The Pragmatic Programmers

图灵编程语言集萃

Dart for Hipsters

Fast, Flexible, Structured Code for the Modern Web

Dart语言程序设计

【美】Chris Strom 著

韩国亮 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

韩国信 译

人民邮电出版社

POSTS & TELECOM PRESS

※版权信息※

书名：Dart语言程序设计
作者：[美]Chris Strom
排版：吱吱
出版社：人民邮电出版社
出版时间：2013-01-01
ISBN：9787115296948

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；既可以用在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]。

【注】① 翻译此书时为0.10版。—译者注

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。

【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

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。

【注】③ http://www.chromium.org/spdy。

【注】④ http://spdybook.com/。

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。

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，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。

【注】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。

【注】② http://api.dartlang.org。

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()。

【注】③ 一种以下划线连结单词的命名方式。—译者注

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！

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。

【注】① http://www.dartlang.org/dartium/。

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。

【注】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

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.findByElementId()来查找元素，并且用#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。

【注】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

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}
]
```

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HMTL字符串①。

【注】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

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() {

   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 。

【注】① http://www.dartlang.org/dartium/。

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。

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帮助的。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。

【注】① 在Dart中，int和double都继承自num。—译者注

```
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。

【注】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

```
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"
```

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。

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

```
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);

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

【注】① 当前List已支持map和reduce方法。——译者注

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。

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]。

【注】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

2.6 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。

【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

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
```

【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

```
// 2012-03-01 14:31:12.000

var now = new Date.now();

// 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
```

【注】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

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②下完成 Dart 代码的编写。默认情况下，Dart 运行在“生产”模式下③，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。

【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

【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

【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

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。

```
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。

【注】① Dart是动态类型语言，也是强类型语言，类似于Ruby。—译者注

```
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。

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。

【注】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。

```
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。

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_printer②。

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

【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

```
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。

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

【注】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

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。

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。

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。

【注】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

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。

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。

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

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的列表。下面是一些简单的示例：

【注】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。

【注】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

```
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。

【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

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：

```
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。

【注】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

```
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><img src="img01.png"/>
    <li><img src="img02.png"/>
    <li><img src="img03.png"/>
    <!-- ... -->
  </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：

```
gallery(title, photographer) {
   return new Element.html("""
     <div id="gallery">
      <h2>${title}</h2>
      <ul>
         <li><img src="img01.png"/>
         <li><img src="img02.png"/>
         <li><img src="img03.png"/>
         <!-- ... -->
      </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。

【注】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

```
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。

【注】② https://developer.mozilla.org/en/DOM/Element.insertAdjacentHTML。

```
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 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。

```
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：

【注】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

```
<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]。

【注】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

【注】② 当前Dart语言已支持更加强大的链式调用语法，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

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】③ http://api.dartlang.org/html/Element.html。

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。

【注】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

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

【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

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。

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。

【注】① http://coffeescript.org/。

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 。

【注】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

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 JavaScrip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。

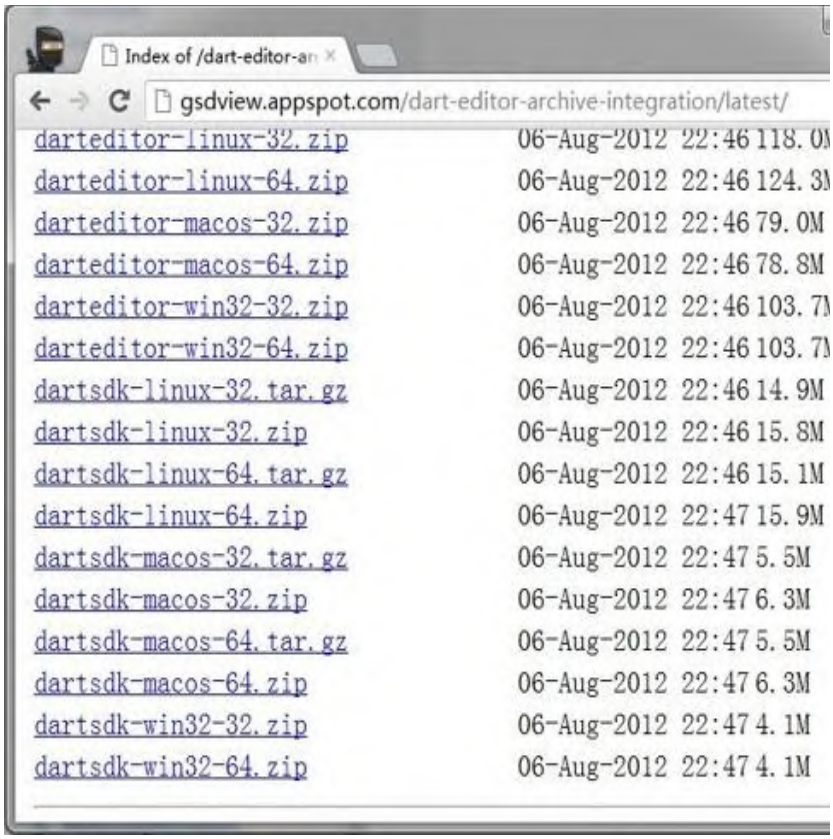
以后只会更好。

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。SDK②是下图中文件名以"dartsdk"开头的那些。

【注】① http://gsdview.appspot.com/dart-editor-archive-continuous/latest/。

【注】② SDK中包含了dart vm、dart2js、pub、dartdoc等。—译者注



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
|   +-- 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。

【注】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

```
$ 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。

```
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Dar t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）。

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

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。

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。

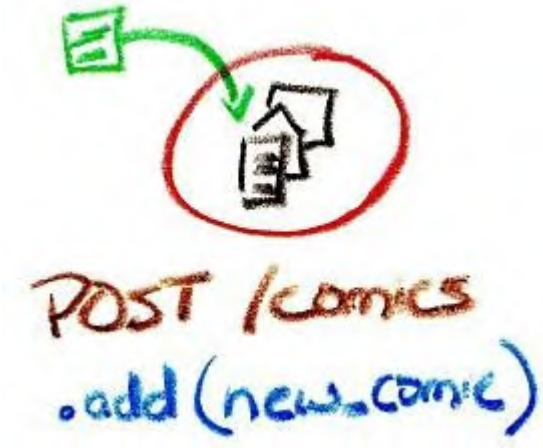
6.1 Dart中的MVC

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。

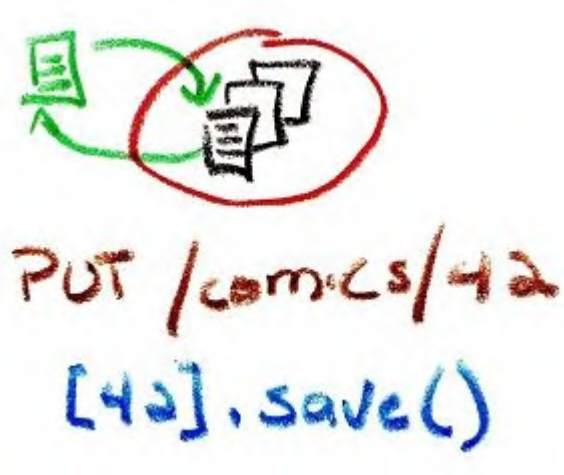
回忆一下第 1章，我们的漫画书集合可以通过对/comics上的HTTP GET方法来获取。在这个MVC中，我们称它为fetch()方法。

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]。

【注】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

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modles。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。

【注】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

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（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()。

【注】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

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。

【注】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

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']}";

  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（<li>）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(el, 'click .delete', delete)这条语句的意思是，在 DOM 元素 el 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el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】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

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。

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。

【注】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

```
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。

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
```

【注】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

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：

【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 {
   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。

```
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]。

【注】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

```
class Planet {
  static List rocky_planets = const [②
```

【注】② const是用于修饰常量值的。—译者注

```
      '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 = [];
      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。

【注】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

```
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。

【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

```
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。

【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

```
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。

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。

【注】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

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。

【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

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。

【注】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

```
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。这有效地消除了在JavaScript中困扰对象初始化的复杂条件。

【注】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

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。

【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-contructor-and-top.htm/。—译者注

为了避免重复自身，我们使用重定向构造函数。

```
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ComicsColl-ection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。

【注】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

```
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。

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：

【注】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

```
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 接口只不过是要求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。

【注】② 其实就是要求实现提供一个Events类型的on字段。—译者注

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
- 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 {
  CollectionEventList load_listeners, insert_listeners;
  CollectionEvents() {
    load_listeners = new CollectionEventList();
    insert_listeners = new CollectionEventList();
  }
  CollectionEventList get load => load_listeners;
  CollectionEventList get insert => insert_listeners;
}
```

当创建 CollectionEvents 类的新实例时，构造函数就会构建两个CollectionEventList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CollectionEventList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 CollectionEventList implements EventListenerList {
  List listeners;
  CollectionEventList() {
```

```
    listeners = [];
  }
  CollectionEventList add(fn) {
    listeners.add(fn);
    return this;
  }
  bool dispatch(CollectionEvent event) {
    listeners.forEach((fn) {fn(event);});
    return true;
  }
}
```

在CollectionEventList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 CollectionEventList 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大文件中。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的问题。

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]。

【注】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

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。

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）。

【注】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

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。

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]。

【注】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

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、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。

【注】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

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);

}
```

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。

【注】① http://www.commonjs.org/。

【注】② https://github.com/amdjs/amdjs-api/wiki/AMD。

【注】③ http://wiki.ecmascript.org/doku.php?id=harmony:modules。

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) {
```

```
        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。

【注】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

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。

【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

因此，像下面这样就不行：

```
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。

【注】① http://japhr.blogspot.com/2012/01/dart-mixins.html。

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");
```

```
  }
}
```

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Dar t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。

【注】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

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。

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。

【注】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get er/set er。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

```
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。

【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

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。

```
// *** 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]。

【注】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。

【注】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

这样，我们的动态save()方法可以像下面这样实现②：

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

```
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。

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。

【注】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

【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

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:uniffest/unittest.dart';
```

这样，我们就为测试做好了准备 ③ 。

【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/dartest/dartest.dart';
import "../public/scripts/HipsterCollection.dart";
main() {
  // 在这里编写测试!
   new DARTest().run();
}
```

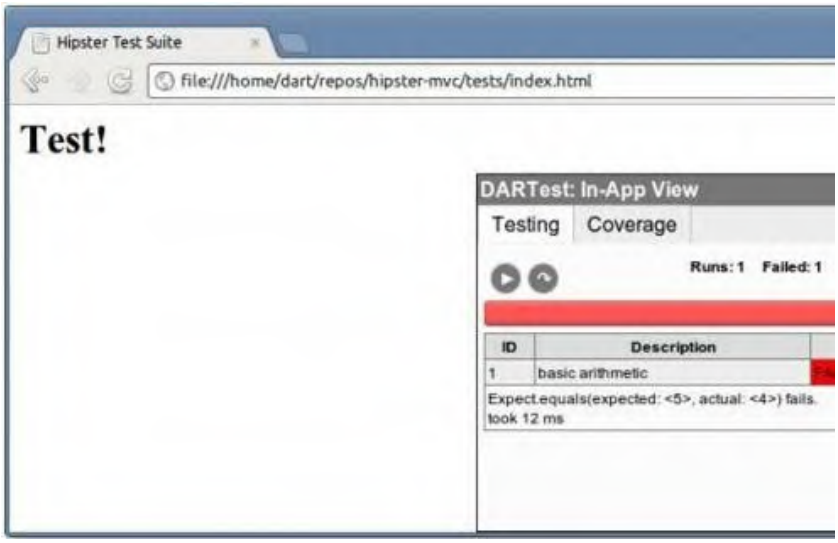
除了导入以外，我们创建了一个DARTest实例，并且用run()方法启动测试集[①]。然而，没有测试代码之前什么也不会发生，所以下面我们写一个测试。在行为驱动开发

（Behavior-Driven Development）的惯例中，都是以一个失败的测试开始的。

【注】① 本章所用的dartest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

```
testing/tests/02test.dart

import 'package:unittest/unittest.dart';

import 'lib/dartest/dartest.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。



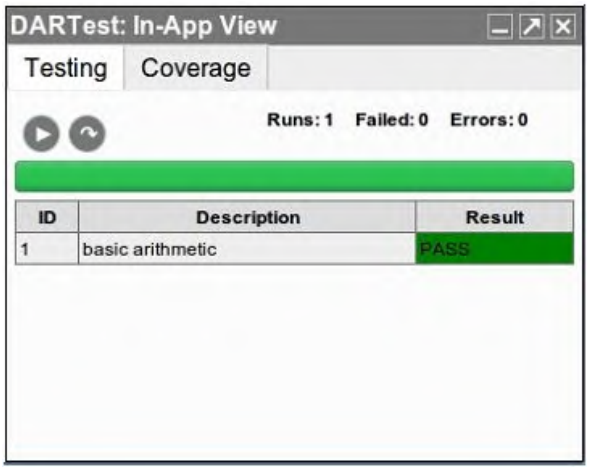
如果以（非Web的）默认形式运行，会在命令行上产生如下输出：

```
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));

});
```

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。例如，我们可以验证，如果没有URL，HipsterCollection#fetch()方法会失败。

【注】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

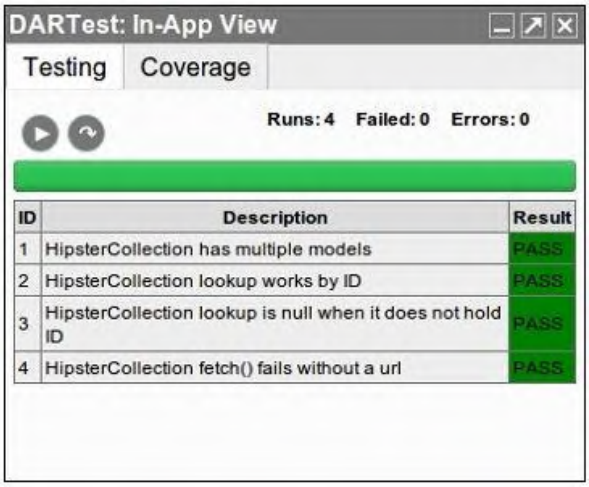
```
  test('HipsterCollection fetch() fails without a url', () {

    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。

【注】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

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});

});
```

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。

【注】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

```
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) {
      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。

【注】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

```
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。

【注】① http://en.wikipedia.org/wiki/Doomsday_rule。

```
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。

【注】① https://developer.mozilla.org/en/CSS/-moz-transition。

• 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—并是Dart语言当前唯一支持的[②]。

【注】② 现在也支持IndexedDB。——译者注

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。

```
// 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。

【注】① http://raphaeljs.com/。

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。

【注】① http://code.google.com/p/dartbox2d/。

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！