里要特别注意的是，我们可以在函数参数列表中指定可选参数的默认值。在这个例子中，变量 `b1` 的默认值是字符串 `'who'`^①。

通过对象字面量确实有了改进。可选参数在类和实例方法中甚至更强大，像我们将在第 7 章中看到的那样。

关于 `this` 的简要说明：Dart 与 JavaScript 的不同之一是 `this` 关键字的使用方式。Dart 对此的观点是，`this` 与当前函数毫不相干。相反，`this` 是保留给对象使用的并始终指向当前对象。在 Dart 语言中，没有对 `this` 的绑定（binding）、应用（applying）或调用（calling）。换句话说，`this` 与函数无关。我们将在第 7 章中再讨论这一点，不过会比较简短，因为它在 Dart 中非常简单。

3.4 下一步做什么

本章所述内容的许多方面都还在快速演进中。Dart 语言缺少一些当前 JavaScript 程序员可用的功能：没有反射，而且函数中也没有访问参数的 `arguments` 属性。

即使没有这些，我们也已经看到 Dart 语言在我们能做的事情上是非常强大的。我们将在第 11 章再次回到这个主题上。

① 在 M1 版本中，可选参数分为按名使用和按位置使用的可选参数。按位置的可选参数使用方括号的形式，用等号定义参数的默认值，在使用时不能指定参数名，仅根据参数位置对应。而命名的可选参数使用花括号的形式，用冒号定义参数的默认值，在使用时必须指定参数名，所以参数名也是 API 的一部分。一个函数和方法只能使用其中的一种形式。详见 <http://www.dartlang.org/articles/m1-language-changes/> 中相关的部分。——译者注

第 4 章

操作 DOM

如果不能访问和操作 DOM^①，我们就不能编写 Web 应用程序。遗憾的是，Dart 语言不能完全废除这些已经建立并且经常令人抓狂的 DOM API。幸运的是，在操作网页方面，Dart 语言提供了一套更友好且兼容的库。

4.1 dart:html

负责与网页中的对象进行交互和对对象进行操作的库是 `dart:html`。我们将在第 10 章介绍更多关于库方面的内容。目前，只要认为它们和其他语言（当然除了 JavaScript）中的库是类似的即可——一种封装了逻辑和物理实现相分离的机制。

`dart:html` 不是原始的 DOM API。`dart:html` 核心库是 Dart 按照如果从头开始的话 DOM 编程应该是什么样而设计的。

4.2 查找元素

获取 DOM 的主要入口点是 `document.query()` 和 `document.queryAll()`^②。二者都使用 CSS 选择器作为参数。前者返回一个匹配的元素，后者返回所有匹配的元素列表。下面是一些简单的示例：

① Document Object Model: http://en.wikipedia.org/wiki/Document_Object_Model。

② 导入 `dart:html` 后，也可以去掉 `document` 前缀，直接使用 `query` 和 `queryAll` 方法。——译者注

```
document.query('h1');  
// => 文档中的第一个<h1>  
  
document.query('#people-list');  
// => id是'people-list'的元素  
  
document.query('.active');  
// => 'active'类的第一个元素  
  
document.queryAll('h2');  
// => 所有的<h2>元素
```

`query()`和`queryAll()`实际上是 `Element` 类的方法^①。`Document` 像其他表示 DOM 元素的类一样，是 `Element` 的子类。在实践中，这意味着，如果我们要限制查找某个特定的元素，可以先找到一个元素，然后再从它继续查。

```
var list = document.query('ul#people-list');  
  
var last_person = list.query(':last-child');  
  
last_person.innerHTML  
// => 'Bob'
```

或者我们可以用链式调用：

```
document.  
  query('ul#people-list').  
  query(':last-child').  
  innerHTML;  
// => 'Bob'
```

要点：尽管链式的 `query()` 方法调用可能暗示了 jQuery 风格的组合特性，但是 Dart 语言决定不用集合（set）封装式的操作^②。

尝试实现一下在一个无序列表中高亮显示人名的功能。在 jQuery 中，大概会像下面这样写：

```
$('li', 'ul#people-list').  
  attr('class', 'highlight');
```

在 Dart 语言中，我们只能手工迭代每个元素：

① `Document` 继承了 `Element`，而 `Element` 又继承了 `NodeSelector`，实际是定义在 `NodeSelector` 中的。——译者注

② 这里是指 jQuery 可以对找到的一组元素都执行后续操作的能力，也就是自动完成了集合的迭代功能。而 Dart 不准备这样做，就像下面的示例代码一样，Dart 需要用户自己对 `queryAll` 的结果做迭代操作。——译者注

```
document.
  query('ul#people-list').
  queryAll('li').
  forEach((li) {
    li.classes.add ('highlight');
  });
```

尽管 Dart 处理 DOM 要比纯 JavaScript 容易，但是在某些方面 jQuery 做得更好一点。

在 Dart 中查找页面元素的主要方法是 `query()` 和 `queryAll()`，它们很完美地按你的期望做，使得查询 DOM 很容易。

4.3 添加元素

在 HTML 文档中创建并添加新元素需要组合两个 Dart 概念：Element 和 Node。本质上，节点（node）和元素（element）在 Dart 中表示同样的事物，它们参与典型的 DOM 编程——节点是二者中更为一般的概念。节点可以代表元素、元素的属性，甚至是文本，然而元素只能表示 HTML 的标签对象（tag object）。

在 Dart 中，我们可以使用 Element 的一个叫 `html` 的命名构造函数来方便地创建一个新元素^①。

```
var gallery = new Element.html('<div id="gallery">');
```

`Element.html()` 这个命名构造函数并不限于新建一个单一的元素。我们可以新建要插入页面中的更复杂的 HTML。

```
var gallery = new Element.html("""
<div id="gallery">
  <ul>
    <li>
    <li>
    <li>
    <!-- ... -->
  </ul>
</div>
""");
```

构建较大的 HTML 片段的时候，Dart 中的字符串插值特别适合。结果几乎可以类似于模板：

^① 也可以用 `new Element.tag('div')` 或 `new DivElement()` 新建 div 元素，然后再添加其他属性。——译者注

```
gallery(title, photographer) {
  return new Element.html("""
    <div id="gallery">
      <h2>${title}</h2>
      <ul>
        <li>
        <li>
        <li>
        <!-- ... -->
      </ul>
      <h3 class="footer">
        Photos by: ${photographer}
      </h3>
    </div>
  """);
}
```

要在文档中插入元素，最容易的方法是获得要向其中添加新元素的 `NodeList` 对象^①。

```
var gallery = new Element.html('<div id="gallery">');
document.
  query('#content').
  nodes.
  add(gallery);
```

当需要插入更复杂的元素时，Dart 支持标准的 `insertAdjacentHTML()` 方法^②。这比 jQuery 中 `Element` 的 `prepend()`、`append()`、`before()` 和 `after()` 方法冗长得多。甚至稍微更短的 `insertAdjacentElement` 方法也没法与 jQuery 的等价物相比。

```
var gallery = new Element.html('<div id="gallery">');
document.
  query('#content').
  insertAdjacentElement('afterBegin', gallery);
```

上面这段代码将把 `gallery` 元素插入 `<div>` 内容节点的开始位置（等价于 jQuery 的 `prepend()` 方法）。

在 Dart 中创建元素相当方便。把它们添加到文档或节点的列表中也相对容易。当我们需要执行更加复杂的元素插入操作时，Dart 仍有许多不足之处。理想情况下，这些在不久之后会有所改进。

4.4 删除元素

从文档中删除一个元素非常有 Dart 特色。

① `NodeList` 其实就是一个标准的 `List`，只增加了少量方法。在 `dart:html` 中，元素的 `classes`、`attributes`、`nodes` 其实都是标准的集合对象，因此使用的是标准集合的方法来完成操作。——译者注

② <https://developer.mozilla.org/en/DOM/Element.insertAdjacentHTML>。

```
document.  
  query('#content').  
  query('#gallery').  
  remove();
```

上面的代码会先找到 ID 是"content"的元素，然后再在其中找到"gallery"元素并把它从页面中删除^①。如果对应的页面是类似于下面这样的形式：

```
<body>  
<div id="content">  
  <div id="gallery"/>  
  <p class="instructions">...</p>  
</div>  
</body>
```

那么 remove() 执行之后将是下面这样：

```
<body>  
<div id="content">  
  <p class="instructions">...</p>  
</div>  
</body>
```

与在 JavaScript 中不同，Dart 中要删除元素不需要先找到元素的父元素，这一点做得不错。

4.5 更新元素

我们已经讨论了从父元素中添加元素和删除元素。除了这些动作以外，大部分常见的操作是在元素上添加和删除 CSS 类。下面这段代码将从<blockquote>标签中删除"subdued"类，并添加上"highlighted"类：

```
document.  
  query('blockquote').  
  classes.  
  remove('subdued');  
  
document.  
  query('blockquote').  
  classes.  
  add('highlighted');
```

这里，在对 CSS 类的操作上，我们再次看到了 Dart 中基于集合（set-based）的

^① 这里的第一个查找是多余的，因为 ID 是唯一的，不需要先限定范围。——译者注

方法^①与 jQuery 的链式方法的不同。在 jQuery 中，删除和添加类可以在几个连续调用的单条语句中完成，而在（当前的）Dart 中，我们必须分别使用两条语句完成^②。

除了操作 CSS 类，Element 类还允许修改 innerHTML。

```
document.  
  query('blockquote').  
  innerHTML = 'Four score and <u>seven</u> years ago...';
```

操作 CSS 类和更新 innerHTML 的功能已经涵盖了直接修改元素的大部分常见情况。如果有更多需求，最好先从 Element 类的文档^③开始查看。

注意：dart:html 核心库的实现中，包括了大部分用于内部表示实际 DOM 对象的具体类和接口。库的实现也包含很多链接到浏览器底层的 C 代码^④，用于执行实际的操作。大部分情况下不用关心它们是如何实现的。

4.6 DOM 就绪

在 Dart 中没必要用一个 on-DOM-ready 事件处理程序。Dart 语言做了一个明智的决定，直到 DOM 就绪（ready）时才开始执行处理。^⑤

真是易如反掌！

4.7 下一步做什么

Dart 语言中的 HTML 库展示了一些既熟悉又有所不同的操作网页的方法。它并不是一个像 jQuery 那样的完整的高层次的解决方案。但是，它提供了一个更好的基础，在此之上可以构建出高层次的库。

① 元素上的所有的 CSS 类都在其 classes 属性中，而 classes 其实就是一个标准集合 Set。因为 Set 的 add 和 remove 方法并不返回自身，所以没法链式调用。——译者注

② 当前 Dart 语言已支持更加强化的链式调用语法，使用两个点（..）的方法调用将返回调用对象自己而不是方法的返回结果，这样根本不需要去特意设计 API 也能支持链式调用。这里可以这样写：document.query('blockquote').classes.remove('subdued')..add('highlighted')。——译者注

③ <http://api.dartlang.org/html/Element.html>。

④ Dart 中可以使用 native 关键字做本地扩展，更多信息可以参考这篇文章：<http://www.dartlang.org/articles/native-extensions-for-standalone-dart-vm/>。——译者注

⑤ Web 应用中，不管是本地执行还是编译为 JavaScript 执行，Dart 代码一般都是在 DOMContentLoaded 之后才执行（main 函数是执行的入口点）。——译者注

第 5 章

编译为 JavaScript

在 Dart 语言刚刚发布时，每一个主要浏览器厂商以及 WebKit 项目都宣称他们无意在其浏览器中嵌入 Dart VM。在一个对非标准语言的支持建议中，他们甚至间接地表达了许多对 Dart 的不满。如何能让另一种语言变成标准看起来是个棘手的问题。幸运的是，Google 对此有个计划。

同 CoffeeScript^①的传统做法一样，Dart 项目包含了一个能够把 Dart 代码编译为 JavaScript 代码的编译器。它的目标是，即使 Dart 语言不能一夜之间成为标准，它也能给那些厌烦了诡异的 JavaScript 语言的 Web 开发者另外一种选择。我们现在就能使用这种现代的 Web 编程语言进行编码，但仍然可以支持市场上种类繁多的浏览器。

由 Dart 编译器生成的 JavaScript 代码中包含了各种 Dart 库对应的间接代码，还包括把 Dart DOM 调用翻译成 JavaScript 所生成的 JavaScript 代码，以及支持 Dart Ajax 所生成的 JavaScript 代码。Dart 中差不多所有特性都有对应生成的 JavaScript 表示，这些也被包含在了编译后的输出中^②。

这听上去会很大，是这样。在第一次发布时，编译器生成了数万行的 JavaScript 代码。

当然，编译器正在持续改进中。目前还没有对生成的代码做压缩/优化，但是编译

① <http://coffeescript.org/>。

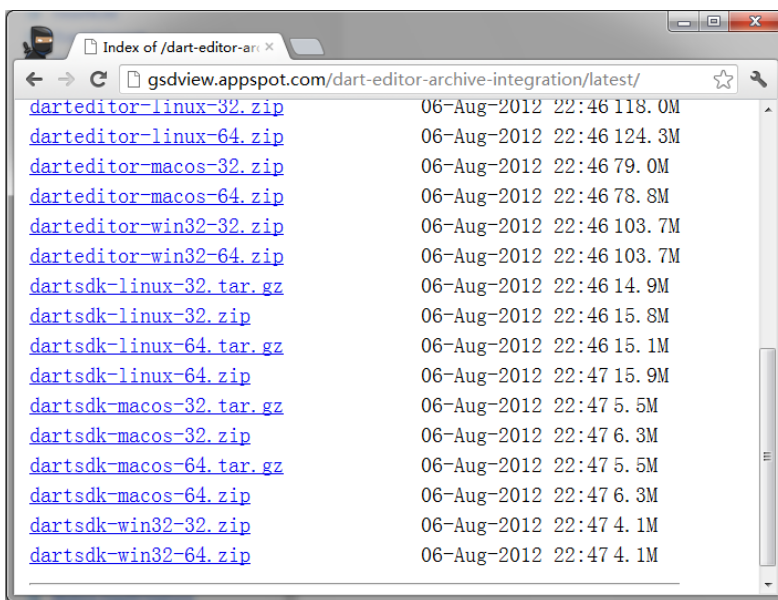
② 目前编译的输出内容只会包含需要用到的部分，而不是语言的所有对应部分。——译者注

器生成的 JavaScript 的代码已经在几千行，而不是几万行了。考虑到 Dart 还做了许多其他 JavaScript 库（如 jQuery）所做的事情，这已经算是不错的开始了。

以后只会更好。

5.1 用 dart2js 编译为 JavaScript

把 Dart 代码编译为 JavaScript 代码的工具是 dart2js。dart2js 编译器可以在最新构建的 Dart 软件开发工具包（SDK）^①中找到。SDK 是下图中文件名以"dartsdk"开头的那些。



这里 Ubuntu 用户应该使用 dartsdk-linux-32.zip 或 dartsdk-linux-64.zip。

解压之后，我们会看到 SDK 中包含了整个 Dart 库（core、html、io、json）。

```
.
+-- bin
+-- lib
```

① <http://gsdview.appspot.com/dart-editor-archive-continuous/latest/>。

③ SDK 中包含了 dart vm、dart2js、pub、dartdoc 等。——译者注

```
|   +-- core
|   +-- html
|   +-- io
|   +-- isolate
|   +-- json
|   +-- uri
|   +-- utf
+-- pkg
|   +-- compiler
|   +-- dartdoc
|   +-- unittest
+-- util
```

除了 Dart 核心库之外，SDK 也包含了支持文档的 `dartdoc` 和编译为 JavaScript 的 `dart2js` 的库的代码。

实际上，`dart2js` 的使用方法再基本不过了。当前，没有改变行为的命令行选项^①。只需要一个要编译的 Dart 代码（的文件名作为参数）就可以简单地运行 `bin/dart2js` 脚本。

```
$ dart2js main.dart
```

编译器会给出编译完成的提示，我们现在有了 Dart 代码对应的 JavaScript 版本了。

```
$ ls -lh
-rw-rw-r-- 1 cstrom cstrom 462 2012-02-03 12:19 main.dart
-rw-rw-r-- 1 cstrom cstrom 7.0K 2012-02-03 12:19 main.dart.js
```

嗯，可能不是太小。

如果在编译 Dart 代码时遇到错误，`dart2js` 会让我们知道哪儿出现了错误。

```
$ dart2js main.dart
main.dart:5:3: error: cannot resolve document
  document.query('#foo');
  ^^^^^^^
Error: Compilation failed.
```

有件事要记住，在编译为 JavaScript 时，`dart2js` 仅在应用程序的级别工作，而不是在类的级别工作。考虑一下这种情况：把我们的漫画书应用程序改写成某种流行的 MVC 模式。

^① 目前 `dart2js` 已支持 `-o` 和 `-c` 两个选项参数，具体请参考 <http://www.dartlang.org/docs/dart2js/>。——译者注

```
comics.dart
Collection.Comics.dart
HipsterModel.dart
Models.ComicBook.dart
Views.AddComic.dart
Views.AddComicForm.dart
Views.ComicsCollection.dart
```

没办法把单个的类编译为可用的 JavaScript。

```
$ dart2js Models.ComicBook.dart
Models.ComicBook.dart:1:1: Could not find main
library comicbook;

Error: Compilation failed.
```

如果包含了 main 入口点的脚本引用了其他库，或者如果这些库又引用了别的库，那么所有这些都会被拉入生成的 JavaScript 中。下面在 import 语句中的这 3 个库将会被拉入编译后的 JavaScript 中：

```
import 'ComicsCollection.dart';
import 'ComicsCollectionView.dart';
import 'AddComicView.dart';

main() { /* ... */ }
```

类似的，下面的 ComicBook 模型也会被包含到 dart2js 生成的 JavaScript 中，因为它在 ComicsCollection 类中被引用到。

```
library comics

import 'Models.ComicBook.dart';

class ComicsCollection { /* ... */ }
```

在某些情况下，用 Dart 语言编写类，再把它们编译为可用的 JavaScript 的这种做法还不错。然而，目前我们只能编译整个应用，而不能编译部分。

5.2 维护 Dart 与 JavaScript 并存

随着 Dart 和 dart2js 的演进，生成的 JavaScript 的性能会得到改进。在某些情

况下，编译后的 Dart 代码可能会与典型的 JavaScript 程序员写出的代码相媲美，甚至可能更好。但是，就算和编译后的 JavaScript 一样快，它也绝不可能和原生运行的 Dart 一样快。

于是问题就变成了如何发送 Dart 代码给支持 Dart 的浏览器，而发送编译后的 JavaScript 给其他浏览器？

答案很简单：包含一小段 JavaScript 代码用于检测浏览器是否支持 Dart，如果不支持就加载对应的 JavaScript。如上一节所见，如果我们编译了 main.dart 脚本，那么 dart2js 就会生成一个对应的 JavaScript 版本 main.dart.js 文件。

下面这段 JavaScript 代码就会做这件事：

```
if (!navigator.webkitStartDart) loadJsEquivalentScripts();

function loadJsEquivalentScripts() {
  var scripts = document.getElementsByTagName('script');
  for (var i=0; i<scripts.length; i++) {
    loadJsEquivalent(scripts[i]);
  }
}

function loadJsEquivalent(script) {
  if (!script.hasAttribute('src')) return;
  if (!script.hasAttribute('type')) return;
  if (script.getAttribute('type') != 'application/dart') return;

  var js_script = document.createElement('script');
  js_script.setAttribute('src', script.getAttribute('src') + '.js');
  document.body.appendChild(js_script);
}
```

Dart 官方也提供了一个类似的脚本^①。大部分情况下，那个脚本应该更合适，因为它还做了其他的事情（如启动 Dart 引擎）。

检查 Dart 引擎是否可用非常简单——如果浏览器知道如何启动 Dart 引擎，那么它就是支持 Dart 的。

```
if (navigator.webkitStartDart)
```

这在我们探索 Dart 的旅程中迟早会有用。

① http://dart.googlecode.com/svn/branches/bleeding_edge/dart/client/dart.js。

上面的 JavaScript 脚本的剩余部分相当简单。loadJsEquivalentScripts() 函数对 DOM 中每个<script>标签调用 loadJsEquivalent()。这个方法中有几个卫述句 (guard clause) 来保证 Dart 脚本正常。然后在 DOM 中添加一个新的.js 的<script>标签来加载等价的 JavaScript。

为了使用这个 JavaScript 检测脚本, 我们把它保存到一个 dart.js 文件中, 并添加到包含了 Dart 代码<script>标签的网页中。

```
<script src="/scripts/dart.js"></script>

<script src="/scripts/main.dart"
      type="application/dart"></script>
```

支持 Dart 的浏览器会直接执行 main.dart。其他浏览器会忽略未知的 "application/dart" 类型并执行 dart.js 中的代码, 然后创建一个新的<script>标签, 它的内容是 dart2js 编译后的 main.dart.js 文件。

最后, 对于支持 Dart 的浏览器, 我们有超快的代码。对于支持 Dart 和不支持 Dart 的浏览器, 我们都得到了优雅的、结构化的、现代的代码。即使是在 Dart 演进的早期, 我们也做到了两全其美。

5.3 下一步做什么

把 Dart 代码编译为 JavaScript 代码的能力意味着, 我们不必等到获得浏览器的支持就能享受 Dart 语言的威力。现在, 我们就可以用 Dart 编写 Web 应用, 并期望它们可以被所有人使用。这是件好事, 因为在下一章中, 我们将把这个简单的 Dart 应用程序带到更高层次上, 而且我们不想离主流太远。

第二部分 有效的编程技术

熟悉了 Dart 语言的基础部分后，现在是开始探索 Dart 语言中独特之处的时候了。我们首先从一个具体项目开始，把第 1 章中那个非常简单的应用转换成一个成熟的 MVC 客户端库。幸运的是，在 Dart 中这很容易做到。

以这个 MVC 库为开始，我们将开始讨论 Dart 语言中最令人兴奋的一些特性：卓越的面向对象编程支持和高度可定制的事件系统。

第 6 章

项目：Dart 中的 MVC

在本章中，我们将第一次体验到编写 Dart 代码的真实感觉。到目前为止，我们的讨论还没有超出熟悉的部分——至少没有超出那些与 JavaScript 相似的部分。

我们将利用第 1 章中那个非常简单的漫画书集合的应用，把它转换成一个 MVC 设计模式。因为这是基于客户端的，所以不是模型-视图-控制器（Model-View-Controller），而是模型-集合-视图（Model-Collection-View），再加上一个路由（Router），类似于 Backbone.js。

我们从用 Dart 语言实现对象的集合开始，然后再具体描述这些对象本身。一旦有了适当的基础，我们就进一步了解视图和模板。

本章是另一个“项目”章节，所以我们将忽略某些 Dart 语言的细节而专注于编写代码。

6.1 Dart 中的 MVC

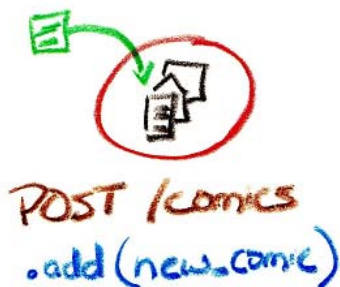
这个 MVC 库的基础是对象的集合，而不是对象自身。这个集合映射到一个 REST 形式的 Web 服务上，它有一个清晰的 API 用于添加、删除和更新记录。

回忆一下第 1 章，我们的漫画书集合可以通过对 `/comics` 上的 HTTP GET 方法来获取。在这个 MVC 中，我们称它为 `fetch()` 方法。

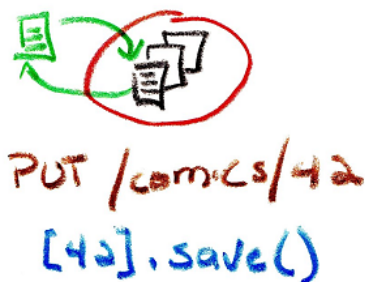


如果以 REST 形式表示资源，我们可以把 `/comics` 视作 URL 的根，因为它是集合上各种操作的 URL 的公共前缀。

例如，添加一本新漫画书到集合中是对 `/comics` 上的一个 HTTP POST 操作。我们称这个操作为 `add()` 方法。



要更新一本漫画书的信息，我们使用 HTTP PUT 方法。为了标识出要更新的具体是哪一本漫画书，我们需要在 URL 的子路径上提供一个 ID：PUT `/comics/42`。从这个 MVC 视角来看，我们先获取记录，然后更新它，最后用 `save()` 方法保存该记录。



最后，要从集合中删除记录，我们使用 `destroy()` 方法。这将导致一个 HTTP DELETE 方法，在集合 URL 上也要包括 ID。



让我们开始编写它的代码吧。

6.2 实现集合

回忆第 1 章中的项目应用，我们的 `main.dart` 看起来是下面这样的：

`your_first_dart_app/comics_initial_main.dart`

```
import 'dart:html';
import 'dart:json';
main() {
  load_comics();
}
load_comics() {
  // 在这里具体实现
}
```

`load_comics()` 方法从 `/comics` 上获取漫画书集合，并显示在网页上。

在这个 MVC 中，集合对象负责获取记录，视图对象负责显示集合的内容。

`mvc/main.dart`

```
import 'dart:html';
import 'dart:json';
main() {
  var my_comics_collection = new Comics()
    , comics_view = new ComicsView(
      el: '#comics-list',
      collection: my_comics_collection
    );
}
```

```
my_comics_collection.fetch();
}
```

我们稍后再看视图。首先，我们按下面的方式定义这个集合类：

mvc/collection_class.dart

```
class ComicsCollection implements Collection {
  // 在这里实现这个集合
}
```

我们把 `ComicsCollection` 类声明为 `Collection` 接口的实现。除了在名字上相似，让 `ComicsCollection` 实现这个接口也是有道理的，因为我们可能想要让漫画书集合的行为类似于一个 `Collection`，可能会访问集合上方法，如 `filter()`、`forEach()`、`length` 等。

注意：在编写脚本或者库的初期实现时，我们可以轻松地丢掉类型信息。但如果我们要编写希望被别人使用的库，类型就是必需的。澄清一下，（在技术上）编写可复用的代码不用类型信息也是可以的，但这不是使用 Dart 语言的好习惯^①。

在正式介绍前，我们先描述一下 `ComicsCollection` 类的构造函数。在 Dart 语言中，构造函数的名字与类名相同。

mvc/collection_constructor.dart

```
class ComicsCollection implements Collection {
  CollectionEvents on;
  List models;
  // 构造函数
  ComicsCollection() {
    on = new CollectionEvents();
    models = [];
  }
}
```

我们首先声明了两个实例变量 `on` 和 `models`。从类型信息上看，变量 `on` 保存集合上的事件列表。我们将在第 8 章中了解更多信息。现在，只要知道这个对象在我们的集合中连结着自定义事件的注册和分发。`models` 属性是一个简单的数组，保存集

^① Dart 语言中的类型是可选的，写不写都可以。但是，类型信息对于表达代码的意图、提高代码可维护性等都是非常有好处的。所以，官方建议，对外使用的接口应该使用类型信息，而在实现内部可以忽略类型信息。——译者注

合中的对象列表。

这个构造函数本身没有参数，不过它定义了这两个实例变量为 `CollectionEvents` 和 `List` 对象的初始状态。

接下来，我们定义一些 `Collection` 接口的方法。

`mvc/collection_collection_methods.dart`

```
class ComicsCollection implements Collection {  
  // ...  
  void forEach(fn) {  
    models.forEach(fn);  
  }  
  
  int get length {  
    return models.length;  
  }  
  
  operator [](id) {  
    var ret;  
    forEach((model) {  
      if (model['id'] == id) ret = model;  
    });  
    return ret;  
  }  
}
```

我们在 `forEach()` 方法的实现中只是简单地委托给了实例变量 `models`。
`length()` 方法也以同样的方式实现，尽管我们把它声明为一个 `getter` 方法。`getter` 方法允许我们以 `object.length` 的形式使用，而不是 `object.length()` 的形式。在代码中减少括号的数量总是好的。

在 `Collection` 相关方法的声明中，最让人感兴趣的是操作符 `[]`^①。`Dart` 语言支持很多操作符。这里这个特殊的操作符提供了一种通过 `ID` 参数查找集合中特定对象的方法。也就是说，如果我们有一个漫画书的集合，那么通过 `comics[42]` 就可以获得 `ID` 是 42 的那本漫画书了。

我们已经使这个漫画书集合的行为类似于一个 `Collection` 接口了，现在需要让它成为一个支持 `Ajax` 行为的对象。为了讨论方便，我们不会完整地实现所有增删改查

① 此处的 `[]` 操作符并不是 `Collection` 接口中定义的方法，而是 `List` 中定义的，但 `forEach` 和 `length` 都是 `Collection` 中的方法。另外，`Map` 接口也使用 `[]` 操作符。——译者注

(CRUD) 操作，而是专注于从后端数据存储中获取对象、在数据存储中新建对象和从数据存储中删除对象。

我们在第 1 章中已经知道在 Dart 中如何通过 Ajax 获取数据。在这里，当数据加载完时，我们将调用一个私有方法^①`_handleOnLoad()`。

mvc/collection_fetch.dart

```
class ComicsCollection implements Collection {
  // ...
  void fetch() {
    var req = new HttpRequest();
    req.on.load.add((event) {
      var request = event.target,
          list = JSON.parse(request.responseText);
      _handleOnLoad(list);
    });
    // 动作，资源，异步模式
    req.open('get', url, true);
    req.send();
  }
}
```

在我们的第一个应用中所做的只是填充一个用户界面元素，然而在这里我们要做得更有框架的风格。这样，我们要建立一个内部集合并触发事件。

mvc/collection_handle_on_load.dart

```
class ComicsCollection implements Collection {
  // ...
  _handleOnLoad(list) {
    list.forEach((attrs) {
      var new_model = new ComicBook(attrs);
      new_model.collection = this;
      models.add(new_model);
    });

    on.load.dispatch(new CollectionEvent('load', this));
  }
}
```

对于收到的每一组模型的属性，我们新建一个模型对象，设置它的 `collection` 属性为当前的集合，然后把它添加到 `models` 集合列表中。一旦完成这些，我们就发

^① 在 Dart 语言中，没有 `private` 关键字，而是以下划线开头的名称来表示私有，并且私有范围是相对库而言的。——译者注

出一个 load 事件给所有可能对此事件监听感兴趣的对象。

模型本身并不要求知道集合（事实上，它不应该直接与集合通信）。我们这里做的赋值是为了使模型能够重用集合上的 URL 来查找、新建和更新后端对象。模型与集合将通过事件广播的方式通信，正如我们这里用 on 属性所做的。

在 CollectionEvents 对象的事件中有一种是 load 事件。这里我们给所有监听集合被加载了的对象发出一个 CollectionEvent 对象。正如我们将要看到的，这是我们触发一个视图对象来显示自己的方式。

当然，我们甚至还没有介绍模型基类，所以我们下一步要实现它。

6.3 实现模型

在集合中有一个 models 属性用于存储所有的模型对象，每个模型有一个 attributes 属性。和集合类似，模型也需要暴露一个 on 属性，作为模型生成的事件的结合点。回忆一下前面的内容，集合给模型传递了一个它自身的引用，这样模型就能快速访问集合的属性了（例如 url）。

这样，我们可以开始定义这个模型类了：

mvc/model_constructor.dart

```
class ComicBook {
  Map attributes;
  ModelEvents on;
  HipsterCollection collection;

  ComicBook(this.attributes) {
    on = new ModelEvents();
  }
}
```

我们在这个类中声明了 attributes、on 和 collection 属性，这些应该都很熟悉了。ComicBook 实例的每个属性会自动产生一对 setter/getter 方法（例如 comic_book.collection = my_comics 和 comic_book.attributes['title']）。我们之前还没见过接收参数的构造函数。它不但接收了一个参数，而且参数是以 this.foo 的形式写的，它和下面这样写的效果是一样的：

```
class Foo
  var bar;

  Foo(bar) {
    this.bar = bar;
  }
}
```

在参数列表中使用 `this` 是 Dart 语言提供的一种方便的快捷方式。

```
class Foo
  var bar;
  Foo(this.bar);
}
```

不管怎样，由于这个模型是类似于 `Map` 的，所以我们将使用操作符 `[]` 来获取属性值。

`mvc/model_operator.dart`

```
class ComicBook {
  // ...
  operator [](attr) => attributes[attr];
}
```

我们将在第 7 章讨论更多关于操作符的内容，不过这个方法的意图十分清楚。当我们在模型上直接查找一个属性时（例如 `comic_book['title']`），它会返回 `HashMap` 类型 `attributes` 中的值。这种定义函数体的快捷形式有时非常方便^①。

模型是否有 ID 属性的问题使得 `ComicBook` 的 URL 有点儿复杂。如果模型有 ID，那么我们认为模型是之前已经在后端保存过的。这种情况下，更新操作是一个 `PUT` 操作，URL 是根资源加上 ID（例如 `/comics/42`）。否则，这是一个新模型，需要 `POST` 到根资源 URL 上（例如 `/comics`）。回想一下，如果模型有漫画书集合，那么 URL 的根可以从该集合上获知。

`mvc/model_url.dart`

```
class ComicBook {
  // ...
  get url => isSaved() ?
    urlRoot : "$urlRoot/${attributes['id']}";
}
```

^① 如果函数体中只是返回一个表达式，就可以用 `=>` 这种写法。——译者注

```
get urlRoot => 'comics';

isSaved() => attributes['id'] == null;
}
```

url 和 urlRoot 的写法只是一种品味的问题。像前面这样写，读起来类似于 Makefile 中的依赖关系。

于是，我们现在可以定义 save() 方法了。要保存一个客户端模型涉及以下事项：

- 一个 HttpRequest 对象，通过它模型数据会被发送到后端的数据存储中；
- JSON 函数，用于在传输之前把数据序列化为 json 字符串以及解析响应的数据；
- XHR 对象的 on.load（也就是在响应成功时）事件上的一个监听器；
- 用服务器返回的数据替换模型上 attributes 的内容；
- 分发“模型已保存”事件，这样漫画书集合和视图能够适当地更新自己；
- 调用一个可选的回调函数，以便 save() 方法的调用对象能够适当地响应。

喔！一个简单的保存功能就有这么多事要做，并且下面的代码也反映了这些。

mvc/model_save.dart

```
import 'dart:json';
class ComicBook {
  // ...
  save([callback]) {
    var req = new HttpRequest()
      , json = JSON.stringify(attributes);

    req.on.load.add((load_event) {
      var request = load_event.target;

      attributes = JSON.parse(request.responseText);

      var model_event = new ModelEvent('save', this);
      on.save.dispatch(model_event);
      if (callback != null) callback(model_event);
    });

    req.open('post', url, true);
    req.setRequestHeader('Content-type', 'application/json');
    req.send(json);
  }
}
```

现在这些看起来应该很熟悉了。我们创建了一个 XHR 对象，用它打开一个到模型 url 上的 POST 请求，设置 HTTP 首部的内容类型为 JSON，然后发送序列化后的模型属性。我们也建立了一个请求监听器，在监听到加载成功的事件后，它将更新模型的属性，并分发自己的模型事件，如果提供了回调函数参数，就会调用它。

最后，我们定义一个看起来非常熟悉的 `delete()` 方法。

mvc/model_delete.dart

```
class ComicBook {
  // ...
  delete([callback]) {
    var req = new HttpRequest();

    req.on.load.add((event) {
      var request = load_event.target;

      var event = new ModelEvent('delete', this);
      on.delete.dispatch(event);
      if (callback != null) callback(event);
    });

    req.open('delete', "${url}", true);
    req.send();
  }
}
```

就像用 Backbone.js 一样，在 MVC 栈中不同部分之间的主要通信手段是事件。幸运的是，Dart 中分发事件是很容易的：`on.delete.dispatch(event)`。监听这些事件也很容易，正如我们接下来在视图部分所看到的那样。

6.4 实现视图

在这个简化的 MVC 库的三个部分中，视图无疑是其中最轻量的。如前所述，由于降水模式（Precipitation Pattern），视图位于 MVC 栈的上层。同样，它允许直接与模型或集合通信（但不赞成这样）。视图可以根据用户输入更新模型，并且视图需要知道模型的属性的值才能显示它们。这样，视图需要暴露出一个 `collection` 属性和一个 `model` 属性。

mvc/view_properties.dart

```
class ComicsView {
```

```
var collection, model;  
Element el;  
}
```

在视图中，这些属性都是可选的——集合视图的子类的职责是了解这是一个集合视图，并因此需要访问 `collection` 属性。**Dart** 在函数和构造函数中有处理可选参数的机制，所以我们在这里使用这种机制声明 `el`、`model` 和 `collection` 为可选的。

mvc/view_constructor.dart

```
class ComicsView {  
  // ...  
  ComicsView({this.el, this.model, this.collection});  
}
```

这一行代码传达了很多信息。首先看一下 `el` 属性。通过命名参数 `el` 可以注入一个现存的 DOM 元素：`new ComicsView(el: '#comics-list')`。在 **Dart** 语言中，在花括号中声明的可选参数是命名参数。使用命名参数的方法是：名字后跟着冒号和想要赋的值。

在构造函数中，可选参数的一个很酷的特性是：如果它属于对应的实例变量，我们就不需要在构造函数中做明确的赋值。为了表示这个可选参数属于对应的同名实例变量，我们要在构造函数的参数列表中以 `this.model` 和 `this.collection` 的方式声明。比如，要注入一个集合到视图对象中，我们可以使用 `collection` 命名参数：`new ComicsView(collection: comics_collection)`。

现在，我们有了一个视图对象，但是它还没有履行它的职责：在用户界面上显示。为了实际显示一个视图，`render()` 方法可以这样定义：把应用了集合的模板结果赋值给视图的 `Element el` 的 `innerHTML`。这里，模板仅仅是迭代整个漫画书集合，为每个模型生成一个漫画书的 HTML。

mvc/view_collection_render.dart

```
class ComicsView {  
  // ...  
  render() {  
    el.innerHTML = template(collection);  
  }  
  
  template(list) {  
    if (list.length == 0) return '';  
  }  
}
```

```

    var html = new StringBuffer();
    list.forEach((comic) {
      html.add(_singleComicBookTemplate(comic));
    });
    return html.toString();
  }

  _singleComicBookTemplate(comic) {
    return ""
      <li id="{comic['id']}">
        {comic['title']}
        ({comic['author']})
        <a href="#" class="delete">[delete]</a>
      </li>"";
  }
}

```

私有方法 `_singleComicBookTemplate()` 很有趣。它使用了 Dart 中的多行字符串和字符串插值，以产生近似于典型模板的效果。

我们不用显式地调用视图的 `render()` 方法。相反，视图订阅了集合的事件。当集合被加载或更新的时候，视图就会被刷新。为了做到这一点，在构造函数中需要订阅事件。

mvc/view_constructor_subscribe.dart

```

class ComicsViews {
  // ...
  ComicsView([this.el, this.model, this.collection]) {
    _subscribeEvents();
  }
}

```

我们目前只关心两个事件：`add` 和 `load`。当集合第一次被加载或者有新记录被添加到集合时，我们声明我们希望视图显示自己。

mvc/view_collection_subscribe_events.dart

```

class ComicsView {
  // ...
  _subscribeEvents() {
    if (collection == null) return;

    collection.on.load.add((event) { render(); });
    collection.on.insert.add((event) { render(); });
  }
}

```

```
}  
}
```

添加事件监听器的 API 简单明了。更重要的是，我们不用担心 `this`。即使在被事件监听器调用的匿名函数中，方法也会继续在它所在的类中被调用。在这种情况下，消除 `apply()`、`call()` 和 `bind()` 的复杂性确实很有好处。

令人惊讶的是，这就是渲染模板需要的全部了。我们在 `main()` 入口点创建了一个漫画书集合对象，并把它传给视图，然后执行 `fetch()` 方法。

mvc/main.dart

```
import 'dart:html';  
import 'dart:json';  
main() {  
  var my_comics_collection = new Comics()  
    , comics_view = new ComicsView(  
      el: '#comics-list',  
      collection: my_comics_collection  
    );  
  my_comics_collection.fetch();  
}
```

当 `fetch()` 方法完成时，它会广播一个 `load` 事件，此时视图将显示自己。

此时，我们已经完全把最初的应用程序改成了一个 MVC 风格的实现。与最初的实现不同，这种方法知道如何从后端添加和删除记录。现在让我们开始添加从集合中删除漫画书的功能吧。

6.5 实现删除

为了允许用户从这个 MVC 应用中删除一条记录，我们需要在用户界面上添加事件处理程序。这可以通过在初始化过程中调用另一个方法来实现——`_attachUiHandlers()`。

mvc/view_collection_attach_ui_events.dart

```
class ComicsViews {  
  // ...  
  ComicsView([this.el, this.model, this.collection]) {  
    _subscribeEvents();  
    _attachUiHandlers();  
  }  
  // ...  
}
```

```

_attachUiHandlers() {
  attach_handler(el, 'click .delete', delete);
}
delete(event) {
  var id = event.target.parent.id;
  collection[id].delete(callback:(_) {
    event.target.parent.remove();
  });
}
}

```

这里，我们把包含了 `class="delete"` 的元素上的点击事件委托给视图的 `delete()` 方法。集合中当前对象的 ID 存储在父元素 (``) 的 ID 属性上（我们也可以使用 HTML5 的数据属性）。通过 ID，我们可以找到集合中的模型，这样就可以调用模型的 `delete()` 方法。我们提供了一个回调函数，当模型成功地从后端数据存储中删除时，我们就可以删除网页上的对应元素。

这在定制代理事件时会很有用。因为当类刚被解释时，集合可能还没初始化，并且新元素可能将被添加到视图中，所以定义一个在当前和页面更新时工作的事件处理程序是绝对必要的。

代理事件还没有在 Dart 中内建，不过我们可以添加它们。

mvc/view_attach_handler.dart

```

class ComicsView {
  // ...

  attachHandler(parent, event_selector, callback) {
    var index = event_selector.indexOf(' ');
    , event_type = event_selector.substring(0,index)
    , selector = event_selector.substring(index+1);

    parent.on[event_type].add((event) {
      var found = false;
      parent.queryAll(selector).forEach((el) {
        if (el == event.target) found = true;
      });
      if (!found) return;

      callback(event);

      event.preventDefault();
    });
  }
}

```

```
    });  
  }  
}
```

`attach_handler(e1, 'click .delete', delete)` 这条语句的意思是，在 DOM 元素 `e1` 上添加一个特定事件的处理程序。这里的 `event_type` 变量是 `click`，`selector` 变量是 CSS 类 `.delete`。我们在父元素 `e1` 上添加了一个单击事件监听器。每次在元素上发生点击事件时，`attachHandler()` 在所有的子元素中查找 `.delete` 样式。如果找到，就调用提供的回调函数。

Dart 还没有提供代理事件，理想情况下，随着语言的演进，这个早期的遗漏会被弥补。与此同时，我们的简单解决方案就够用了。现在我们可以从用户界面上删除漫画书了。

6.6 下一步做什么

在本章中我们真正地领略了 Dart 语言的威力。我们把一个非常初级的 Dart 应用程序转换成了一个非常函数式的 MVC 库。我们也看到一些面向对象和基于事件的编程。但有时忽略了 Dart 语言对这两种编程范式的支持细节，所以在接下来的两章中，我们将探索它们的更多细节。

当我们在第 9 章再次回到这个项目时，我们将看到 Dart 语言最酷的早期特性之一：真正的库。

第 7 章

类和对象

在第 6 章中我们广泛使用了类和对象来构建一个 MVC 库。从中我们学到了两点：第一，如果不用一些面向对象编程的话，很难用 Dart 做重要的工作；第二，用 Dart 完成面向对象编程很容易。在本章，我们将形式化说明 Dart 语言是如何对待它的类和对象的。

7.1 类是顶级概念

Dart 语言采用了典型的面向对象编程的方式，这是一种重要的、受欢迎的方式，与 JavaScript 使用的基于原型（prototype-based）的方式相区别。基于原型的语言确实提供了一些好处，但是不太易用。

正如我们所见，Dart 语言使用 `class` 关键字引入了类。

```
class ComicsCollection {  
    // 在这里描述类的操作方法...  
}
```

在 Dart 语言中，这就是定义类所需的全部——没有神奇的 `function` 构造函数，不需要重量级的库来提供典型的类定义^①，仅仅是 `class` 跟着一个类名。

注意：尽管在 Dart 语言中类是顶级概念，但并不是顶级对象。正如上一章所见，不能像其他语言中那样把类名作为变量传递。

^① 在 JavaScript 中，类是通过 `function` 定义的。要像其他语言那种典型的定义方式，一般是由第三方 JavaScript 库提供的，如 ExtJS、MooTools 等。——译者注

7.2 实例变量

实例变量也就是在类中声明的变量。

```
class ComicsView {  
    ComicsCollection collection;  
    ComicsModel model;  
    Element el;  
    // ...  
}
```

在这个例子中，我们声明了 3 个不同类型的实例变量。我们也可以把这 3 个实例变量声明为可变类型（`var collection, model, el;`），但是作为好的库的维护者，最好能够明确声明它们的类型。

在类中，实例变量只需要用变量名访问——不需要在变量前用 `this.`，除非是为了区分局部变量。

默认情况下，实例变量是公有的（`public`），就是说它们可以被子类访问，也可以在类外部访问。对任何公有的实例变量，**Dart** 语言自动为其创建一个与变量名同名 `getter` 和 `setter` 方法。例如，我们要访问漫画书视图的集合，可以这样做：

```
comics_view.collection;  
// => 一个 ComicsCollection 的实例
```

要让视图切换到一个新集合上，可以这样：

```
comics_view.collection = new_collection;
```

公有的实例变量使用起来很方便，但如果需要做访问控制就要小心使用。

私有实例变量

有些情况下，公有的实例变量让人有所顾虑。如果类库不想在使用环境中直接暴露实例变量，那么它应该声明为私有变量。在 **Dart** 语言中，私有变量就是以下划线开头的变量（例如 `_models`）。例如，如果我们不想集合对象被改变，我们应该把集合声明为私有变量，同时仍然暴露公有的 `getter` 方法。

```
class ComicsView {
  // 因为是下划线开头，所以是私有的
  ComicsCollection _collection;
  ComicsCollection get collection {
    // 这里也许做访问限制...
    return _collection;
  }
}
```

要点：私有变量仅在定义它们的库中可用。如果子类定义在基类之外的库中，那么子类不能访问超类的私有成员。

这说明一个重要的原则：不要在超类和子类间访问彼此的私有成员。虽然它们在同一个库中的时候可以这么做，但是如果它们被重构到不同的库中，就有麻烦了，所以不要这么做。

我们会在第 11 章遇到这个限制。

7.3 方法

方法就是类中的命名函数。在下面这个方法中，通过给实例变量 `el` 的 `innerHTML` 赋值为模板的结果，显示当前视图。

```
class ComicsView {
  // ...
  render() {
    el.innerHTML = template(collection);
  }
}
```

在类中，方法可以被直接调用——并不要求像 JavaScript 中那样在方法前要插入 `this.`（尽管也可以这么做）。在前面的例子中，`template()` 是一个实例方法，它用视图的 `collection` 属性作为参数调用。

在 Dart 语言中，声明方法的返回类型，或者在没有返回值时声明 `void`，被认为是好的实践。

```
class ComicsView {
  // ...
  void render() {
    el.innerHTML = template(collection);
  }
}
```

除了常规的方法，Dart 语言也支持特殊的 `setter` 和 `getter` 方法，以及操作符方法。

1. `getter` 和 `setter`

`getter` 方法是一种没有参数，在类型之后、方法名之前用关键字 `get` 声明的方法^①。

```
class ComicsCollection extends Collection {  
  String get url => '/comics';  
}
```

直接用它们的名字来使用 `getter` 方法，类似于其他语言中获得对象的属性。

```
// 不需要括号!  
comics_collection.url;  
// => '/comics'
```

Dart 语言也支持用于赋值的 `setter` 方法。它是用 `set` 关键字声明，并接收一个（要赋予的新值）参数的方法。

```
class ComicsCollection extends Collection {  
  String _url;  
  void set url(new_url) {  
    _url = new_url;  
  }  
  String get url => _url;  
}
```

`setter` 方法非常有趣，因为它覆盖了赋值操作符。要在 `Comics` 类中设置一个新的 URL，我们不需要把它作为参数传递给 `url()` 方法。而是直接给它赋值：

```
comics_collection.url = shiny_new_url;
```

Dart 语言把赋值识别为一个特殊的 `setter` 操作，并且在类定义中寻找 `set` 关键字来决定如何处理。

`setter` 和 `getter` 方法超越了那些强制我们选择某种弱的约定来表明意图的语言。

2. 操作符

事实上，在 Dart 的类中可以描述许多类似操作符的方法。其他操作符都使用关键字 `operator` 声明。

^① 现在，Dart 语言中定义 `getter` 方法时不需要方法名后的空参数列表，也就是那对圆括号，而以前是必需的。——译者注

我们在 `ComicsModel` 类中看到过一个操作符，它作为一种访问模型属性的方法。

```
class ComicsModel {
  // ...
  operator [](attr) => attributes[attr];
}
```

用这个操作符，我们就能用下面这种方式来查找 `Comic` 对象的 `title` 属性了：

```
comic['title'] // => "V for Vendetta"①
```

方括号是目前为止 `Dart` 语言中最常用的操作符，但是还支持很多其他操作符：`==`、`<<`、`>`、`<=`、`>=`、`-`、`+`、`/`、`~/`、`*`、`%`、`|`、`^`、`&`、`<<>`、`[]=`、`[]`、`~`和`!`。

还有一个 `call` 关键字，它允许我们描述当一个对象被应用（`apply`）时应该做什么^②。例如，如果我们想要调用一个模型对象（如 `comic()`）作为保存方法的别名，我们应该这样写：

```
class ComicsModel {
  operator call() {
    this.save();
  }
}
```

然后，保存方法就可以这样使用了：

```
comic_book();
```

3. 用 `noSuchMethod` 做元编程

在早期，`Dart` 语言提供了有限的元编程功能。在运行时动态改变行为的一种方法是使用特殊的 `noSuchMethod()` 方法。当 `Dart` 尝试定位一个方法调用的时候，它首先在当前类中查找明确的定义。如果没找到，那么会检查超类及所有祖先类。如果还是没有，那么 `Dart` 将在当前类中调用 `noSuchMethod()` 方法——如果它声明了。

在 `noSuchMethod()` 方法被调用时，它将接收要调用的方法名和传递的参数列表。

① 《V 字仇杀队》。本书使用了一些漫画书的示例，后文中也用了一些漫画书的名字，如《超人》（`Superman`）、《睡魔》（`Sandman`）。——译者注

② `call` 操作符目前还没有实现，计划将在 `Milestone1` 版本中实现。详见：<http://www.dartlang.org/articles/emulating-functions/>。——译者注

```
class ComicsModel {  
  // ...  
  noSuchMethod(name, args) {  
    if (name != 'save') {  
      throw new NoSuchMethodException(this, name, args);  
    }  
    // 这里实现保存操作...  
  }  
}
```

我们将在第 11 章了解 `noSuchMethod()` 方法的更多细节。

注意：将来随着 Dart 语言的演进，可能会添加更多的动态机制，但是由于两点原因这不是优先的工作事项：

1. 这不利于代码自动完成 (Code Completion)；
2. 这是编译器不能发现的一种常见 bug 来源。

对于不依赖代码自动完成功能的人而言，第一点不是很有说服力。Ruby 和 JavaScript 程序员可能会反对第二点——动态语言的特性成了一种常见 bug 来源的观点。即便如此，它们确实阻碍了编译器捕获潜在的错误。

不管怎样，Dart 语言并不反对在将来变成更动态的语言，只不过不是当务之急。

7.4 静态方法和静态变量（也称为类方法和类变量）

Dart 语言包括类变量和类方法的概念，尽管它们被认为是有害的。它们被视为必要之恶 (a necessary evil)，并且确实是。它是由 `static` 关键字引入的^①。

```
class Planet {  
  static List rocky_planets = const [②  
    '水星', '金星', '地球', '火星'  
  ];  
  
  static List gas_giants = const [  
    '木星', '土星', '天王星', '海王星'  
  ];  
  
  static List known() {  
    var all = [];
```

① 以前 Dart 中的 `static final` 字段或顶层的 `final` 变量只支持常量表达式，M1 版本已支持任意表达式（并采用惰性初始化）。——译者注

② `const` 是用于修饰常量值的。——译者注

```

    all.addAll(rocky_planets);
    all.addAll(gas_giants);
    return all;
  }
}

```

调用一个静态方法就像调用实例方法一样，但是要以类名调用（类自身是方法的接收者）^①。

```

Planet.known()
// => ['水星', '金星', '地球', '火星',
// '木星', '土星', '天王星', '海王星']

```

有趣的是，在实例方法中可以把静态方法看作其他实例方法^②。

```

class Planet {
  // ...
  static List known() { /* ... */ }

  String name;
  Planet(this.name);

  bool isRealPlanet() {
    return known().some((p) => p == this.name);
  }
}

```

在上面的代码中，实例方法 `isRealPlanet()` 调用了静态方法 `known()`，就像它也是实例方法一样。用这种方法，我们能够发现海王星是行星，而冥王星不是^③。

```

var pluto = new Planet('冥王星');
var neptune = new Planet('海王星');

neptune.isRealPlanet()
// => true

pluto.isRealPlanet();
// => false

```

警告：因为在这种形式下，Dart 语言把静态方法当作实例方法，所以不允许实例方法与类方法同名。

① 在类的外部，只能以类名来直接调用静态方法，不能使用类的实例对象调用静态方法。——译者注

② 在实例方法中，可以直接调用静态方法，或用类名来调用，但不能用 `this` 调用静态方法。——译者注

③ 2006 年冥王星被划为矮行星，所以现在太阳系是八大行星。——译者注

7.5 接口

在 Dart 语言中，接口用于描述在多个实体类中共享的功能。它描述了一组应被类所实现的方法。在函数和方法调用中，它也是一种限制参数类型的手段^①。

我们的 ComicsCollection 类实现了内建的 Collection 接口。

```
class ComicsCollection implements Collection {  
  void forEach(fn) {  
    models.forEach(fn);  
  }  
  
  int get length {  
    return models.length;  
  }  
}
```

这里这个类向其他类表明，它支持 Collection 的所有方法，如 forEach 和 length。

动态语言的信徒喜欢鸭子类型（duck typing），它相当于在问“谁关心类型是什么，只要对象支持 forEach 方法就行”^②。实际上，如果你愿意，Dart 语言可以让你摆脱这类行为。就是说，使用接口来声明为什么要支持这些方法的做法更符合 Dart 语言的习惯。

如果你需要支持多个接口，在类的声明中用逗号简单地分开就可以。

```
class ComicsCollection implements Collection, EventTarget {  
  // 集合的方法  
  void forEach(fn) { /* ... */ }  
  int get length { /* ... */ }  
  
  // EventTarget  
  Events get on => _on;  
}
```

澄清一下，在 Dart 语言中这些（显式地实现接口）都不是必需的，但是明智地使

① 在 M1 版本中，Dart 取消了明确的 interface，仅包含抽象方法的抽象类就相当于接口。类也是接口，每个类都有一个隐式的接口，包含类的所有实例成员。这样实现一个接口就是实现一个类。类和接口是统一的。——译者注

② Dart 语言本质上是动态语言，类型是可选的，所以它是支持“鸭子类型”的。当然要运行在生产模式下，而不是在检查模式下。——译者注

用接口在很大程度上改善了代码的可读性。

7.6 子类

在 Dart 中，我们说子类用新功能来扩展超类。正如在第 9 章中将会看到的，大部分集合的功能被分离到 `HipsterCollection` 超类中。`Comics` 子类只需要少量方法来扩展 `HipsterCollection`。

```
class Comics extends HipsterCollection {  
  get url => '/comics';  
  modelMaker(attrs) => new ComicBook(attrs);  
}
```

`extends` 关键字对于阅读有明显的益处，它提高了代码库整体的可维护性。

注意：当前，Dart 类实现多种行为的唯一方式是实现多个接口。每个类只能有一个超类。`Mixin`（或 `Trait`）的实现仍在进行中。

抽象方法

在前面的示例代码中，`url` 和 `modelMaker` 在基类中都是抽象方法^①。

```
abstract class HipsterCollection {  
  HipsterModel modelMaker(attrs);  
  String get url;  
}
```

这表示 `HipsterCollection` 是一个抽象类（需要子类才能使用），并且理想情况下非抽象类应该会覆盖这些方法。如果子类没有实现这些方法，代码不会有编译时错误。然而，使用未实现的方法肯定会遇到异常。

7.7 构造函数

Dart 语言的构造函数有很多惊奇之处。它结合了两种类型的构造函数：生成式构造函数和工厂构造函数。二者的差异在于如何创建新对象。生成式构造函数帮我们创

^① 在 M1 版本中，抽象方法前不需要 `abstract` 标识符，没有实现的方法就是抽象方法，有实现的方法就是非抽象方法。——译者注

建新对象，而我们只需要负责初始化内部的状态。而在工厂构造函数中，我们需要自己负责创建并返回新对象。正如我们将看到的，生成式构造函数的简单性和工厂构造函数的灵活性都非常强大。

生成式构造函数是二者中更常用的，所以先介绍它。

1. 简单的生成式构造函数

借用第 6 章中的例子，构造函数的最简单的形式看起来像是一个与类同名的方法。

```
class ComicsCollection {
  CollectionEvents on;
  List<ComicsModel> models;

  // 构造函数
  ComicsCollection() {
    on = new CollectionEvents();
    models = [];
  }
}
```

这个类的构造函数没有参数并给两个实例变量赋值为默认值。在生成式构造函数中没有 `return` 语句——我们仅影响对象的内部状态。

2. 命名构造函数

JavaScript 可以用 `arguments` 对象/数组完成很多事情。Dart 语言并不支持这种概念。它使用可选参数（见第 3 章）和命名构造函数弥补这一点。命名构造函数是一种创建特定构造函数的机制。例如，如果我们想要以一个属性列表创建一个 `ComicsCollection`，那么我们可以声明一个 `ComicsCollection.fromCollection` 构造函数。

```
class ComicsCollection {
  ComicsCollection() { /* ... */ }

  ComicsCollection.fromCollection(collection) {
    on = new CollectionEvents();
    models = [];
    collection.forEach((attr) {
      var model = modelMaker(attr);
      models.add(model);
    })
  }
  // ...
}
```

这允许我们像下面这样实例化一个集合对象：

```
var comics_collection = new ComicsCollection.fromCollection([
  {'id': 1, 'title': 'V for Vendetta', /* ... */ },
  {'id': 2, 'title': 'Superman', /* ... */ },
  {'id': 3, 'title': 'Sandman', /* ... */ }
]);
```

就像普通的生成式构造函数一样，命名构造函数也是以类名开始的，然后跟着一个点和名字来表示（如 `fromCollection`）。正如其他生成式构造函数一样，命名构造函数并不返回任何东西，它们仅仅是改变对象的内部状态。

除了一个标准构造函数，类可以有任意数量的命名构造函数^①。这允许我们通过一系列良好命名的且只做一件事的构造函数来实现专门的对象实例化。这有效地消除了 JavaScript 中困扰对象初始化的复杂条件。

命名构造函数对于代码的可读性和可维护性是巨大的胜利。

3. 重定向构造函数

一旦我们开始有效地使用 Dart 中的多个构造函数，我们很快就会陷入重复逻辑的麻烦。例如，模型基类中两个不同的构造函数都需要建立用于监听和广播事件的 on 属性。

```
class ComicsModel {
  Map attributes;
  ModelEvents on;
  ComicsModel(this.attributes, [this.collection]) {
    on = new ModelEvents();
  }
  ComicsModel.fromMap(this.attributes) {
    on = new ModelEvents();
  }
}
```

注意：我们不能在声明时简单地给 on 赋值为 `new ModelEvents()`，因为它不是编译时常量^②。

为了避免重复自身，我们使用重定向构造函数。

① 注意，命名构造函数与类的成员共享相同的命名空间，所以应避免与方法名冲突。——译者注

② 在 M1 版本中，Dart 已经不再限制类的实例字段和静态字段声明时的初始化表达式必须是编译时常量，所以就不存在这个问题了。详见 <http://news.dartlang.org/2012/09/Simplify-your-constructor-and-top.htm/>。——译者注

```
class ComicsModel {
    Map attributes;
    ModelEvents on;

    ComicsModel(this.attributes, [this.collection]) {
        on = new ModelEvents();
    }

    // fromMap 构造函数重定向到通用的构造函数
    ComicsModel.fromMap(attributes): this(attributes);
}
```

现在，on 属性仅声明在一个地方——默认构造函数 `new ComicsModel()`。这对于可维护性是不错的双赢。

重定向使用冒号引入。冒号后跟着的是重定向的目标。在这里我们用 `this()` 指定了默认构造函数。重定向也可以指向其他命名构造函数：`this.withTitle(title)`。它也可以指向超类构造函数或超类的命名构造函数。例如，`ComicBook` 模型可能需要定义一个多产作者的构造函数。

```
class ComicBook extends ComicsModel {
    ComicBook(): super();
    ComicBook.byNeilGaiman(): super({author: 'Neil Gaiman'});
}
```

要点：前面这个例子展示了隐式构造函数。默认情况下，子类构造函数会隐式调用超类的默认（无参数）构造函数（`:super()`）。但是，如果超类中没有默认（无参数）构造函数的定义，那么必须自己显式地调用一个超类的构造函数。

4. 构造函数的参数

我们已经看到了为命名构造函数提供参数的一个例子。常规构造函数也一样。

```
class ComicsModel {
    Map attributes;
    ModelEvents on;

    ComicsModel(attributes) {
        this.attributes = attributes;
        on = new ModelEvents();
    }
    // ...
}
```

Dart 语言提供了给实例变量赋值的非常方便的方法。在前面的例子中，我们是在构造函数块中对 `this.attributes` 做了赋值，而更简单的方法是直接用 `this.attributes` 来声明参数。

```
class ComicsModel {
  Map attributes;
  ModelEvents on;
  ComicsModel(this.attributes) {
    on = new ModelEvents();
  }
  // ...
}
```

这样做使表达的意图非常清晰。与把实例变量的赋值和构造函数中其他初始化和赋值混在一起相比，放在参数列表中使意图变得非常清楚。这样，构造函数的函数体只需关心自身，做好创建类实例需要的事情。

这种在参数列表中声明实例变量的惯例甚至也可以用在可选参数中。

```
class ComicsView {
  ComicsCollection collection;
  ComicsModel model;
  Element el;

  ComicsView({el, this.model, this.collection}) {
    if (el != null) {
      this.el = (el is Element) ? el : document.query(el);
    }

    this.post_initialize();
  }
  // ...
}
```

在这个例子中，构造函数做了一点儿真正的工作（如果没有提供第一个参数，那么就从文档中查找一个元素）。由于声明了 `this.model` 和 `this.collection` 作为可选参数，给它们赋值的意图很清晰——不需要在函数体中增加麻烦。

可选的赋值参数用被赋值的实例变量的名字传递。

```
var comics_view = new Views.Comics(
  el: '#comics-list',
  collection: my_comics_collection
);
```

像命名构造函数一样，Dart 语言的赋值参数帮你保持代码整洁。

5. 工厂构造函数

Dart 语言定义了一类返回特定对象实例的特殊构造函数。目前我们所见过的构造函数都是在操作刚刚创建出来的对象的内部状态，但是对象创建本身和返回这个新创建的对象是由 Dart 完成的。

```
class ComicsCollection {  
  ComicsCollection() {  
    on = new CollectionEvents();  
    models = [];  
  }  
  // ...  
}
```

如果通过 `new ComicsCollection()` 实例化一个对象，会得到一个 `ComicsCollection` 类的对象，其中的 `on` 和 `models` 实例变量是它们的初始状态。这涵盖了 90% 的面向对象编程，但有时我们会有更多的要求。

例如，如果我们不想要创建新对象怎么办？如果类应该返回一个以前组装的对象的缓存副本怎么办？如果需要类总是返回同一个实例怎么办？如果需要构造函数返回一个完全不同的对象怎么办？Dart 语言定义了工厂构造函数来回答所有这些问题。工厂构造函数和普通构造函数的语法差异在于 `factory` 关键字和构造函数的返回值。看一下 `Highlander` 类的工厂构造函数。

```
class Highlander {  
  static Highlander the_one;  
  String name;  
  factory Highlander(name) {  
    if (the_one == null) {  
      the_one = new Highlander._internal(name);  
    }  
    return the_one;  
  }  
  // 私有的命名构造函数  
  Highlander._internal(this.name);  
}
```

`Highlander` 的工厂构造函数检查类变量 `the_one` 是否已经定义了。如果没有，它会通过一个私有的命名构造函数创建一个新的实例赋值给它，并返回 `the_one`。如果 `the_one` 已经定义了，那么不会再创建新实例，而是返回之前定义的 `the_one`。

这样，我们就创建了一个单例类。

```
var highlander = new Highlander('Connor Macleod');
var another = new Highlander('Kurgan');

highlander.name;
// => 'Connor Macleod'

// 试试 Kurgan...
another.name;
// => 'Connor Macleod'
```

工厂构造函数的用途不限于返回当前类的缓存副本。它可以返回任何对象^①。

```
class PrettyName {
  factory PrettyName(name) {
    return "Pretty $name";
  }
}
```

这个简单的类的实例将是一个（Pretty 开头）字符串。

```
new PrettyName("Bob");
// => "Pretty Bob"
```

合理地使用，它会非常强大。

7.8 下一步做什么

Dart 语言提供了一些非常好的面向对象编程特性。语言中的大部分重点看起来都是为了使代码更整洁、意图更清晰而设计的。尽管它支持 `this` 关键字，表示当前对象，但是它的使用远不及在 JavaScript 中那么普遍，并且相关的规则并不难懂。有效地使用生成式构造函数、工厂构造函数和重定向构造函数在很大程度上使你的 Dart 代码尽可能保持整洁。

我们将在第 11 章再次讨论类。更具体地说，是一些关于在真实环境下更好地使用的启示。

^① 在检查模式下，要求返回的是当前类的子类型，而在生产模式下没有这种限制，可以返回任何对象。——译者注

第 8 章

事件

不管什么语言，浏览器事件都是按同样的方式被捕获、接收和冒泡（bubble）的。所以，Dart 中事件的行为是应该类似于 JavaScript 风格的，再加上一点 Dart 自身的特色。

8.1 普通事件

例如，考虑下面这个点击事件的处理程序，它把被点中的元素的边框变为亮橙色。

```
var el = document.query('#clicky-box');
el.on.click.add((event) {
  el.style.border = "5px solid orange";
});
```

这里，我们把一个匿名函数添加到发生点击事件时调用的回调函数的列表中。这比等价的 JavaScript 代码略紧凑一点：`el.addEventListener('click', callback_fn)`。不过这里是 Dart 语法实现的，特别是因为 Dart 语言是强类型的。

在添加事件监听器的链式调用中，首先遇到的是 `Element` 的 `on` 属性。`on` 属性是一个 `ElementEvents` 对象，它暴露了各种支持的事件所对应的 `getter` 方法。

定义 `on` 属性的 `ElementEvents` 类是 `Events` 的子类，所以在与其他基于事件的类一起工作时，我们可以期望类似于 `thing_with_events.on.event_name` 的

使用模式^①。例如，HttpRequestEvents 类暴露了 on 属性上的诸如 error、load 和 progress 这样的事件处理程序。实际上，在第 1 章中我们见过这种例子。Ajax 处理程序 on.load 看起来像下面这样：

```
req.on.load.add((event) {
  var list = JSON.parse(req.responseText);
  container.innerHTML = graphic_novels_template(list);
});
```

Ajax 的事件处理和元素的事件处理看起来一样，你猜对了，因为 Element 和 HttpRequest 都实现了同一个接口 EventTarget。EventTarget 接口只不过是要它的实现者提供一个 getter 方法 on，它自身实现了 Events 接口^②。这就是 Dart 的结构化代码中对于现代 Web 的“结构化”的组织。

继续看这个添加监听器的调用链 el.on.click.add() 中的 click。getter 方法 click 与 on 上面的其他任意属性一样，返回一个 EventListenerList 接口。这个接口暴露了 3 个很重要的方法：add()、remove() 和 dispatch()。前两个方法显然是用于在 EventListenerList 中添加和删除监听器的，而 dispatch() 方法是用于手工触发事件的。

如果需要触发一个点击事件，可以这样做：

```
el.on.click.dispatch(new Event("My Event"));
```

注意：在分发元素事件的时候，提供一个 Element 作为事件的目标被认为是一种最佳实践。

8.2 自定义事件系统

Dart 中有许多不同的事件系统。本章目前为止，已经探索过 ElementEvents 系统。不包括子接口，至少有 7 个事件系统：

- HttpRequestEvents
- WindowEvents

① dart:html 的事件处理都采用了这种模式 object.on.event.add(handler)，但在整个 Dart 语言中并非总是如此，比如在 dart:io 中使用的是 object.onEvent = stuff 模式。哪种设计更好是与使用环境相关的。——译者注

② 其实就是要求实现提供一个 Events 类型的 on 字段。——译者注

- DOMApplicationCacheEvents
- AbstractWorkerEvents
- EventSourceEvents
- HttpRequestUploadEvents
- ElementEvents

要构建自己的事件系统，我们需要按照同样的惯例暴露一个 `on` 属性。像内建的 `EventTarget` 对象一样，这个 `on` 属性将暴露一些定制的事件类型，其上可以添加和删除事件监听器。也就是说，它需要提供一组获得 `EventListenerList` 类型的不同的 `getter` 方法。

再次回顾 `ComicsCollection` 类，我们已经看到 `on` 属性是 `CollectionEvents` 类的一个实例。

```
class ComicsCollection implements Collection {
    CollectionEvents on;
    ComicsCollection() {
        on = new CollectionEvents();
    }
    // ...
}
```

`CollectionEvents` 类是我们自己设计的。它实现了内建的 `Events` 接口，并且暴露了 `load` 和 `insert` 作为事件监听器的容器。

```
class CollectionEvents implements Events {
    CollectionEventListener load_listeners, insert_listeners;
    CollectionEvents() {
        load_listeners = new CollectionEventListener();
        insert_listeners = new CollectionEventListener();
    }
    CollectionEventListener get load => load_listeners;
    CollectionEventListener get insert => insert_listeners;
}
```

当创建 `CollectionEvents` 类的新实例时，构造函数就会构建两个 `CollectionEventListener` 的实例——一个存放 `on.load` 事件监听器的列表，另一个存放新记录插入到集合中的事件监听器的列表。

关于描述 `on.load` 和 `on.insert` 的 `CollectionEventListener` 类，我们需要它实现 `EventListenerList` 接口。这是一个相对简单的类，它的主要目的是，维护一个在

特定事件发生时要触发的回调函数的列表。如本章前文所述，`EventListenerList` 接口要求我们定义 `add()` 和 `remove()` 方法，用于向内部的监听器列表添加和删除回调函数。它还需要定义一个 `dispatch()` 方法，用于在一个事件生成时能够调用所有的监听器。

```
class CollectionEventListener implements EventListenerList {
    List listeners;

    CollectionEventListener() {
        listeners = [];
    }

    CollectionEventListener add(fn) {
        listeners.add(fn);
        return this;
    }

    bool dispatch(CollectionEvent event) {
        listeners.forEach((fn) {fn(event)});
        return true;
    }
}
```

在 `CollectionEventListener` 的定义中要特别注意的是，`add()` 方法返回它自己的当前实例。这允许我们一次添加多个回调函数。

```
hipster_collection.
  on.
  insert.
  add(event) { /* 第一个监听器 */ }.
  add(event) { /* 第二个监听器 */ }.
  add(event) { /* 第三个监听器 */ };
```

注意：这不允许我们在一个语句中同时添加不同类型的事件。在前面的例子中，`add()` 方法返回的 `CollectionEventListener` 实例是专门处理插入事件的。要定义一系列加载事件的回调函数，我们还需要另外一条 `hipster_collection.on.load.add()` 语句。

最后，我们看一下要生成的实际事件。在一个自定义的事件里，真正唯一需要做的是定义一个 `getter` 方法 `type`，它通常与事件监听器列表的名字一样（如 `load` 和 `insert`）。为了满足 `ComicsCollection` 的需要，我们还需要收集集合对象自身以及可选的模型对象（在插入和删除时可以很方便）。

```
class CollectionEvent implements Event {
  String _type;
  ComicsCollection collection;
  ComicsModel _model;

  CollectionEvent(this._type, this.collection, [model]) {
    _model = model;
  }

  String get type => _type;

  ComicsModel get model => _model;
}
```

为了更符合 Dart 语言的惯例，这个类也应该暴露一个 `target` 方法。

```
class CollectionEvent implements Event {
  // ...
  EventTarget get target => collection;
}
```

反过来，这表明我们的 `ComicsCollection` 类要实现 `EventTarget` 接口。

```
class ComicsCollection implements Collection, EventTarget {
  // ...
}
```

我们已经看到了这个事件系统是如何使用的。当添加了新的模型或从后端加载完成时，`ComicsCollection` 基类会分发这些事件。

```
class ComicsCollection implements Collection {
  // ...
  add(model) {
    models.add(model);
    on.insert.
      dispatch(new CollectionEvent('add', this, model));
  }
  _handleOnLoad(list) {
    // 在这里添加操作
    on.load.dispatch(new CollectionEvent('load', this));
  }
}
```

通过监听这些事件，在漫画书集合从后端数据存储中加载完成或者有一个新的记

录添加到集合时，Comics 视图集合就更新显示。

```
class ComicsView {  
  // 被构造函数调用  
  _subscribeEvents() {  
    if (collection == null) return;  
    collection.on.load.add((event) { render(); });  
    collection.on.insert.add((event) { render(); });  
  }  
}
```

像这种基于事件的方法的主要好处在于，它优雅地分离了关注点。集合不需要知道视图的任何事情。集合在正常工作过程中仅仅是分发它的事件——另一边，视图只关注事件通知，这样它就能立即更新自己。

8.3 下一步做什么

Dart 语言显示了丰富的能够满足开发者需要的事件系统。由 `Events`、`HttpRequestEvents` 和其他类实现的事件系统很容易使用而且符合直觉。当这种简单的机制不能满足需要时，Dart 语言允许我们定义自己的事件系统。

在第 14 章中，我们将讨论让分离的代码块进行通信的另一种手段 `Future` 和 `Isolate`。它们都有助于保持代码被良好地分解和维护。

第三部分 代码组织

随着我们第一次体验到 Dart 语言的强大，现在是时候看一些 Dart 语言真正独特的地方了——库系统。之前，我们自己构建了一个 MVC 库，似乎我们仍然受 JavaScript 的限制。就是说，我们把所有的东西都放到了一个文件中。Dart 语言自身包含了一个精致的内建的库系统。正如我们将看到的那样，这意味着编写大型的库不仅可能而且很容易。

第 9 章

项目：提炼库

回想第 6 章，我们把一个简单的 Dart 应用程序重写为 MVC 风格的，类似于知名的 Backbone.js。这样，我们很难重用这些代码——无论是用于自己的代码库还是共享给其他人。

在本章中，我们将把这些 MVC 类分解为一个可重用的库。这会涉及两个独立的工作：利用我们刚掌握的 Dart 语言的面向对象的技能和 Dart 语言出色的库系统。最终的结果将促进代码的复用性以及更好的代码可维护性。

在本章中，我们也将涉及一些其他多数语言书籍中不会有的内容：正在讨论的语言的一些实际局限性。

9.1 要提炼什么，要保留什么

对于第 6 章中的每一个集合、模型和视图的类，我们现在要面对的是提炼什么问题。

1. 集合重构：除了硬编码的所有东西

正如在第 6 章所做的，我们首先从客户端 MVC 库的核心开始：漫画书集合。因为集合是到 REST 后端上松散的代码映射，我们应该能够从 ComicsCollection 中提炼许多东西到超类 HipsterCollection 中。任何特定于漫画书的部分，如 URL/comics，都应该保留在 ComicsCollection 中。（理想情况下）剩余的部分可

以拿出来与其他 REST 风格的后端应用共用。

除了超类 `HipsterCollection` 中的部分，`ComicsCollection` 应该是像下面这样：

```
class ComicsCollection extends HipsterCollection {
  // url => 集合的根 url
  // 其他漫画书特定的方法
}
```

如果我们把所有东西移动到 `HipsterCollection` 中，那么一开始它应该看起来像下面这样：

`mvc_library/collection_skel.dart`

```
class HipsterCollection implements Collection {
  var on, models;

  // 构造函数
  HipsterCollection() {
    on = new CollectionEvents();
    models = [];
  }

  // TODO: 从子类获得 URL
  // MVC 相关的
  fetch() { /* ... */ }
  create(attrs) { /* ... */ }
  add(model) { /* ... */ }

  // 集合相关的
  void forEach(fn) { /* ... */ }
  int get length { /* ... */ }
  operator [](id) { /* ... */ }
}
class CollectionEvent implements Event { /* ... */ }
class CollectionEvents implements Events { /* ... */ }
class CollectionEventList implements EventListenerList { /* ... */ }
```

除了构造函数重命名为 `HipsterCollection()` 以外，很少有需要改变的。事实上，我们计划中需要改变的是获取 URL 的方面，以及一个 `create()` 方法，用它创建模型就不再需要硬编码的 `new ComicsBook()`。

为了使子类能够告知超类 `HipsterCollection` 具体的 URL 是什么，我们的第一反应是在超类中设置一个属性，并交给实现者去定义。

```
class HipsterCollection implements Collection {  
  var url;  
  // ....  
}
```

然后子类可以定义这个 `url`。

```
// 这么做有问题!!!  
class ComicsCollection extends HipsterCollection {  
  var url = '/comics';  
}
```

但是，这样做不行。因为在 Dart 语言中，定义在子类中的实例变量对超类不可用。如果实例变量是在构造函数或方法中设置的，那么超类就可以看到变化。在前面我们尝试在子类中定义它。这样，如果 `HipsterCollection` 尝试访问 `url` 属性，那么它不会看到具体类 `ComicsCollection` 中定义的这个属性。所以，我们需要把它改为一个 `getter` 方法^①。

`mvc_library/comics_with_url.dart`

```
class ComicsCollection extends HipsterCollection {  
  get url => '/comics';  
}
```

因为 `getter` 方法 `url` 需要在超类 `HipsterCollection` 中引用，但是又必须在子类中定义，所以我们应该把它声明为抽象方法。

`mvc_library/collection_with_abstract_url.dart`

```
abstract class HipsterCollection implements Collection {  
  String get url;  
  // ...  
}
```

这样 `fetch()` 方法不需要修改就可以工作。`getter` 方法 `url` 看起来就像是类中的另一个实例变量。

说完 URL，让我们把注意力转移到一个机制，使具体类能够告诉超类如何创建模型对象。这比处理 `getter` 方法 `url` 更棘手。

^① Dart 语言中子类可以覆盖超类的实例方法和 `getter/setter` 方法，但是不能以字段的形式覆盖超类中的字段，这样会提示与超类中的实例变量冲突，而应该用 `getter/setter` 方法覆盖超类的字段，或者在子类构造函数中对其赋值。——译者注

例如，在 `Backbone.js` 中，通过一个属性传递要创建的模型类。

`mvc_library/backbone_sub_class.js`

```
var Comics = Backbone.Collection.extend({
  model: ComicBook
});
```

但是在 `Dart` 语言中这样做不行，因为类不是顶级对象。所以，没办法把一个类名赋值给一个变量或者用它作为 `Hash/Map` 的值。所以，我们只能用一个工厂方法解决，给定模型的属性，返回一个我们需要的模型的新实例。也就是说，我们要添加一个 `modelMaker()` 方法。

`mvc_library/comics_with_model_maker.dart`

```
class Comics extends HipsterCollection {
  get url => '/comics';
  modelMaker(attrs) => new ComicBook(attrs);
}
```

回到超类 `HipsterCollection`，我们也把它声明为抽象方法，同时更新 `create()` 方法来使用这个方法。

`mvc_library/collection_with_abstract_model_maker.dart`

```
abstract class HipsterCollection implements Collection {
  HipsterModel modelMaker(attrs);
  // ...
  create(attrs) {
    var new_model = modelMaker(attrs);
    new_model.save(callback:(event) {
      this.add(new_model);
    });
  }
}
```

`getter` 方法 `url` 和 `modelMaker()` 方法都是 `JavaScript` 中容易而 `Dart` 中难的例子。但这些不是一成不变的——`Dart` 可能不久以后会支持这两种用例中的一种或全部更容易地实现。

有局限性的原因只是简单的优先级问题。`Dart` 语言设计者青睐于定义一个结构良好的、传统的、面向对象的范式而不是把类作为顶级对象。他们青睐于尽量封装实例变量而不是在类之间共享定义，并且他们的选择似乎得到了良好地支持，只要我们的“变通方法”能够是单行小程序。

2. 模型重构：这里没什么

HipsterModel 类的实现甚至更简单——所有东西都搬过来，子类除了重定向构造函数外什么都不用做。

mvc_library/model_comic_book.dart

```
class ComicBook extends HipsterModel {  
  ComicBook(attributes) : super(attributes);  
}
```

我们再次遇到需要把子类构造函数参数显式地传递给超类的情况^①。除此之外，基类 HipsterModel 会处理好一切的（还记得，url 来自于集合对象）。

如果要直接使用 HipsterModel 的实例，我们还需要在子类中覆盖 getter 方法 urlRoot。

mvc/model_url_root.dart

```
class ComicBook extends HipsterModel {  
  // ...  
  get urlRoot => 'comics';  
}
```

这并不是说 HipsterModel 很简单，它仍然要负责更新和删除记录，同时负责生成集合和视图可以监听的事件。在从 ComicBook 提炼代码到 HipsterModel 中的时候，并没有遇到什么奇怪或复杂的地方。

3. 视图和初始化

可选（参数）构造函数语法的一个不足之处是，需要我们在子类中显式地委派它。

mvc/view_sub_class_constructor.dart

```
class Comics extends HipsterView {  
  Comics({collection, model, el}):  
    super(collection:collection, model:model, el:el);  
}
```

好吧，虽然这不是非常麻烦，但是如果未来版本的 Dart 语言能够把它缩短为一行就更好了。

^① 因为超类中没有无参数的默认构造函数，只有无参数的默认构造函数才能省略。——译者注

视图需要能够监听模型事件和集合事件，但这与子类的定义非常相关。为了适应这一点，我们需要在构造函数中调用一个 `post_initialize()` 方法。

mvc/view_sub_class_constructor2.dart

```
class HipsterView {
  // ...
  HipsterView({el, this.model, this.collection}) {
    this.el = (this.el is Element) ? el : document.query(el);
    this.post_initialize();
  }
  void post_initialize() { }
```

`post_initialize()` 方法可能看上去最好定义成一个私有方法，但是 Dart 语言不允许从超类中访问子类的私有方法。

要点：在 Dart 语言中，所有的私有方法都是相对于定义它的库而言的^①。

这样，我们可以像下面这样定义 Comics 视图：

mvc/view_sub_class_constructor3.dart

```
class ComicsViews {
  // ...
  ComicsView({this.el, this.model, this.collection}) {
    _subscribeEvents();
    _attachUiHandlers();
  }
}
```

私有方法 `_subscribeEvents()` 和 `_attachUiHandlers()` 与第 6 章中的一样，前者是用于在屏幕上显示漫画书集合的事件处理程序，后者是从集合中删除漫画书的 UI 事件处理程序。不论哪种情况，它们都是特定于漫画书应用的，而不属于通用 MVC 库。

9.2 真正的库

此时我们的 `main.dart` 文件变得非常拥挤。里面有 `main()` 函数入口点、

^① 在 Dart 中库是私有单元，在同一个库中可以访问彼此的私有方法，但在其他库中不可见。所以上面说 `post_initialize` 不能定义为私有方法是相对库的使用者而言的。——译者注

HipsterCollection（和相关的事件类）、HipsterModel（和相关的事件类）、HipsterView 和各种具体类。抛开太大不利于维护不说，我们将如何实现可重用性呢？

为了解决这两个问题，我们需要把类移动到另外一个单独的文件中。为了将来的重用和可维护性，我们将利用 Dart 语言内建的对库的支持，使这种转变既平滑又定位恰当。

从 HipsterCollection 开始，我们先创建一个 HipsterCollection.dart 文件。为了使它成为一个正式的 Dart 库，我们需要以 library 指令开始^①，还需要用 import 指令显式地导入 HipsterCollection 所必需的核心包。

mvc_library/collection_library.dart

```
library hipster_collections;

import 'dart:html'; // 事件相关的类需要
import 'dart:json';

class HipsterCollection implements Collection {
  // 这里实现集合...
}
```

当我们把 ComicsCollection 类提取到它自己的 ComicsCollection.dart 文件中时，也需要声明一个 library 语句，还需要用 import 导入 HipsterCollection 类，这样才能继承它。定义一个 HipsterCollection 基类的子类就是要继承 HipsterCollection 并定义那两个抽象方法。它看上去应该像下面这样。

mvc_library/collection_comics.dart

```
library comics collection;

import 'HipsterCollection.dart';
import 'Models.ComicBook.dart';

class Comics extends HipsterCollection {
  get url => '/comics';
  modelMaker(attrs) => new ComicBook(attrs);
}
```

^① library 指令用于声明一个库，后面是库的名字。每个 Dart 应用程序也是一个库，即使它并没有使用 library 声明。——译者注

注意：我发现在 MVC 的文件名中最好使用匈牙利命名法（Hungarian notation），例如 `Models.ComicBook.dart` 和 `Collections.ComicBook.dart`，但不要用在类名上，因为那会使代码很嘈杂。

下一章我们再介绍 `library` 指令的细节，它非常强大。本章中，我们最后要讨论的是，在把所有东西都移动到单独的库文件中之后，`main.dart` 入口点会变成什么样。

`mvc_library/main.dart`

```
import 'Collections.Comics.dart' as Collections;
import 'Views.Comics.dart' as Views;
main() {
  var my_comics_collection = new Collections.Comics()
    , comics_view = new Views.Comics(
      el: '#comics-list',
      collection: my_comics_collection
    );

  my_comics_collection.fetch();
}
```

我们实例化了一个漫画书集合，然后把它和列表的 DOM ID 传递给视图的构造函数。最后，我们获取集合的数据，并在适当的时候通过各种事件触发视图来显示自己。

注意 `import` 语句中的 `as` 选项。为了使代码尽可能保持整洁，`Comics` 视图集合和 `Comics` 集合都是以 `Comics` 作为类名定义的。

```
// Collections.Comics.dart
class Comics extends HipsterCollection { /* ... */}

// Views.Comics.dart
class Comics extends HipsterView { /* ... */ }
```

如果以 `ComicsView extends HipsterView` 这样的方式声明视图，名字似乎显得过于冗长了。但是，如果不这样，当二者在同一个上下文中使用的时候，类定义冲突就是非常现实的问题。此时，使用 Dart 语言中 `import` 语句提供的 `as` 选项就非常方便。

以 `Collections` 为前缀导入 `Collections.Comics.dart`，所有的顶级类定义现在都要以 `Collections.*` 为前缀引用。现在要实例化一个集合对象，使用 `new`

`Collections.Comics()`。这对代码的组织有巨大帮助，特别是与定义了许多库的应用程序打交道的时候（这在 MVC 应用程序中很典型）。

注意：对于好奇者，这个 MVC 库的最终版本在 <https://github.com/eee-c/hipster-mvc/> 可以找到。

9.3 下一步做什么

我们在这里很好地利用了第 7 章的面向对象知识。同时我们暴露了一点 Dart 语言的不足。比如，不能把类名作为变量传递，在子类中不能以实例变量的形式覆盖超类。超类不能（在不同的库中）访问子类的私有方法。甚至一些现实问题可能是我们在面向对象代码中所期望的，但 Dart 对这些问题有自己的理由，正如我们将在接下来的几章要探索的。

尽管存在一些局限性，我们也发现一些相当普通的变通方法。通过变通方法，我的意思是还不错的“Dart 方式”，在面向对象的重构中我们有效地利用了 Dart 语言出色的库系统。

在浏览器中，管理代码重用和可维护性似乎是不可能征服的挑战，然而 Dart 处理库却是很容易掌握的。通过分解 MVC 库并把类提取到它们自己的文件中，我们可以很容易地找到和维护代码库的特定方面。同时，我们也没有牺牲使用代码的易用性。除了引入了少量的 `import` 和 `library` 语句外，与当初都在一个大文件中相比，我们的代码并没有什么改变。

在下一章，我们将更深入地了解库，然后讨论 `part`，和类似的 `import` 语句。当我们在第 11 章回到这个项目时，我们将把它改为使用本地存储。

第 10 章

库

JavaScript 已经有 17 年的历史了。在这段时间里，它一直缺少一个简单的库的加载机制。这并不是因为缺少需求。有许多独立的解决方案，甚至有若干尝试性的标准（CommonJS^①、AMD^②和 ECMAScript harmony modules^③）。随着这些标准在各种实用性和采纳状态中停滞不前，社区已经产生了更多的加载插件。

尽管有这些尝试，但加载额外 JavaScript 库的最可靠的方式还是利用额外的<script> 标签，并且将合并多个文件到一个压缩过的 JavaScript 脚本中。没有一种解决方案是没有问题的（如加载顺序和部署复杂性）。

非常幸运，Dart 语言有内建的库的概念。更好的是，它很容易使用。Dart 语言当前支持两种将功能导入到 Dart 代码中的方法：part 和 import。

10.1 part 语句

part 指令用于把任意 Dart 代码块包含到当前的上下文中。例如，我们可能有一个漂亮的打印函数以及一些局部变量存储在 pretty_print.dart 中。

libraries/pretty_print.dart

```
var INDENT = '  ';  
pretty_print(thing) {
```

① <http://www.commonjs.org/>。

② <https://github.com/amdjs/amdjs-api/wiki/AMD>。

③ <http://wiki.ecmascript.org/doku.php?id=harmony:modules>。

```
if (thing is List) print("${INDENT}${Strings.join(thing, ', ')} ");
else print(thing);
}
```

对于要被包含到另一个文件中的源文件，它并没有什么特别的要求^①。要实际包含它，我们只需简单地提供源文件的路径就可以了。路径可以是相对的也可以是绝对的。

libraries/print_things.dart

```
part 'pretty_print.dart';

main() {
  var array = ['1', '2', '3']
    , hash = {'1': 'foo', '2': 'bar', '3': 'baz'}
    , string = "Dart is awesome";

  pretty_print(array);
  pretty_print(hash);
  pretty_print(string);
}
```

这会产生下面的输出：

```
1, 2, 3
{1: foo, 2: bar, 3: baz}
Dart is awesome
```

如果需要重用这个方便的打印函数而不复制和粘贴，我们只需在任何需要它的源文件中使用同样的 `part` 语句即可。

限制

`part` 功能的一个诱惑是用它把一组方法混入到一个类中。因为 Dart 缺少多继承^②，所以这是一种绕过该限制的方法。但是，`part` 语句（也包括 `import` 语句）必须声明在源文件的顶部。

因此，像下面这样就不行：

① 被包含的文件中不能使用指令，如 `import` 和 `library`，除了 `part of`。每个库可以由多个文件组成，它们通过 `part` 被包含到库的主文件中，由这一个文件包含库需要的所有指令。——译者注

② 将来 Dart 语言可能会支持 Mixins 方式的多继承，详见：<http://news.dartlang.org/2012/08/darts-modest-proposal-for-mixins.html>。——译者注

```
class Circle {
  part 'pretty_print.dart';
  var x,y;

  Circle.pretty(this.x, this.y) {
    pretty_print(this);
    pretty_print(x);
    pretty_print(y);
  }
}
```

使用 `noSuchMethod` 的方式可以绕过这种限制，但是这样不太好^①。

10.2 import 语句

更感兴趣的是 `import` 语句，它允许我们在代码中导入要使用的类。从一开始，Dart 语言就支持这个在所有服务器端语言中都有的标准功能。更好的是，它在浏览器中也一样能用。

与 `part` 语句不同，被导入的源文件要求在文件顶部必须有 `library` 语句。例如，考虑一个用于程序计时的秒表（`stopwatch`）类，它提供了漂亮的打印功能。

`libraries/pretty_stop_watch.dart`

```
library stopwatch;

class PrettyStopwatch {
  Stopwatch timer;
  PrettyStopwatch() {
    timer = new Stopwatch();
  }
  start() {
    timer.start();
  }
  stop() {
    timer.stop();
    print("Elapsed time: ${timer.elapsedMilliseconds}ms");
  }
}
```

^① <http://japhr.blogspot.com/2012/01/dart-mixins.html>。

`library` 语句后面的标识符是库的名字，可以在名字中使用下划线做分隔，如 `web-components`。

如果我们想要知道计数一千万次需要多少时间，我们应该这样使用这个漂亮的秒表类：

libraries/time_counting.dart

```
import 'pretty_stop_watch.dart';

main() {
  var timer = new PrettyStopwatch().start();
  for(var i=0; i<10000000; i++) {
    // 只是计数
  }
  timer.stop();
}
```

和 `part` 语句一样有用，`import` 语句有更多的潜力帮助我们组织代码。`Dart` 不仅擅长面向对象，而且它使得共享和重用类库非常容易，即使在浏览器中。

注意：`Dart` 很智能，只会对 `part` 和 `import` 的文件加载一次，不论它们出现在多少个不同的地方。

前缀导入

`import` 语句允许我们设置导入类的命名空间。即使我们一般只导入在 `Dart` 代码中直接需要的类，但命名冲突仍然很有可能发生。

例如，考虑一下在我们的 `MVC` 应用程序中同时需要 `Comics` 集合和 `Comics` 视图的情况。二者都被声明为 `Comics` 类（尽管它们分别继承于 `HipsterCollection` 和 `HipsterView`）。如果我们尝试直接导入它们，`Dart` 语言编译器会抛出一个已定义的异常。

要绕过这种潜在的限制，我们要在导入语句中加上前缀。

libraries/prefixed_imports.dart

```
import 'collections/Comics.dart' as Collections;
import 'views/Comics.dart' as Views;
import 'views/AddComic.dart' as Views;
```

这样做之后，我们就不能直接引用 `Comics` 视图类或 `Comics` 集合类，而是使用

Views.Comics 和 Collections.Comics。

libraries/using_prefixes.dart

```
main() {
  var my_comics_collection = new Collections.Comics(),
      comics_view = new Views.Comics(
        el: '#comics-list',
        collection: my_comics_collection
      );
}
```

前缀暗示在 Dart 中没有全局的对象命名空间。main()入口点有它自己的、与类和对象隔离的工作区。各种被导入的库都有自己的对象命名空间。这减少了很多代码组织的麻烦，是 Dart 语言鼓励我们编写干净代码的另一种方式。

10.3 核心 Dart 库

Dart 语言定义了一组核心库，它的文档可以在网站 <http://api.dartlang.org> 公开查看。在编写本书时，核心库包括 dart:core、dart:isolate、dart:html、dart:io、dart:json、dart:uri 和 dart:utf。每个库都定义了一些可以用在自己的应用中的公共类。

要使用这些核心库，与使用我们自定义的库一样，需要使用 import 语句。

```
import 'dart:html';
import 'dart:json';
```

包管理工具“pub”^①：尽管它还处于开发的早期阶段，但 Dart 语言已经包含了一个包管理工具 pub。最终这个工具将成为一个可以从中心资源库安装包的工具。它已经能够解析和安装包的依赖。对于本地 Web 开发，使用这个工具的一个早期示例在 <http://japhr.blogspot.com/2012/05/dart-pub-for-local-development.html> 可以找到。

10.4 下一步做什么

内建了组织代码的能力是 Dart 语言的重要改进。Dart 采用了轻量级的语法，使我们在面对杂乱的客户端代码时不再有借口了。

^① pub 已正式发布，更多信息请参考官方网站 <http://pub.dartlang.org/>。——译者注

第四部分 可维护性

在我们学习了如何组织 Dart 代码之后，现在是时候学习一些使代码保持可维护的策略了。首先，我们要更新 MVC 库，提供一些与远程后端（甚至是本地）进行数据同步的方法。然后，我们要学习一下 Dart 语言的新特性之一：测试。

第 11 章

项目：变化的行为

我们当中那些来自于动态语言背景的人期望在运行时能够执行各种动态的行为。只是基于状态对响应做修改并不能让我们满足，我们希望修改实现。

例如，在 JavaScript 中可以在任何时候替换对象原型上的方法。在 Ruby 中，没有什么能够阻止我们用 `lambda` 或 `Proc` 替换函数。当新手把它看作魔法的时候，我们则着迷于元编程。

在 Dart 语言中，没有那么多使用“魔法”的机会，但是仍有可能会使用。为了探索这个主题，我们再次回到漫画书目录应用。这次，我们把 Ajax 的后端调用替换为浏览器本地存储。

11.1 用 `noSuchMethod()` 改变类行为

在第 7 章我们首次遇到了 `noSuchMethod()` 方法。现在我们要试着将它用作在 MVC 库中改变 `save()` 方法的行为的一个手段。

回想一下，当 `HipsterModel` 调用 `save()` 方法时，它以 JSON 形式将自身属性发送到 REST 接口的数据存储中，并建立一个成功更新的处理程序。

`varying_the_behavior/xhr_save.dart`

```
class HipsterModel {  
  // ...  
  save([callback]) {
```

```

var req = new HttpRequest()
    , json = JSON.stringify(attributes);

req.on.load.add((event) {
    attributes = JSON.parse(req.responseText);
    on.save.dispatch(event);
    if (callback != null) callback(event);
});

req.open('post', '/comics', true);
req.setRequestHeader('Content-type', 'application/json');
req.send(json);
}
}

```

为了成功地用本地存储实现替换这个实现，我们需要在本地保存，确保调用同样的回调函数，并且确保分发同样的事件。要在 `localStorage` 中保存模型，我们要用新的或更新过的模型数据覆盖数据库的内存副本，并保存整个数据库。

varying_the_behavior/local_save.dart

```

class HipsterModel {
    // ...
    save([callback]) {
        var id, event;

        if (attributes['id'] == null) {
            attributes['id'] = hash();
        }

        id = attributes['id'];
        collection.data[id] = attributes;

        window.
            localStorage[collection.url]=JSON.stringify(collection.data);

        event = new Event("Save");
        on.save.dispatch(event);
        if (callback != null) callback(event);
    }
}

```

我们稍后再关心 `localStorage` 的细节。现在，我们已经手工用 `localStorage` 版本的 `save()` 方法替换了原来的 Ajax 实现。这并非是让别人使用这个库的长期办法。

一种可能的解决方法是创建 `HipsterModel` 的子类 `LocalHipsterModel`，由它来包含这种新的行为。不过，这种解决方法并不令人满意，因为它要求作为库作者的我们要构建和维护针对各种存储行为的子类，我们的用户可能会想要：`Ajax`、`LocalStorage`、`Indexed DB`、`WebSocket` 等。更糟的是，几乎不可能预期每个行为可能需要的所有特性。

要取代这种创建一系列子类的方式，我们可以使用 `noSuchMethod()` 方法。回想一下第 7 章，`noSuchMethod()` 方法是 Dart 中任何无法定位到被调用的方法的地方的最后手段。

Dart 语言用两个参数调用 `noSuchMethod()` 方法：被调用的方法名和提供给方法的参数的列表^①。任何 `noSuchMethod()` 方法的实现都应该做的第一件事是，确保接收的方法是它能够处理的已知方法。

```
class HipsterModel {
  // ...
  noSuchMethod(name, args) {
    if (name != 'save') {
      throw new NoSuchMethodException(this, name, args);
    }
    // ...
  }
}
```

在这里，如果是 `save()` 以外的任何方法被调用，我们就立刻抛出异常，来表明这里没有这个方法。

警告：当前没办法在类继承链中实现多个 `noSuchMethod()` 方法。如果 `ComicBook` 模型使用 `noSuchMethod()` 方法改变了保存算法，那么 `HipsterModel` 基类中的 `noSuchMethod()` 方法就无法生效^②。这严重限制了 `noSuchMethod()` 方法合理使用的场景。特别是，它只能用在不会被扩展的具体类中。

做了适当的保护后，我们已经准备好了要调用的本地存储实现的保存方法或者 `Ajax` 实现的保存方法。这个方法把参数直接传给了处理对应行为的两个私有方法。

① 仅在当前 `noSuchMethod()` 方法的参数才是一个方法名和一个参数列表，将来可能会不同。根据语言规范，`noSuchMethod()` 方法的参数应该是一个 `InvocationMirror` 类型的对象，只是当前还没有实现。该对象描述了方法的性质，判断是常规方法还是 `getter/setter`。另外 `InvocationMirror` 还包括一个 `ArgumentDescriptor` 类型的字段，它包含了实际参数的详细信息：一个位置参数的 `List` 和一个命名参数的 `Map`。详细信息可以参考语言规范和这篇文章：<http://www.dartlang.org/articles/emulating-functions/>。由于规范中的这部分还没有实现，所以本书中的这部分是按照当前的情况介绍的。——译者注

② 因为子类覆盖了超类的这个方法，所以子类中的 `noSuchMethod()` 方法生效。——译者注

```
// *** 这么做有问题 ***
class HipsterModel {
    // ...
    noSuchMethod(name, args) {
        if (name != 'save') {
            throw new NoSuchMethodException(this, name, args);
        }
        if (useLocal()) {
            _localSave(args);
        }
        else {
            _ajaxSave(args);
        }
    }
}
```

然而，这样有问题，因为`_localSave()`和`_ajaxSave()`方法期望接收的是实际调用的参数。而在前面的实现中，我们传递的是一个包含了所有参数的 `List`。实际上，我们需要手工提取出参数，并放置到合适的参数位置上。

```
// *** 这样做仍然有问题 ***
class HipsterModel {
    // ...
    noSuchMethod(name, args) {
        if (name != 'save') {
            throw new NoSuchMethodException(this, name, args);
        }

        if (useLocal()) {
            _localSave(args[0]);
        }
        else {
            _ajaxSave(args[0]);
        }
    }
}
```

前面的代码仍然有问题。`HipsterModel` 的 `save()` 方法是使用一个可选的 `callback` 参数调用的^①。

① 还有个问题，因为是可选参数，所以如果没有传递这个参数的话，直接用 `args[0]` 就会出错。——译者注

```
new_model.save((event) {  
  this.add(new_model);  
});
```

`noSuchMethod()` 方法会忽略可选参数的标签。这样，它将把 `callback` 参数作为第一个参数。因为 `HipsterModel#save()` 方法只接收一个参数，所以这个不会有问题。

要点：在声明的方法中命名的可选参数可以以任意顺序放置，并且仍然是相同的含义，如 `new Comics(el: '#comics', collection: my_comics)` 与 `new Comics(collection: my_comics, el: '#comics')` 一样。这样，用 `noSuchMethod()` 方法动态实现的方法被限制为以特定的顺序调用。言下之意是，`noSuchMethod()` 方法只能用于参数数量固定的方法^①。

这样，我们的动态 `save()` 方法可以像下面这样实现^②：

```
class HipsterModel {  
  // ...  
  noSuchMethod(name, args) {  
    if (name != 'save') {  
      throw new NoSuchMethodException(this, name, args);  
    }  
    if (useLocal()) {  
      _localSave(callback: args[0]);  
    }  
    else {  
      _ajaxSave(callback: args[0]);  
    }  
  }  
}
```

正如我们所见，`noSuchMethod()` 方法的使用场景限于参数数量固定的具体类。同时也隐式地限于在一个类中使用。在这个 MVC 库的案例中，我们需要一种在模型和集合中同步数据的机制。

我们需要一个类，用来描述这种同步行为，并且允许开发人员按需注入不同的行为。

① 因为当前的 `noSuchMethod()` 方法只能知道实际调用顺序的参数的列表，而可选参数是可有可无的，并且命名参数的调用顺序又是不固定的，所以没办法正确地处理这些参数。因此只有固定位置（即非可选）的参数或者单个可选的参数才能正确地处理，尚不能正确处理可选或命名参数。——译者注

② 这要求把原来的方法 `save([callback])` 改为 `save({callback})` 的形式。——译者注

11.2 通过依赖注入实现同步

`noSuchMethod()` 方法是 Dart 语言中一个非常强大的工具，但是它受限于在单一的一个类或接口中使用。这种情况下，我们需要一种注入同步行为的机制，这种行为可以被模型和集合共享。

如第 7 章所见，Dart 语言不直接支持这样做。但是，也有办法。

让我们创建一个 `HipsterSync` 类，负责管理数据同步行为。最终，依靠 `HipsterSync` 的各种库将会调用静态方法 `HipsterSync.call()` 来分发“增删改查”操作。然而，在查看之前，我们需要一个执行 Ajax 请求的默认行为。

varying_the_behavior/hipster_sync_default.dart

```
library hipster_sync
import 'dart:json';

class HipsterSync {
  static _defaultSync(method, model, [options]) {
    var req = new HttpRequest();
    _attachCallbacks(req, options);
    req.open(method, model.url, true);

    // POST 和 PUT 的 HTTP 请求必须有请求内容
    if (method == 'post' || method == 'put') {
      req.setRequestHeader('Content-type', 'application/json');
      req.send(JSON.stringify(model.attributes));
    }
    else {
      req.send();
    }
  }
}
```

由于我们已经实现了最初的基于 Ajax 的应用，并且又将其转成了一个 MVC 库，所以现在这些看起来很正常。我们创建了一个 `HttpRequest` 对象，打开并发送请求。有点不同的是，我们需要支持用 POST 和 PUT 请求传递请求内容 (body)，用一个简单的条件判断就可以支持这种行为。

使用这个同步行为的所有类都需要能够在 `HttpRequest` 对象加载成功后分发事件。`_attachCallbacks()` 静态方法负责这件事。

varying_the_behavior/hipster_sync_default_callbacks.dart

```
class HipsterSync {
  static _default_sync(method, model, [options]) {
    var req = new HttpRequest();
    _attachCallbacks(req, options);
    // ...
  }

  static _attachCallbacks(request, options) {
    if (options == null) return;
    if (options.containsKey('onLoad')) {
      request.on.load.add((event) {
        var req = event.target,
            json = JSON.parse(req.responseText);

        options['onLoad'](json);
      });
    }
  }
}
```

这个 `_attachCallbacks()` 方法通过 `options` 参数中的 `onLoad` 回调函数重写了 `HipsterModel#save()` 方法。

varying_the_behavior/hipster_model_save_with_sync.dart

```
class HipsterModel {
  // ...
  save([callback]) {
    HipsterSync.call('post', this, {
      'onLoad': (attrs) {
        attributes = attrs;

        var event = new ModelEvent('save', this);
        on.load.dispatch(event);
        if (callback != null) callback(event);
      }
    });
  }
  // ...
}
```

这样，我们就把数据同步工作交给 `HipsterSync` 来完成。`HipsterSync.call()` 的前两个参数是说，在同步的时候应该使用 **POST** 方法，并从当前模型对象上获得要发送给后端存储的数据。

此时，我们可以查看 `HipsterSync.call()` 方法了。正如我们可能期望的，如果没有提供替代的同步策略，它就调用 `_defaultSync()` 方法。

varying_the_behavior/hipster_sync_call.dart

```
class HipsterSync {
  static call(method, model, [options]) {
    if (_injected_sync == null) {
      return _defaultSync(method, model, options);
    }
    else {
      return _injected_sync(method, model, options);
    }
  }
  // ...
}
```

有趣的行为是 `_injected_sync()` 方法。它看起来像另一个静态方法，但实际上它是一个类变量。用户库可以通过 setter 方法 `sync` 向库中注入这个函数。

varying_the_behavior/hipster_sync_injected.dart

```
class HipsterSync {
  static var _injected_sync;
  static set sync(fn) {
    _injected_sync = fn;
  }
  static call(method, model, [options]) {
    if (_injected_sync == null) {
      return _defaultSync(method, model, options:options);
    }
    else {
      return _injected_sync(method, model, options:options);
    }
  }
  // ...
}
```

要注入的函数需要接收和 `_defaultSync()` 函数一致的参数。

警告：同时有一个 setter 方法 `sync` 和一个类方法 `sync` 感觉更合理。但是，如果声明了一个与 setter 方法同名的方法，Dart 语言会抛出一个“已定义”的错误。因此，我们需要声明一个 setter 方法 `sync` 和一个静态方法 `call`。

所有这些都准备好了，让我们把 `HipsterSync` 策略切换为 `localStorage`。这

需要回到应用程序的 `main()` 入口点。目前，我们限制自己只支持 GET 操作。

`varying_the_behavior/main_with_local_sync.dart`

```
import 'HipsterSync.dart';

main() {
  HipsterSync.sync = localSync;

  // 建立集合和视图...
}

localSync(method, model, [options]) {
  if (method == 'get') {
    var json = window.localStorage[model.url],
        data = (json == null) ? {} : JSON.parse(json);

    if (options is Map && options.containsKey('onLoad')) {
      options['onLoad'](data.values);
    }
  }
}
```

这相当棒，只需一行就可以为整个框架注入一个完全不同的数据同步行为。

值得指出是，由于 Dart 语言对库的管理方式，在 `main.dart` 文件中设置 `Hipster.sync` 应该是下面这样：

```
// main.dart
import 'HipsterSync.dart';

main() {
  HipsterSync.sync = localSync;
  //
}
```

这将影响到 `HipsterSync` 的类变量 `_injected_sync`，它可被 `HipsterModel` 所见。

```
// HipsterModel.dart
library hipster models;

import 'HipsterSync.dart';
```

```
class HipsterModel {  
  //从 main.dart 能够看到 HipsterSync._injected_sync  
}
```

当然对 `HipsterCollection` 也是如此。

```
// HipsterCollection.dart  
library hipster collections;  
  
import 'HipsterSync.dart';  
  
class HipsterCollection {  
  // 从 main.dart 能够看到 HipsterSync._injected_sync  
}
```

`main.dart`、`HipsterModel.dart`、`HipsterCollection.dart` 和 `HipsterSync.dart` 这些都是独立的文件，并且 Dart 语言确保在一处定义的 `HipsterSync` 类可以被其他文件可见。在 JavaScript 中唯一与之等价的是 `Backbone.js` 所定义的 `Backbone.sync`。`Backbone` 会声明一个全局变量（例如 `Backbone`），并指导开发人员，需要在所有其他代码之前通过 `<script>` 标签把它包含进去。使用类似 `require.js` 的库会使你得到近似于 Dart 的行为，但是在语言一开始就这样做比事后 18 年再添加它要好很多。

11.3 下一步做什么

正如第 7 章提到的，Dart 语言不太重视元编程必需的动态语言特性。尽管如此，在 Dart 语言中仍能够实现一些相当漂亮的动态语言特性。`noSuchMethod` 方法确实是一种简单的方法，用它可以实现元编程的重要部分。虽然它只能用于实例方法，但是这应该可以覆盖开发人员 80% 的动态编程需求。如果不行，还可以利用 Dart 语言的函数式性质来实现更广泛的动态语言特性。

这就是说，在 Dart 语言中对于哪些可以用动态的方式实现还是有限制的。一旦语言规范中的反射和镜像部分实现完成，Dart 语言应该会成为动态语言程序员的理想语言。

第 12 章

测试

注意：本章描述的是 Dart 语言的“前沿”^①功能。Dart 语言差不多每个方面都在变化，但是这里使用的测试框架甚至还没进入常规的变化状态^②。即使如此，这也是有必要包含的重要主题。

随着 Web 应用变得越来越复杂，仅依靠类型检查来捕捉 bug 是不够的。在本章中，我们要测试这个 MVC 库，因为它已经发展得足够复杂，不能仅依靠类型检查了。

12.1 获得测试框架

当前，Dart 语言的测试框架位于 SDK 中的 `pkg/unittest` 目录。下面的命令将把测试框架导入到测试程序中：

```
import 'package:unittest/unittest.dart';
```

这样，我们就为测试做好了准备^③。

① 此处是双关语，原文是：bleeding-edge。它是 Dart 源码仓库上一个分支的名字，字面意思上表示“前沿”的功能，也表示是这个分支中的代码功能。——译者注

② 作者写作本书时，Dart 语言的单元测试框架部分刚出现不久，所以作者说这里与语言其他部分相比更可能有变化。事实上，之后确实有了一些变化，本章内容在翻译时尽可能修正为最新内容。另外，读者可参考官方最新的 API 和这篇介绍：<http://www.dartlang.org/articles/dart-unit-tests/>。——译者注

③ 当前单元测试框架（unittest）已作为 Dart 库的一部分，包含在 SDK 中。——译者注

12.2 2+2=5 应该出错

有趣的是，Dart 的测试框架能够并且打算直接运行在我们的应用程序中。它的输出将弹出在应用的正常显示之上。在开发进行的同时，能够及时获得构建正确的保证是非常有价值的。我们将从一个伪测试页面和一段 Dart 测试代码开始。

任何满足下面要求的 HTML 都可以：

- 加入要测试的 Dart 代码；
- 启动 Dart 引擎。

要测试 HipsterCollection 类的伪测试页面是下面这样的：

testing/tests/01.html

```
<html>
<head>
  <title>Hipster Test Suite</title>
  <script type="application/dart"
    src="HipsterCollectionTest.dart"></script>
  <script type="text/javascript">
    // 启用 Dart
    navigator.webkitStartDart();
  </script>
</head>
<body>
<h1>Test!</h1>
</body>
</html>
```

我们在 HipsterCollectionTest.dart 中导入两个需要的测试库和我们自己的源代码。它也需要声明 main() 入口点，因为我们没在别处声明。

testing/tests/HipsterCollectionTest.dart

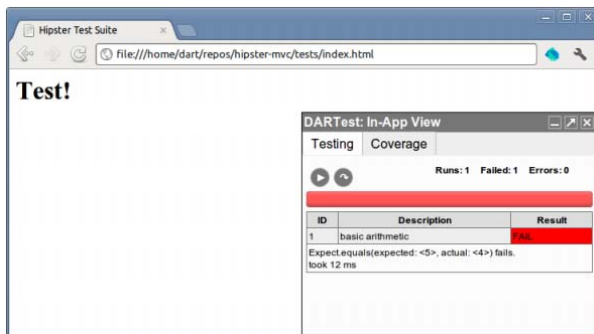
```
import 'package:unittest/unittest.dart';
import 'lib/darttest/darttest.dart';
import "../public/scripts/HipsterCollection.dart";
main() {
  // 在这里编写测试!
  new DartTest().run();
}
```

除了导入以外,我们创建了一个 `DARTest` 实例,并且用 `run()` 方法启动测试集^①。然而,没有测试代码之前什么也不会发生,所以下面我们写一个测试。在行为驱动开发(Behavior-Driven Development)的惯例中,都是以一个失败的测试开始的。

testing/tests/02test.dart

```
import 'package:unittest/unittest.dart';
import 'lib/darttest/darttest.dart';
import "../public/scripts/HipsterCollection.dart";
main() {
  test('basic arithmetic', (){
    expect(2 + 2, equals(5));
  });
  new DARTest().run();
}
```

要运行这个测试,我们要在浏览器中加载这个 HTML, 点击 `DARTest: In-App View` 弹出窗口中的运行按钮, 然后将看到下面所示的内容。



如果以（非 Web 的）默认形式运行, 会在命令行上产生如下输出:

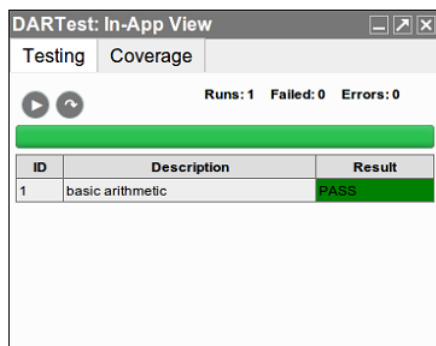
```
FAIL: basic arithmetic
  Expected: <5>
    but: was <4>.
```

一个失败的测试! 要让测试通过, 我们只需简单地修正这个数学计算。

```
test('basic arithmetic', (){
  expect(2 + 3, equals(5));
});
```

① 本章所用的 `darttest` 已经从 Dart 中移除, 不再使用, 所以本章中所述的这种在页面中的运行方式已经失效。代码中的删除线表示已失效的部分。除了这种运行方式外, 翻译时已经修正了原文的所有其他测试代码为最新内容。新的测试直接使用 `test` 方法, 默认是在标准输出中输出, 通过配置也支持其他形式的输出, 如 HTML 的输出结果。详见 <http://www.dartlang.org/articles/dart-unit-tests/#configuring-the-test-environment>。——译者注

重新加载页面会得到测试集通过的结果。



默认形式的输出是：

```
PASS: basic arithmetic
```

```
All 1 tests passed.
```

注意：每次代码或测试被修改后需要重新加载页面。这反映了测试集会在真实应用中，而不是在伪页面中存在。这背后的目的是为了开发者能同时看到实际应用和测试集。

既然我们已经有了编写测试的基本概念，让我们用 `HipsterCollection` 类的真实测试替换这个没意义的算数测试。

```
test('HipsterCollection has multiple models', () {
  HipsterCollection it = new HipsterCollection();
  it.models = [{ 'id': 17 }, { 'id': 42 }];

  expect(it.length, equals(2));
});
```

这是 `HipsterCollection` 类中对 `getter` 方法 `length` 的简单测试。`test()` 函数接收两个参数：一个用于描述测试的字符串和一个匿名函数，匿名函数中至少要包含一个预期 (`expect`)。在做了一点儿准备之后，我们使用 `expect()` 函数测试了相等条件的预期检查。

除了基本的相等测试，`Dart` 还支持许多方便的匹配方法^①，包括从近似范围 (`closeTo`) 到基本类型检查 (`isNull`、`isFalse`、`containsValue`)，甚至还支持异常检查。

① `expect` 方法的第一个参数是实际结果。第二个参数是一个 `Matcher`，用于判断结果与预期的匹配，默认的是 `isTrue`。前面我们用的 `equals` 就是一个 `matcher`，下面要用的 `matcher` 是 `throws`，表示预期会发生异常。`unittest` 中包含了各种常用的 `matcher`。——译者注

例如，我们可以验证，如果没有 URL，HipsterCollection#fetch()方法会失败。

```
test('HipsterCollection fetch() fails without a url', () {
  HipsterCollection it = new HipsterCollection();
  expect((){it.fetch();}, throws);
});
```

尽管这个测试框架还很年轻，但它已经可以在测试中对同类行为的测试做分组了。例如，我们可能需要测试 HipsterCollection 查找功能的两个方面。

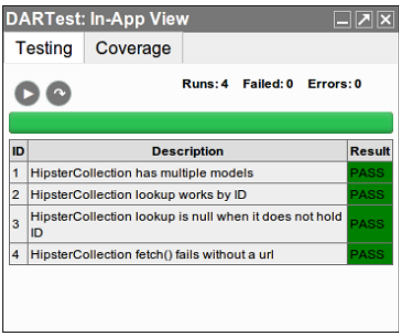
```
group('HipsterCollection lookup', () {
  var model1 = {'id': 17},
      model2 = {'id': 42};

  HipsterCollection it = new HipsterCollection();
  it.models = [model1, model2];

  test('works by ID', () {
    expect(it[17].values, equals([17]));
    expect(it[17].keys, equals(['id']));
    expect(it[17], equals(model1));
  });

  test('is null when it does not hold ID', () {
    expect(it[1], isNull);
  });
});
```

这样，我们有了一个通过了 4 个测试的测试集。



异步测试

Dart 语言像 JavaScript 一样，是函数式语言。函数式语言呈现出独特的测试挑战。内

建的预期在这里帮不了我们太多，但是测试框架可以。在这种情况下，我们选择的工具是 `expectAsyncN` 函数^①（后缀 N 是指回调函数中参数的个数，如 `expectAsync0`），它允许我们设置一个在回调函数中执行的预期。

例如，考虑添加一个新元素到 `HipsterCollection`。在这种情况下，预期的是任何插入事件的监听器都会被调用。如果我们只有一个监听器，那么我们期望这个回调函数被调用。这可以表示为：

```
test('HipsterCollection add dispatch add event', () {
  var newCallback = expectAsync0(() {
    // 原来的回调函数
  }, 1);
  // 这里执行测试调用...
});
```

`expectAsyncN` 函数将原回调函数包装在一个新的回调函数中，在测试代码中用这个新的回调函数替代原来的那个。测试框架直到回调函数被实际调用执行后才认为测试完成。`expectAsyncN` 的第一个参数是用于测试的原回调函数，第二个参数是一个名为 `count` 的可选参数，表示回调函数将被调用的次数，默认值是 1。

测试框架直到回调函数成功地调用了指定的次数后，才认为测试通过。如果出现异常或者超时则认为测试失败。

要测试插入事件，我们需要添加一个包装后的回调函数到 `on.insert` 监听器列表中，然后再添加一个新元素到集合中。

```
test('HipsterCollection add dispatches insert events', () {
  // 这里不关心数据同步
  noOpSync(method, model, [options]) {}
  HipsterSync.sync = noOpSync;

  HipsterCollection it = new HipsterCollection();

  var callback = expectAsync1((event) {
  }, 1);
  it.on.insert.add(callback);

  it.add({'id': 42});
});
```

① 当前，由于 Dart 语言的限制，不得不根据回调函数的参数个数使用不同的函数。将来计划只通过单一的一个 `expectAsync` 函数实现，而不管参数的个数情况。——译者注

这样，在 `HipsterCollection` 类中，我们有了 5 个相当有用的回归测试。

12.3 下一步做什么

尽管测试框架还很新，但测试集给出了重要的承诺。它不仅已经支持了常规测试，而且它也支持了有时更困难的异步测试。这个测试集可以运行在应用程序之上，它应该会帮助开发人员更快地实现回归测试。

第五部分 Dart 的高级使用

此时，我们已经很好地掌握了 Dart 语言的基本功能。接着，我们开始讨论 Dart 语言演化的下一步是什么。首先，我们通过删除回调函数来清除 MVC 库中最后的 JavaScript 思维的痕迹。回调函数——许多 JavaScript 程序员的痛苦之源——将被替换为 Completer、Future 和 Isolate，它们提供一种更简单的方式描述稍后将发生的事情。最后，我们讨论 Dart 语言对 HTML5 的支持和 Dart 的下一步方向。

第 13 章

项目：终结回调函数的地狱

影响很多大型 JavaScript 代码库的问题之一是，不可避免地纠缠于遍布在各处的回调函数。保持执行框架小型化非常有用，只需看看 Node.js 的崛起就能证明这一点。但是，甚至大部分有经验的 JavaScript 程序员都被无数的回调函数和事件所困扰。至此，我们的 MVC 库展示了许多与 JavaScript 方法同样的特征——并且，我们甚至不得不尝试错误处理！正如我们在第 8 章所见，Dart 事件的语法与 JavaScript 有所不同，但是方式非常相同。这不应是使用回调函数的场合。让我们看看 Future 是如何显著提高了 Dart 应用程序的长期可维护性的。

13.1 Future

在我们上次看到 HipsterModel 的时候，我们用了一个名为 HipsterSync 的数据同步层替换了直接的 Ajax 调用。在 save() 方法中，看起来像这样：

```
class HipsterModel {
  //...
  save([callback]) {
    HipsterSync.call('post', this, {
      'onLoad': (attrs) {
        attributes = attrs;
        var event = new ModelEvent('save', this);
        on.load.dispatch(event);
        if (callback != null) callback(event);
      }
    });
  }
}
```

```

    });
  }
  //...
}

```

这个方法至少有 3 个问题。首先，调用 `call()` 方法的参数太多——我们需要表明这是一个“post”方法，并且必须要有 `this` 参数，这样 `HipsterSync` 才能知道要同步什么——但是 `options` 参数仅描述了一种副作用，并且不是执行主线程所必需的。其次，`onLoad` 回调函数的可读性不太理想，被嵌到了 `options` 参数中。最后，每次尝试调用回调函数时都要进行 `null` 检查，这并不是很好的编程实践。

让我们使用 `Future` 对象来代替回调函数的方式。事实上，`Future` 只不过是形式化回调函数的对象。如果我们把 `HipsterSync.call()` 改为返回一个 `Future`，我们就可以通过 `then()` 方法注入回调函数。

```

class HipsterModel {
  //...
  save([callback]) {
    HipsterSync.
      call('post', this).
      then((attrs) {
        this.attributes = attrs;
        on.load.dispatch(new ModelEvent('save', this));
        if (callback != null) callback(event);
      });
  }
  // ...
}

```

这一处小改动使 `HipsterSync.call()` 的意图清晰了很多，它做的不过是 `POST this`（当前模型）到后端数据存储。一旦完成，我们将从数据存储的返回中获得属性并做下面这些事：

- 更新模型的属性；
- 通知所有监听器 `load` 事件已完成；
- 如果存在就调用回调函数。

这看上去要好一些，但是仍然有 `null` 检查的问题。有时候，像 `then()` 中这样的条件语句是必需的。更通常的情况是，`null` 检查表示我们的代码需要更好的抽象。这里就是这种情况。

我们可以让 `save()` 方法返回一个 `Future` 来代替可选的回调函数。生成一个

Future 对象的最简单的方法是，实例化一个新的 Completer 对象，它有一个 getter 方法 future 返回相关的 Future 对象。

```
Future<HipsterModel> save() {  
  Completer completer = new Completer();  
  // ...  
  return completer.future;  
}
```

这里，我们声明 save() 方法的返回类型是 Future<HipsterModel>，这明确地表示我们的 Future 将返回一个 HipsterModel 对象。就是说，成功保存后，save() 方法将发送回一个当前模型的副本给 then() 函数。例如，如果需要记录新创建的模型的 ID，我们可以这样使用 then() 方法：

```
var comic_book = new ComicBook({'title': 'Batman'});  
comic_book.  
  save().  
  then((new_comic) {  
    print("I got assigned an ID of ${new_comic['id']}");  
  });
```

我们仍然需要告诉 save() 方法，如何在发生事件的调用上下文中实际通知 then() 语句。这也是通过生成 Future 对象的 Completer 对象实现的。要通知 Future，对象 Completer 已经完成，我们用要发送给 then() 方法的值作为参数，调用它的 complete() 方法。

```
class HipsterModel {  
  // ...  
  Future<HipsterModel> save() {  
    Completer completer = new Completer();  
    HipsterSync.  
      call('post', this).  
      then((attrs) {  
        this.attributes = attrs;  
        on.load.dispatch(new ModelEvent('save', this));  
        completer.complete(this);  
      });  
    return completer.future;  
  }  
  //...  
}
```

当 HipsterSync.call() 成功完成时，下面就要更新模型的属性，分发事件并

标记 `save()` 方法完成。通过完成 `Completer`（即 `Completer#complete()`），调用 `save()` 方法的代码就可以在 `then()` 方法中去做它们需要做的事情了。这几乎像是英语阅读，真不错。

更重要的是，我们显著地改进了可读性，并因此提高了 `save()` 方法的可维护性。我们不再需要关心 `save()` 方法中的可选回调函数参数。现在只有两件事会影响 `HipsterSync` 的 `call()` 调用（请求的方法和模型）。没有了额外的 `Map` 对象 `options` 的干扰。最后，通过把这个方法变成一个 `Future`，我们消除了条件判断。这样做，使调用者更舒服。

13.2 Future 中的错误处理

到目前为止，我们简单地忽略了如何处理异常的问题。如果我们的 `POST` 请求遇到了后端的 400 系列的错误会发生什么？如何设计这个 `MVC` 库，以便开发者能够恰当地处理异常？

如果我们使用 `options`，答案可能是需要在 `options` 的 `Map` 中添加另一个回调函数。幸运的是，`Completer` 和 `Future` 有处理这种情况的正式机制。`Completer` 调用 `completeException()` 方法，通知出了问题，与此同时，`Future` 通过 `handleException()` 方法处理这个异常。

当请求状态不正确的时候（例如，状态值大于 299），`HipsterSync` 中默认的数据同步行为将通过 `completeException()` 方法通知一个异常条件给 `future`。

```
class HipsterSync {
    //...
    static Future _defaultSync(method, model) {
        var request = new HttpRequest(),
            completer = new Completer();
        request.
            on.
            load.
            add((event) {
                var req = event.target;
                if (req.status > 299) {
                    completer.
                        completeException("That ain't gonna work: ${req.status}");
                }
            })
            else {
                var json = JSON.parse(req.responseText);
            }
        }
    }
```

```

        completer.complete(json);
    }
  });
  //执行打开和发送请求
  return completer.future;
}
}

```

`completeException` 中的参数值并不要求一定是 `Exception` 的子类——可以是任何对象。在这里，我们提供了一个简单的字符串。

```

if (req.status > 299) {
  completer.
    completeException("That ain't gonna work: ${req.status}");
}

```

回到模型类上，我们需要处理这个异常情况。`then()` 方法返回 `void`，所以不能用链式调用。这要求我们把 `HipsterModel.save()` 方法返回的 `Future` 对象先保存为局部变量^①，然后在 `Future` 中注入一个 `then()` 的行为和一个 `handleException()` 的行为。

```

class HipsterModel {
  //...
  Future<HipsterModel> save() {
    Completer completer = new Completer();
    Future after_call = HipsterSync.call('post', this);

    after_call.
      then((attrs) { /* ... */ });

    after_call.
      handleException((e) { /* ... */ });
    return completer.future;
  }
}

```

因为 `HipsterModel#save()` 自身就是一个 `Future`，所以它应该用自己的 `completeException()` 来处理异常。

```

after_call.
  handleException((e) {

```

① 再次提示，Dart 语言通过级联调用（`cascade`）语法已支持在任意对象上继续连续调用，而无需要求方法返回自身对象才能做连续调用。——译者注

```
completer.completeException(e);  
return true;  
});
```

13.3 下一步做什么

正如我们将在第 14 章所见，`Future` 在其他地方也派得上用场。这很容易理解。尽管这个 MVC 库只是一个非常基本的应用，但我们已经显著地改进了库的可维护性，同时对开发者也很容易使用。

使用内建的对象来实现这种行为是 Dart 语言的重要改进。构建类似于 JavaScript 的东西并不难。虽然如此，内建的 `Future` 让我们关注于编写漂亮的代码，而不是关注于编写漂亮代码的代码。

第 14 章

Future 和 Isolate

除了支持熟悉的语法和概念，Dart 语言还带来了一些新特性。其中一类是高级的函数式编程特性，包括诸如 Completer、Future 和 Isolate。

14.1 Completer 和 Future

有一种概念是在稍后的某个时候完成一个任务，Completer 就是封装了这种概念的对象。Dart 语言允许我们在一个 Completer 对象中定义整件事情，而不是传递一个回调函数在将来被调用。

因为 Completer 在将来要触发一个动作，所以 Completer 与 Future 紧密相关。Future 对象中普遍要定义的是一个回调函数，在 Future 对象的 then() 方法中提供。

在最简单的形式中，我们可以创建一个 Completer 对象并从中获得它的 future 属性，这样当 Completer 结束的时候我们可以指定要做什么。最终，一段时间之后，我们用一个消息告诉 Completer 任务结束了。

```
main() {  
  var completer = new Completer();  
  
  var future = completer.future;  
  future.then((message) {  
    print("Future completed with message: $message");  
  });  
}
```

```
completer.complete("foo");  
}
```

它的最后结果是输出如下内容:

```
Future completed with message: foo
```

就它自身而言, Future 提供了一个重要而单一的功能。它只做一件事但是做得很好。

Completer 只能完成一次, 第二次尝试会抛出一个错误。例如:

```
completer.complete("foo");  
completer.complete("bar");
```

将产生下面的错误:

```
Future completed with message: foo  
Unhandled exception:  
Exception: future already completed
```

除了从 future 中发送成功消息外, 它也可能发出一个错误。要支持错误处理, Future 上需要定义一个 `handleException()` 回调函数。

```
main() {  
  var completer = new Completer();  
  
  var future = completer.future;  
  future.handleException((e) {  
    print("Handled: $e");  
    return true;  
  });  
  
  var exception = new Exception("Too awesome");  
  completer.completeException(exception);  
}
```

当 completer 以一个异常完成时, 结果如下:

```
Handled: Exception: Too awesome
```

如果 `handleException()` 回调函数没有返回 `true`, 那么异常就不会被捕获。相反, 它会沿着调用栈向上继续抛出, 直到在什么地方被捕获或者应用程序被强制结束。

14.2 Isolate

顾名思义，Dart 中的 Isolate 主要用于从主执行线程中隔离出要长期运行的函数。Isolate 使用 Future 对象通知调用线程它已经准备好了^①。

```
import 'dart:isolate';

main() {
  SendPort sender = spawnFunction(findDoom);

  print('Certain doom awaits...');

  var year = 2012;
  sender.call(year).then((message) {
    print("Doom in $year is on a $message.");
  });
}
```

我们稍后再看 findDoom() isolate。目前，认识到调用 spawnFunction() 会返回一个 SendPort，它暗示在某处会有一个对应的 ReceivePort，你可以认为它在 findDoom() isolate 中。

继续看这个调用上下文，在 sender SendPort 上调用 call() 方法会返回一个 Future。这样，我们就调用了这个长时间运行的 Isolate，并且不期望马上获得结果。当结果真正返回时，我们用提供的回调函数来处理它。在这里，我们只是简单地打印了结果。

正如前面承诺的，对应的 ReceivePort 以 port 的形式出现在 findDoom() 中。

```
import 'dart:isolate';

findDoom () {
  port.receive((message, replyTo) {
    var response = dayOfDoom(message);
    replyTo.send(response);
  });
}
```

像 spawnFunction() 方法一样，getter 方法 port 在 dart:isolate 库中是全

^① 所有的 Dart 代码都运行在一个 Isolate 上下文中（包括 main）。每个 Isolate 有它自己的堆内存，这意味着其中所有内存中的值，包括全局数据，都仅对该 Isolate 可见。Isolate 之间的通信只能通过传递消息的机制完成。——译者注

局的。如果调用 `spawnFunction()` 方法生成一个 `SendPort`，那么一定会给对应 `port` 生成一个相匹配的 `ReceivePort`。

调用的语义非常直观。我们在 `SendPort` 上调用 `call()` 方法来向 `Isolate` 发送一条消息。`ReceivePort` 是 `Isolate` 上用来接收来自发送者的消息的唯一方式，通过它的 `receive()` 方法。

在 `findDoom()` 方法中使用的计算“世界末日”的算法（`doomsday algorithm`）非常简单。它会找到 2 月的最后一天是星期几。利用简单的助记法，我们就能用它算出一年中任意一天是星期几^①。这只是一个用于演示的小算法。

```
final List<String> dayNames = const [
  'Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'
];

dayOfDoom(year) {
  var march1 = new Date(year, 3, 1, 0, 0, 0, 0)
  , oneDay = new Duration(days:1)
  , date = march1.subtract(oneDay);

  return dayNames[date.weekday - 1];
}
```

跟踪这个完整的消息传输过程，首先 `SendPort` 发送了要计算的年份（2012），然后 `ReceivePort` 会收到它，并调用 `dayOfDoom()` 计算结果。另一个 `SendPort` 被赋值给局部变量 `replyTo`，然后用它把调用的结果发送回去。最后，最初调用 `call()` 方法返回的 `Future` 就收到了回复。

调用这个 `Isolate` 的结果如下：

```
Certain doom awaits...
Doom in 2012 is on a Wed.
```

2012 年世界末日是星期三，所以星期三一定要小心点儿。

14.3 小结

`Completer`、`Future` 和 `Isolate` 可能是我们使用得相对较少的一类解决方案。即便如

^① http://en.wikipedia.org/wiki/Doomsday_rule。

此，它们肯定会被用到，并且当我们需要它们时，不需要再重新发明它们或者选择一个最合适的库，知道这一点很必要。

最新的 JavaScript 版本包含了 Web worker 的概念，但是如果我们需要支持较老的浏览器，就需要我们自己处理了。Node.js 的早期版本支持 promise，它和 Dart 中的 Future 很类似。最终，它们在 Node.js 中被删除了。这样导致每次有这种需求的时候就要重新发明 promise。

感谢在 Dart 语言中，Web worker 和 promise 从一开始就以 Isolate 的形式被支持了。

第 15 章

HTML5 和 Dart

在第 4 章，我们看到了许多例子，Dart 可以很容易地与 DOM 和样式进行交互和操作。本章在基本的 DOM 操作基础之上给出一个指导，通过动画、WebSocket 和其他各种属于 HTML5 的技术，给网页添加一点活力。

本章讨论的大部分是在 JavaScript 中已经实现了的特性。Dart 带来的是一个熟悉的、语法简单和跨平台的兼容性（不需要重复@-webkit 和@-moz 前缀）。

15.1 动画

如果你正在做交互，那么给 2012 年及以后的现代网站上用一点儿动画会大有好处。CSS 属性 transition 是这些较小的、看上去无害的附加功能之一，它实际上包含在相当多的功能中。例如，考虑一下我们的漫画书应用中的表单视图。在显示它的时候，使用淡入（fade in）效果显示会更好。

```
import 'dart:html';
import 'HipsterView.dart';

class AddComicForm extends HipsterView {
  //...
  render() {
    el.style.opacity = '0';
    el.innerHTML = template();
    window.setTimeout(() {
      el.style.transition = '1s ease-in-out';
```

```
        el.style.opacity = '1';  
      }, 1);  
    }  
  }
```

这个 `transition` 属性与 CSS3 中的一样^①。它是一组空格分隔的独立的转换属性，具体描述如下。

- 要应用的转换样式 (`all`、`opacity` 等)。在我们的例子中没包含这个，所以默认是 `all`。
- 动画的持续时间。我们的动画要持续 1 秒钟。
- 动画的功能。有几种可用的功能，包括 `ease`、`ease-in`、`ease-out` 和 `linear`。我们使用的 `ease-in-out` 功能开始慢，然后加速，最后减缓完成。
- 动画开始之前的延迟。因为我们没有包括这个，默认是无延迟（零秒钟）。

注意：在指定初始样式、转换和结束状态时，Dart 有个把转换“优化”掉了的习惯。作为一种变通方法，我们把转换和最终状态放到一个 `setTimeout()` 中。这会有效地将动画带出正常的同步工作流，刚好可以使动画开始生效。

15.2 本地存储

Dart 对于客户端存储的支持还有点儿不稳定，但是正如在第 11 章所见，我们已经可以执行 `localStorage` 了，这已足够了。尽管它是同步的，会有点儿慢，但 `localStorage` 是被支持得最广泛的客户端存储方案——并且是 Dart 语言当前唯一支持的^②。

因为它是同步的（也就是说，它的操作会阻塞客户端应用中的其他活动），所以它不太适合大的数据存储。不过，它很适合小数据集和开发原型。它的同步性质的好处是：使用起来很简单、不繁琐。

用于 `localStorage` 的 API 基本上与传统 JavaScript 的 API 一致。分别存储一个集合中的各个对象对于二者都是低效的。相反，我们用 `List` 或 `Map` 的 JSON 序列化表示来存储。

```
var json = window.localStorage['Comic Books'],  
    comics = (json != null) ? JSON.parse(json) : [];
```

向 `localStorage` 中添加一个记录只是简单地更新反序列化后的数据，再重新序列化并整体存储到数据库中。

^① <https://developer.mozilla.org/en/CSS/-moz-transition>。

^② 现在也支持 `IndexedDB`。——译者注

```
// 这里有个输入错误, 应该是'Sandman', 我们稍后修正它
comics.add({'id':42, 'title':'Sandmn'});
window.localStorage['Comic Books'] = JSON.stringify(comics);
```

这会替换原先存储在 localStorage 中的漫画书数据项。

要更新本地存储中的数据也不过是更新本地表示的数据项, 然后再序列化为 JSON 存储到数据存储中。

```
var json = window.localStorage['Comic Books'],
    comics = (json != null) ? JSON.parse(json) : [];

comics.forEach((comic_book) {
  if (comic_book['title'] == 'Sandmn') {
    comic_book['title'] = 'Sandman';
  }
});
window.localStorage['Comic Books'] = JSON.stringify(comics);
```

类似地, 删除就是通过在本地的副本中删除数据, 然后再序列化后存储到 localStorage 的数据项中来完成的。

```
var json = window.localStorage['Comic Books'],
    comics = (json != null) ? JSON.parse(json) : [];

var awesome_comics = comics.filter((comic_book) {
  return (comic_book['id'] >= 42);
});
window.localStorage['Comic Books'] = JSON.stringify(awesome_comics);
```

这并不比 localStorage 更容易。遗憾的是, 每一个操作都会阻塞浏览器做其他事情。所以, 如果应用有太多的数据在浏览器上, 那么就要考虑一些更好的做法。

要点: 写作本书时, IndexedDB 和 Web SQL 都还没有做好常规使用的准备。本书以后会更新相关内容。

15.3 WebSocket

WebSocket 是一个很棒的新技术, 它允许我们在浏览器和服务器之间进行真正的异步通信。Web 开发者不需要再被迫使用像 comet 或 Ajax 长轮询这样的特殊技巧。浏览器现在可以打开一个到服务器的 WebSocket, 并通过这个连接按需要发送数据。更好的是, 当服务器有新信息的时候, 它可以通过同一个 WebSocket 立即推送数据给用户。

Dart 对 WebSocket 的支持非常不错。例如，在我们的漫画书应用中，使用 WebSocket 代替 Ajax 来实现交换数据的同步层非常简单。

```
import 'dart:html';
import 'HipsterSync.dart';

// ws 在库范围内可见，所以 main() 和 wsSync()
// 将访问的是同一个 WebSocket
WebSocket ws;

main() {
  HipsterSync.sync = wsSync;
  ws = new WebSocket("ws://localhost:3000/");

  // 直到 websocket 打开后才能获取数据，
  // 以便 wsSync 使用已激活的 WebSocket 通信
  ws.
    on.
    open.
    add((_) {
      var my_comics_collection = new Collections.Comics();
      my_comics_collection.fetch();
      // 其他初始化...
    });
}
```

我们以一个适当的 `ws://` 的 WebSocket URL 为参数，通过 WebSocket 的构造函数创建了一个 WebSocket 对象。WebSocket 是完全异步的，包括打开连接。因此，我们添加了一个 WebSocket 的 `open` 事件的监听器。当连接打开时，我们就可以通过 WebSocket 开始执行数据同步操作，如获取数据。

注意：在编写本书时，Dart 不支持声明 WebSocket 的子协议。

通过 WebSocket 发送消息非常简单——我们只需调用 `ws.send(message)` 方法。回想一下，在 HipsterSync 中数据同步方法需要接收两个参数：增删改查的方法和要同步的模型（或集合）。使用这些信息，我们可以构造出要通过 WebSocket 发送到后端的消息。

```
wsSync(method, model) {
  String message = "$method: ${model.url}";
  if (method == 'delete')
    message = "$method: ${model.id}";
  if (method == 'create')
    message = "$method: ${JSON.stringify(model.attributes)}";

  ws.send(message);
}
```


这段代码可以发送消息，但是我们还需要从服务器收到的响应，并依次通知响应处理的剩余部分。正如在第 13 章所见，我们用一个 `Future` 通知 `HipsterSync` 类已经完成。所以，我们的 `wsSync` 层需要它自己的 `Completer` 对象，而且一旦从服务器收到响应它就需要结束处理。

```
wsSync(method, model) {  
  final completer = new Completer();  
  String message = /* 前文的消息构建 */  
  
  ws.send(message);  
  ws.  
    on.  
      message.  
        add(_wsHandler(event) {  
          completer.  
            complete(JSON.parse(event.data));  
          event.target.on.message.remove(_wsHandler);  
        });  
  
  return completer.future;  
}
```

我们的同步函数的返回值是一个 `Future`。`HipsterSync` 期望得到它，并且一旦 `completer` 被标记为结束，就用相应的 `then()` 语句传播这个信息到 MVC 栈中。在 `message` 处理程序中会收到服务器的响应，我们用来自服务器的消息完成这个 `Future`，消息是在消息事件的 `data` 属性中的。

在这里，我们只想要监听来自服务器的一个响应，所以我们在 `completer` 结束后删除了这个处理程序。如果这个消息处理程序被留在原处，它将会继续接收来自服务器后续的响应消息（例如，用户发起新建的响应）。因为在这个同步闭包中引用的 `completer` 已经完成了，所以应用将产生各种关于这个 `completer` 已完成的消息。

Dart 中的 `WebSocket` 与 JavaScript 的 `WebSocket` 很相似，既不更难也不更容易。然而 Dart 中使用了独特的 Dart 方式，这样用起来更轻松。

15.4 Canvas

Dart 语言缺少像 `Raphaël.js`^① 这样的能够减轻与使用 `<canvas>` 元素相关的痛苦的库。即使如此，它还是带来了实现 HTML5 游戏的基础功能。

① <http://dmitrybaranovskiy.github.io/raphael/>。

正如传统的画布一样，Dart 语言也需要一个<canvas>元素和相关的绘图上下文。如果页面已经有了一个<canvas>元素，那么我们通过 `getContext()` 方法获得一个绘图上下文。

```
CanvasElement canvas = document.query('canvas');
CanvasRenderingContext2D context = canvas.getContext('2d');
```

使用这个上下文，我们就可以画各种各样的奇妙的东西。通过下面的示例，我们可以在整个画布上画出一个空的白色的矩形作为背景。

```
int width = context.canvas.width,
    height = context.canvas.height;

// 开始绘图
context.beginPath();

// 清空绘图区域
context.clearRect(0,0,width,height);
context.fillStyle = 'white';
context.fillRect(0,0,width,height);

// 结束绘图
context.closePath();
```

一个普通的白色背景缺乏趣味性。要使其增加点儿趣味，我们可以添加一个简单的红色正方形，用来表示我们在一个游戏空间中的当前位置。如果我们的当前位置是由一个包含 `x`、`y` 坐标的 `Player` 对象封装的，那么我们的初始布局看起来可能像下面这样：

```
// 开始绘图
context.beginPath();

// 清空绘图区域
// ...
// 绘制我的当前位置
context.rect(me.x, me.y, 20, 20);
context.fillStyle = 'red';
context.fill();
context.strokeStyle = 'black';
context.stroke();

// 结束绘图
context.closePath();
```

这会在我的位置上画一个 20 像素大小的正方形，红色的内里和黑色的边框。此时它还需要的是一个处理方向键动作的监听器。当方向键按下时，事件能够通过 `move()`

方法让玩家在适当的方向上移动，并重画整个画布。

```
document.  
  on.  
    keyDown.  
      add((event) {  
        String direction;  
  
        // Listen for arrow keys  
        if (event.keyCode == 37) direction = 'left';  
        if (event.keyCode == 38) direction = 'up';  
        if (event.keyCode == 39) direction = 'right';  
        if (event.keyCode == 40) direction = 'down';  
        if (direction != null) {  
          event.preventDefault();  
          me.move(direction);  
          draw(me, context);  
        }  
      });
```

这里，`draw()` 函数执行和前文同样的 `context` 操作，只是用更新后的位置。

随着 Dart 语言的演进，无疑还将有更多 API 的改进，使它更容易使用。更重要的是，应该会出现许多构建于其上的库可用。现在已经有有了一个 Box2D 库的 Dart 语言的早期移植版本，它可以画一些简单的物理图形。它被命名为 DartBox2D^①，它非常值得一看。

15.5 小结

这些是当前 Dart 开发者可用的各种混杂的工具。但实际上，这就是 HTML5 的样子——在较新的浏览器中可用的各种混杂的技术。它们是被 Dart 语言处理得很到位的技术。这些技术在 Dart 中可能还会进一步改进。

Dart 语言还在演化。引人注目的是，有将近 100 人在全职专注于改进 Dart 语言。更重要的是，围绕 Dart 语言兴起的社区非常活跃。所以，请加入我们 <https://groups.google.com/a/dartlang.org/group/misc/topics>，在社区中发出你的声音！

^① <http://code.google.com/p/dartbox2d/>。

欢迎来到异步社区！

异步社区的来历

异步社区 (www.epubit.com.cn) 是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 = 1 元，购买图书时，在 使用积分 里填入可使用的积分数值，即可扣减相应金额。



特别优惠

购买本书的读者专享异步社区购书优惠券。

使用方法：注册成为社区用户，在下单购书时输入 **57AWG** 使用优惠码，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信服务号



微信订阅号



官方微博



QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn

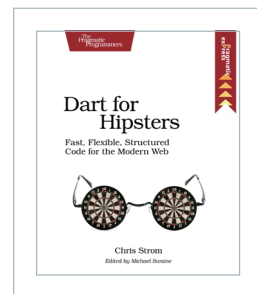
异步社区Kensuke(18319066421) 专享 请尊重版权



Dart语言程序设计

Dart for Hipsters

Fast, Flexible, Structured Code for the Modern Web



2011年10月10日, Google正式推出Dart, 目标是为开发者提供一种更适合现代Web应用的开发语言。

Dart到底是什么?

- ◆ Dart是一款基于类的面向对象编程语言。
- ◆ Dart适合构建结构化的现代Web应用。
- ◆ Dart支持个人项目, 更支持架构复杂的大型项目。
- ◆ Dart代码可以使用原生虚拟机引擎和编译为JavaScript两种方式执行。

本书是第一本关于Dart语言的中文书籍, 介绍了当前Dart语言的最新内容。书中涵盖了Dart语言基础、并发编程、Web编程和HTML5应用等方面的内容, 同时涉及了函数式和面向对象编程。本书使用真实的项目进行讲解, 以帮助读者更好地理解Dart并解决问题。

本书适合编程语言爱好者和Web开发人员阅读。

Chris Strom 作家、程序员、演讲者。他认为自己是一个不懈努力和热爱分享的学习者, 并写了1000多篇博客文章作为写书时的研究笔记, 他的个人主页是<http://japhr.blogspot.com/>。他有超过10年的Perl、Ruby、JavaScript编程经验。除本书之外, 他还著有*The SPDY Book*和*Gaming JavaScript for Girls and Boys*两本书, 并且与人合著了*Recipes with Backbone*一书。

韩国恺 程序员/开发经理, 8年软件开发实践经验, 3年团队管理经验。擅长Web和后端技术, 熟悉多种编程语言, 关注互联网行业。在创业公司负责解决架构、开发和运维等方面的所有技术问题。

The
Pragmatic
Programmers



ISBN 978-7-115-29694-8



9 787115 296948 >

分类建议: 计算机 / 程序设计

人民邮电出版社网址: www.ptpress.com.cn

封面设计: 任文杰

异步社区Kensuke(18319066421) 专享 请尊重版权