Broadview
www.broadview.com.cn

[PACKT] open source
PUBLISHING
community experience distilled

Learning TypeScript

Learning TypeScript中文版

使用TypeScript的特性来更轻松地开发和维护Web应用

[西班牙] Remo H. Jansen 著
龙逸楠 蔡伟 迷走 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

TypeScript 是一个开源的、跨平台且带有类型系统的 JavaScript 超集，它可以编译为纯 JavaScript，然后运行在任意的浏览器和其他环境中。它使开发者可以使用一些未来 JavaScript 标准（ECMAScript 6 和 7）中的特性。TypeScript 为 JavaScript 添加了可选的静态类型、类和模块，让大型 JavaScript 应用可以使用更好的工具并拥有更清晰的结构。

本书通过大量示例，一步一步让读者由浅至深地了解 TypeScript。首先介绍了 TypeScript 的基本概念，接着介绍了像 Gulp 这样的自动化工具，以及对函数、泛型、回调和 promise 的详细解释。还介绍了 TypeScript 中的面向对象特性和内存管理能力。最后，带领读者使用本书讲解的概念编写了一个单页面应用。

本书读者对象

如果你是一个想要学习TypeScript来构建大型Web应用的人，那么本书正适合你。在阅读本书之前，你并不需要拥有任何TypeScript的相关知识。

你将从本书学到：

- 学习 TypeScript 语言的关键特性和运行时
- 开发模块化、可伸缩、可维护以及可适配的 Web 应用
- 编写符合 SOLID 原则的面向对象的代码
- 使用像 Gulp 和 Karma 这样的自动化工具来节省时间
- 使用 Mocha、Chai 和 SinonJS 这样的测试工具来开发健壮的应用
- 通过实践从零开发一个单页面应用框架，以巩固你的 TypeScript 知识
- 通过 TypeScript 使用未来 JavaScript 标准（ES6 和 ES7）中的特性

[PACKT] open source* PUBLISHING community experience distilled

上架建议：Web开发

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：刘　舫
封面设计：吴海燕

ISBN 978-7-121-30047-9
9 787121 300479 >
定价：89.00元

Learning TypeScript

Learning TypeScript中文版

[西班牙] Remo H. Jansen 著
龙逸楠 蔡伟 迷走 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书首先介绍了 TypeScript 的基本语法和基本的自动化工作流配置方法，然后从面向对象入手，着重介绍了面向对象的概念和它的一些最佳实践，并结合例子讲解了如何基于 TypeScript 的类型系统应用这些最佳实践。随后剖析了 TypeScript 在编译后的运行时行为，并从性能与测试的角度讲解了如何编写健壮的 TypeScript 代码，所以书中还包括了性能分析与测试相关的内容。最后介绍了如何使用 TypeScript 结合面向对象、MVC 等概念，并配合本书前面提到的自动化的工作流、面向对象最佳实践、性能优化和测试等内容实现一个单页应用（SPA）框架，并用这个框架构建了一个单页应用。

本书适合使用 TypeScript 来构建大型 Web 应用的开发人员。

版权贸易合同登记号 图字：01-2016-1972

图书在版编目（CIP）数据

Learning TypeScript：中文版/（西）雷莫·H.詹森（Remo H. Jansen）著；龙逸楠，蔡伟，迷走译.—北京：电子工业出版社，2016.11

书名原文: Learning TypeScript

ISBN 978-7-121-30047-9

Ⅰ. ①L… Ⅱ. ①雷… ②龙… ③蔡… ④迷… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2016）第 245530 号

策划编辑：张春雨
责任编辑：刘 舫
印 刷：北京天宇星印刷厂
装 订：北京天宇星印刷厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787 × 980 1/16 印张：21.5 字数：443 千字
版 次：2016 年 11 月第 1 版
印 次：2016 年 11 月第 1 次印刷
定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

译者序

在 JavaScript 社区的荒蛮时代里，构建大型 Web 应用是一件吃力且烦琐的事情。但迫于业务发展的需要，业界一直在探索如何像成熟的工业化的语言那样开发和构建大型的应用。

微软在 2009 年发布了 TypeScript 的第一个版本，它为 JavaScript 带来了类型系统与模块系统（现在已经废弃并鼓励使用 ES6 模块）。而自从 TypeScript 问世，JavaScript 社区就没有停止过对它的议论，有人认为类型系统给 JavaScript 带来的静态检查能力更有利于构建大型应用，而另一些人则认为类型系统会使 JavaScript 丧失其先天的灵活性和动态性，不利于提高开发效率。还有一些人担心 TypeScript 为 JavaScript 带来了太多非标准的特性，很难保证 TypeScript 在未来与 JavaScript 在语言层面上保持高度统一。

译者在翻译本书前，刚刚经历了将一个 CoffeeScript 应用全面使用 TypeScript 重构。CoffeeScript 从某种程度上来看完全是 TypeScript 的对立面，它甚至比 JavaScript 更具动态性与灵活性。而在译者重构的这个项目中，代码量高达上万行，且包含数十个模块，之所以选择使用 TypeScript 重构它，是因为日益增长的代码量与模块数使得团队协同开发的难度越来越大。

也正是因为这次重构，使译者能深入了解 TypeScript。它的类型系统异常强大，可以大大降低团队中成员的沟通成本。以往，团队中的成员需要调用另一个成员写的模块时，必须要仔细研究这个模块中各个 API 的参数、参数类型和返回值。然而因为项目中通常缺乏文档与注释（这是另外的讨论点），所以通常在使用其他人开发的模块时会占用程序员大量的时间去阅读和理解其他人的代码。而 TypeScript 的类型系统无疑是给了我们一个快捷的文档，使得我们能更好更快地使用别人的模块。即使是在拥有良好的注释以及文档的模

块中，TypeScript 服务提供的 Intellisense 功能也能让开发人员如虎添翼，再也不用担心忘记参数类型或忘记方法名。

而另一方面，TypeScript 过强的约束也在开发时给我们带来了很多额外的困扰。比如在开启了--noImplicitReturns 参数后，一些设计成无返回值的代码将会导致编译失败，比如：

```
getOne(_id: string): string {
  const result = cacheService.exist(_id)
  if (result) {
    return cacheService.findById(_id)
  }
}
```

由于这段代码只有一个分支有返回值，所以它将导致编译失败，但我们正是期待代码这样运行。

同时，我们也接触到了一些 TypeScript 的私有功能，比如 enum（枚举）与 reflectMetaData（元数据反射）。出于对使用非标准特性的风险的考量，我们并没有在项目中使用这些特性。

除了强大的类型系统带来的可靠的静态检查以外，我们还惊喜地发现了一个能大大增加项目可维护性与健壮性的实践，那就是依靠 TypeScript 的类与接口将面向对象的 SOLID 原则应用到项目中，这无疑是搭了类似 Java 与 C# 等面向对象语言的顺风车，它能让我们更容易地写出高内聚低耦合的代码。

所以在面对社区各种对 TypeScript 褒贬不一的评价时，希望读者能理性地看待这些声音，并且能够在深思熟虑之后进一步接触 TypeScript，深入了解它的优点与缺点，最终为自己的项目选择合适的工具。

关于作者

Remo H. Jansen 是一位前端工程师、开源项目贡献者、企业家、科技爱好者、游戏爱好者和互联网爱好者。

他来自西班牙的塞维利亚，但目前居住于爱尔兰的都柏林，并在那里做着一份金融服务行业的全职工作。Remo 有着多年的大型 JavaScript 应用开发经验，从航班预定系统到投资组合管理解决方案。

Remo 在 TypeScript 社区中十分活跃。他是都柏林 TypeScript 交流会的组织者，并且是 InversifyJS（一个 TypeScript 应用的控制反转容器）和 AtSpy（一个 TypeScript 应用的测试框架）的作者。在他的个人博客（`http://blog.wolksoftware.com/`）中，他写了许多关于 TypeScript 和其他 Web 技术的博客。

Remo 也是由 Packt Publishing 出版，Nathan Rozentals 撰写的 *Mastering TypeScript* 一书的技术审校者。

若想要与他取得联系，可以访问 `http://www.remojansen.com/`。

致谢

这是我出版的第一本书。在此之前，我经历了相当漫长的学习之路，并且从许多值得感谢的人那里，学到了许多知识。

我首先要感谢地处特里亚纳圣彼得慈幼会（位于西班牙的塞维利亚）的计算机科学学院中的老师们，因为他们让我体会到了教育的价值。

感谢 Packt Publishing 团队的成员们的支持和努力工作，与你们的合作十分愉快。

感谢本书的所有技术审校者，他们无价的反馈和努力工作显著地提升了本书内容的质量。

感谢我的同事和室友，Sergio Pacheco Jimenez 和 Adolfo Blanco Diez，因为我常在半夜与前者进行漫长的技术交流，后者为我提供了大量咖啡饮料的支持。

感谢我的女朋友，Lorraine，我为你无条件的支持和耐心深感荣幸。你就是世界上最好的女朋友，并且还在不断变得更好。

最后，感谢我的家人，感谢你们相信我，为我提供指导，成为我最好的倾听者，支持我的工作，原谅我所犯的错误，以及其他一切你们教会我的东西。感谢我们在一起所经历的开心与伤心的时光。能够成为你们的孙子、儿子和兄弟，让我十分骄傲。

关于审校者

Liviu Ignat 是一位全栈工程师、架构师、科技极客和企业家，从 2004 年以来开始编写商业软件。一开始使用的是 VB6，接着开始使用.NET 和 Java，后来转向了 Web 前端开发。他对函数式语言十分感兴趣，如 F#、Scala、Swift、JavaScript 等。在他的一些最新的服务端 Node.js 项目和大多数使用流行前端框架的项目中已经使用上了 TypeScript。

目前，他正致力于许多项目，大多数项目是 `http://giftdoodle.com/`中的，他是这个公司的 CTO，这家公司中大多数 JavaScript 项目都使用 TypeScript 编写。在他的工作经历中，他使用.NET 编写过分布式后端服务，也编写过复杂的单页 Web 应用。最近他正致力于使用 Node.js 和 Docker 编写微服务，编写单页 Web 应用，以编写 Android 和 iOS 原生应用。

当 Liviu 不写代码时，他喜欢在冬天滑雪，在夏天坐帆船去国外，去世界的其他地方旅行。你可以在 `http://www.ignat.email/`联系到 Liviu。

Jakub Jedryszek 目前是微软的一位软件工程师。在审校本书时，他工作于 Azure Portal，这是世界上使用 TypeScript 编写的最复杂的单页 Web 应用。他也是.NET 开发者的 dotNetConfPL——online 会议的共同发起人之一。他的博客是 `http://jj09.net/`。

Andrew Leith Macrae 最初在 Apple 的产品上开始了他的编程生涯。多年以来，他使用过 Hypercard、Director、Flash 和最近的 Adobe AIR for mobile 开发交互式应用。并且在开发过程中，他也会用到 HTML。他目前是多伦多 Rogers Communications 公司的高级前端工程师，正在使用 AngularJS 和 SASS 进行敏捷开发。

他坚信 TypeScript 是 JavaScript 的未来，TypeScript 带来了强类型面向对象语言中的结构化和规则，为开发大规模 Web 应用的代码编写提供了语义上的便利。

你可以在 `http://adventmedia.net/` 联系到 Andrew。

Brandon Mills 的编程生涯已经有十多年了，他就职过只有两个人的初创公司，也在微软就职过。他在微软参与了 Visual Studio 2013、Azure Tools 和预装于 Windows 10 的 Edge 浏览器项目。他也是 ESLint 项目的核心开发团队成员之一，这是一个开源可配置的 JavaScript 和 JSX 的语法检查工具。他目前在 Node.js 平台上使用 JavaScript 或 TypeScript 编写应用和服务。他的 GitHub 是 `https://github.com/btmills`。

感谢 Scott 的激励和给予我的灵感，感谢 Linda 给予我无条件的爱，感谢 Abby 给予我的耐心，以及感谢 Ashlynn 对我的支持。

Ivo Gabe de Wolff 是一位 ivogabe（创始于 2012 年）名下的自由职业开发者，他正在乌得勒支大学学习数学和计算机科学。当他只有 11 岁时，便开始使用 Game Maker 编写游戏程序。在学了诸如 C# 和 JavaScript 等诸多编程语言后，他在现在的大多数项目中使用 TypeScript。在过去的几年里，他在许多不同的环境中使用过 TypeScript，如移动端应用。目前他主要致力于 Node.js 程序的开发。

另外，他也是许多开源项目的作者，包括 gulp-typescript。你可以在 `https://github.com/ivogabe` 中找到他的项目。如果你想阅读更多有关 TypeScript、JavaScript、Gulp 或数学的内容，也可以看看他的博客 `http://dev.ivogabe.com/`。

前言

在过去的几年里，基于 JavaScript 的 Web 应用的数量呈几何级数进行增长。但是，目前的 JavaScript 标准（ECMAScript 5，又称 ES5）是在许多年前设计出来的，面对如今大规模 JavaScript 应用的复杂性时，它缺少了许多必要的特性。正是由于这些特性的缺失，一些应用的可维护性问题暴露了出来。

新一代的 JavaScript 标准（ECMAScript 6，又称 ES6），旨在解决上述可维护性问题。但它还没有完全实现，且现在我们使用的浏览器与之也不是完全兼容的。所以，ES6 标准的广泛采用，仍被认为是一个漫长的过程。

为了解决这类 JavaScript 的可维护和可扩展性问题，微软花了两年时间开发出 TypeScript，并在 2012 年 10 月，公开发布了它：

> “我们为需要构建和维护大型 JavaScript 程序的团队设计了 TypeScript，以满足他们的需求。TypeScript 可帮助他们在软件组件之间定义接口，并且帮助理解现存 JavaScript 库的行为。同时，使用 TypeScript 的团队可以将代码构建成动态加载的模块，以减少命名冲突的问题。TypeScript 可选的类型系统使开发者们可以使用一些高效的开发工具和最佳实践：静态检查，基于符号的导航，代码补全和代码重构。”
>
> ——TypeScript 语言特性 1.0

即使是具有丰富经验的开发者，也很难给出大规模 JavaScript 应用明确的定义。当谈及这个话题时，我们应该避免使用代码行数来作为评判标准，而是应该以代码中模块的数量和模块之间的依赖关系来评判应用的规模。我们将大规模应用定义为，需要众多开发者一同维护且具有一定复杂度的程序。

本书将会运用简单且易于理解的形式来介绍 TypeScript 众多的特性。当你读完本书时，你应该知晓用 TypeScript 构建大型 JavaScript 应用所需的所有知识。本书不但提供了 TypeScript 核心特性的介绍，还会带领你探索一些有用的工具、代码设计规则、最佳实践以及如何将它们用于生产环境中。

本书涉及的内容

第 1 章介绍了 TypeScript 的核心特性，包括可选的静态类型提醒系统、操作符、函数、接口和模块。同时还附有在实际环境中使用它们的例子。

第 2 章介绍了一些自动化工具，如 Gulp 和 Karma，用以最大化开发者的生产力。另外，本章还介绍了一些在开发 TypeScript 应用时，有助于你使用第三方库的工具。

第 3 章深入探讨了 TypeScript 中的函数。为了成为一个精通 TypeScript 的开发者，本章还会告诉你关于异步编程需要知晓的一些知识。

第 4 章深入探讨 TypeScript 中的面向对象编程，包含类、接口和模块，并且推荐了一些最佳实践（SOLID 原则[1]）。还会包括如继承、混入和泛型，它们都有助于增强代码的可复用性。

第 5 章帮助你理解 TypeScript 运行时的工作机制，将有助于避免一些潜在的性能问题，使得我们成为更高效的 TypeScript 开发者。

第 6 章讲解了高效运用可用的系统资源的必要知识。这一章还会阐述测试 TypeScript 应用性能的方法，以及如何自动化一些用于提升 TypeScript 应用性能的任务。

第 7 章介绍了如何使用 TypeScript 测试工具来进行 BDD（行为驱动开发）测试。在这一章，你将学习到如何使用 Karma、Mocha、Chai 和 Sinon.JS 来编写 TypeScript 单元测试，如何使用 Nightwatch.js 编写端对端测试，以及如何使用 Istanbul 来生成测试覆盖率报告。

第 8 章深入探讨了装饰器，包括类、属性、参数和方法装饰器。该章还会包括关于反射元数据 API 的介绍。

第 9 章介绍了一些现代 Web 应用的核心架构原则。该章会介绍单页面 Web 应用的概念，以及它的通用组件和特性（模型、视图、控制器、路由和模板等）。该章还会通过实现一个单页面 Web 应用框架，来阐述你所需知道的一切。

[1] SOLID 原则：面向对象编程和面向对象设计中的 5 个基本原则，即单一功能、开闭原则、里氏替换、接口隔离以及依赖反转。

第 10 章通过使用 TypeScript 以及本书其他章节所提到的概念，来实现一个单页面 Web 应用。

阅读本书前需要做的准备

本书中的例子都是使用 TypeScript 1.5 编写的，你需要 TypeScript 编译器和一个文本编辑器。本书将会说明如何使用 Atom，但是你也可以使用其他的编辑器，如 Visual Studio 2015、Visual Studio Code 或 Sublime Text。

你还需要一个能够上网的环境来下载一些必要的依赖引用、包和库，如 jQuery、Mocha 和 Gulp。在一些操作系统下安装本书中的一些工具，你可能还需要一个拥有管理员权限的账号。

本书的读者对象

如果你是一个想要学习 TypeScript 来编写漂亮的 Web 应用的中等水平的 JavaScript 开发者，那么本书正适合你。你只需对 jQuery 的基本概念有所了解。

为了让你能最大化地运用 TypeScript 语言和其编译器，本书将会由浅入深地介绍 TypeScript 的语言结构和面向对象的特性。本书还将展示，如何使用强类型、面向对象的原则、设计模式和一些最佳实践来轻松管理复杂的大规模 JavaScript 应用。

约定惯例

本书将会使用不同的书写风格来区分不同种类的信息。以下是这些风格的例子和它们的意义。

正文中的代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、URL、用户输入和推特用户定位将会用代码体书写，如“我们可以通过 `include` 指令将其他上下文包含进来”。

代码块则会是下面这样的风格：

```
class Greeter {
    greeting: string;
```

```
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

如果希望向你强调代码块中的一部分，那么它们将会以粗体展示：

```
function MathHelper() { /* ... */ }

// class method
MathHelper.areaOfCircle = function(radius) {
  return radius * radius * this.PI;
}

// class property
MathHelper.PI = 3.14159265359;
```

任何命令行的输入和输出将是以下这样的：

```
git clone https://github.com/user-name/repository-name.git
```

警告和关键提醒将会在这样的图标后出现。

小提示和小技巧将会在这样的图标后出现。

下载示例代码

你可以从 http://www.broadview.com.cn 下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

目录

1

TypeScript 简介

本书将集中讲解 TypeScript 与生俱来的面向对象特性以及这种特性怎么帮助你写出更优雅的代码。在我们深入介绍 TypeScript 的面向对象特性之前，本章将会带你概览 TypeScript 背后的历史，除此之外还会介绍一些 TypeScript 的基础知识。

在本章，你将了解以下概念：

- TypeScript 的架构
- 类型标注
- 变量和原生数据类型
- 运算符
- 流程控制语句
- 函数
- 类
- 接口
- 模块

TypeScript 的架构

在这一节中，我们将集中了解 TypeScript 的内部结构和最初设计它时的目标。

设计目标

你可以在以下几点中了解到，决定 TypeScript 作为程序语言发展到今天这个形态的设

计目标和架构：

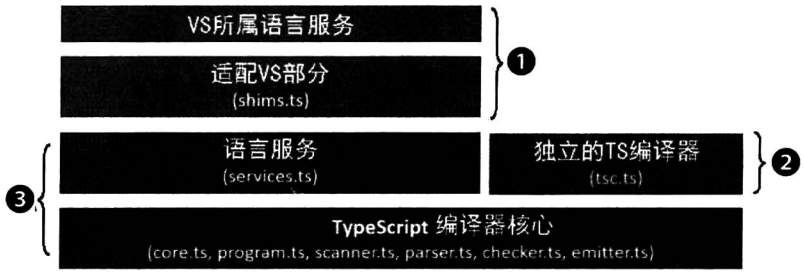
- 对 JavaScript 结构的静态分析很有可能是错误的。微软的工程师们认为防止并排查一些运行时错误的最佳方式是，创造一种在编译期进行静态类型分析的强类型语言。他们同时也设计了一个语言服务层给开发者提供一些更好的工具。
- 与现存的 JavaScript 代码有非常高的兼容性。TypeScript 是 JavaScript 的超集，这意味着任何合法的 JavaScript 程序都是合法的 TypeScript 程序（很少有例外）。
- 给大型项目提供一个构建机制。TypeScript 中加入了基于类（Class）的对象、接口和模块。这些特性可以帮助我们以更好的方式构建代码，也会减少团队内代码集成的时候带来的问题，并且结合最佳的面向对象原则和实践，可以让代码更具可维护性与扩展性。
- 对于发行版本的代码，没有运行时开销。在使用 TypeScript 时，我们通常将程序设计阶段和运行阶段区分开来。术语*设计时代码*（design time code）指的是我们设计程序时编写的 TypeScript 代码，术语*执行时代码*（execution time code）或者*运行时代码*（runtime code）指的是 TypeScript 代码编译后执行的 JavaScript 代码。TypeScript 给 JavaScript 新增了一些特性，但这些特性只在程序设计阶段可以用到。比如，可以在 TypeScript 里声明接口，但既然 JavaScript 不支持接口，那么 TypeScript 编译器在编译出的 JavaScript 代码中就不会声明也不会模拟这个特性。

  微软的工程师们提供了一些类似于代码转换（将 TypeScript 特性转变成纯 JavaScript 实现）和类型擦除（移除静态类型标记）的组件给 TypeScript 的编译器，来让它产生真正纯净的 JavaScript 代码。类型擦除组件不仅仅移除了类型的标注，还移除了所有的 TypeScript 高级语言特性，比如接口。

  并且，默认以 ECMAScript 3 规范为目标的时候生成的代码在浏览器中有非常高的兼容性，当然也支持以 ECMAScript 5 和 ECMAScript 6 为编译目标。一般来说，使用任意受支持的编译目标，我们都可以使用 TypeScript 的特性，但部分特性需要依赖 ECMAScript 5 或更高版本作为编译目标这一先决条件。
- 遵循当前以及未来出现的 ECMAScript 规范。TypeScript 不仅能兼容现有的 JavaScript 代码，它也拥有兼容未来版本的 JavaScript 的能力。大多数 TypeScript 的新增特性都是基于未来的 JavaScript 提案，这意味着许多 TypeScript 代码在将来甚至会变成合法的 JavaScript 代码。
- 成为跨平台的开发工具。微软使用 Apache 协议开源了 TypeScript，并且它可以在所有主流的操作系统上安装和执行。

TypeScript 组件

TypeScript 语言内部主要被分为三层，每一层又被依次分为子层或者组件。在下面这张示意图中，我们可以看到三个层（❶、❷和❸）和每一层内包含的组件（方框）：



在上图中，VS 缩写代表 Microsoft's Visual Studio，是所有微软产品（包括 TypeScript）的官方一体化开发工具。我们将在下一章进一步了解集成开发环境（IDE）。

每一个主要的层都有不同的用途。

- **语言层：**实现所有 TypeScript 的语言特性。
- **编译层：**执行编译、类型检查，然后将 TypeScript 代码转换成 JavaScript 代码。
- **语言服务层：**生成信息以帮助编辑器和其他工具来提供更好的辅助特性，比如 IntelliSense，这是微软提供的一种代码检查与提示服务，可以在编码时提供相关信息。更多相关信息可访问 https://msdn.microsoft.com/zh-cn/library/bb385682.aspx 或自动化重构工具。
- **IDE 整合：**为了利用 TypeScript 的特性，IDE 的开发者需要完成一些集成工作。TypeScript 让工具的发展更加便利从而提高 JavaScript 开发者的生产力而设计。而这些努力带来的结果是，将 TypeScript 集成到 IDE 并不是一件复杂的事情。一个很好的证明是如今最流行的 IDE 都集成了良好的 TypeScript 支持。

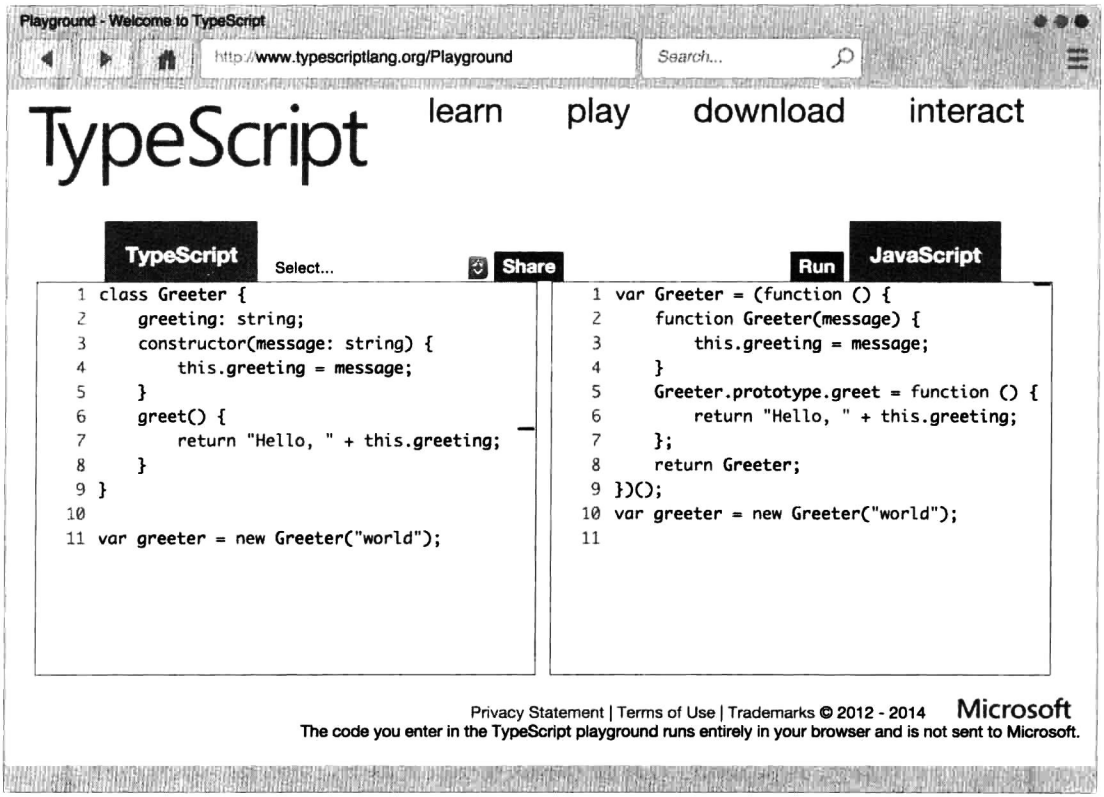
在有关 TypeScript 的其他书籍或者在线资源中，你可能发现有另外一个术语——转译器（transpiler）被提及，而不是编译器（compiler）。转译器也是编译器的一种，它将一种程序语言的源码作为输入，然后输出另一种相似抽象等级的程序的源码。

我们没有必要了解更多关于 TypeScript 编译器原理的细节，这超出了本书的讨论范围。如果你希望了解更多与之相关的知识，比如 TypeScript 语言层面的各种细节，可以移步在线网址：`http://www.typescriptlang.org/`。

TypeScript 语言特性

到这里，你已经知道了 TypeScript 产生的缘由，现在是时候捋起袖子写一些代码了。

在开始学习如何使用一些基本的 TypeScript 构建模块之前，需要先设置好开发环境。要编写 TypeScript 代码，最快捷的方式是使用官方网站提供的在线编辑器 `http://www.typescriptlang.org/Playground`，该编辑器的界面如下图所示。



如上图所示，可以使用左边的编辑器编写 TypeScript 代码，你的代码会自动被编译成 JavaScript 并且插入到右边的文本编辑器中。如果编写的 TypeScript 代码是非法的，那么右

边的 JavaScript 代码就不会刷新。

或者，如果你更倾向于离线编写代码，可以下载并安装 TypeScript 的编译器。如果你使用 Visual Studio，可以从 https://visualstudiogallery.msdn.microsoft.com/107f89a0-a542-4264-b0a9-eb91037cf7af 下载官方 TypeScript 插件（1.5 beta 版）。如果你使用的是 Visual Studio 2015，它默认支持 TypeScript，因此不需要额外下载扩展。

如果你使用其他代码编辑器或者使用的是 OS X 或 Linux 操作系统，可以下载一个 npm 模块。如果你不熟悉 npm 也不需要担心，到目前为止，只需要知道它是 Node 包管理的标准并且是 Node.js 的默认包管理工具即可。

很多流行的编辑器都有 TypeScript 插件，比如 Sublime https://github.com/Microsoft/TypeScript-Sublime-Plugin 和 Atom https://atom.io/opackages/atom-typescript。

为了能使用 npm，你需要先在开发环境中安装 Node.js。可以在 Node.js 的官方网站上 https://nodejs.org/找到相应的安装文件。

在开发环境中安装了 Node.js 之后，就可以在控制台运行以下命令了：

```
npm install -g typescript
```

当全局安装（-g）npm 包时，OS X 用户需要使用 sudo 命令。sudo 命令会提示用户输入凭据，并且使用管理员权限安装 npm 包：

```
sudo npm install -g typescript
```

接着创建一个名为 test.ts 的文件，然后在其中写入下面的代码：

```
var t : number = 1;
```

选择一个文件夹并将这个文件保存进去，然后打开控制台，进入刚才选择的文件夹所在位置，执行以下命令：

```
tsc test.ts
```

tsc 是 TypeScript 编译器的控制台接口，这个命令可以将 TypeScript 文件编译成 JavaScript 文件。它的其他功能将在本书接下来的章节中介绍到。

在刚才的例子中，我们使用 tsc 命令将 test.ts 文件编译成一个 JavaScript 文件。

不出差错的话，你会在与 test.ts 相同的目录下找到 test.js 文件。现在，你已经知

道如何将 TypeScript 代码编译成 JavaScript 代码了，我们可以开始学习 TypeScript 的特性了。

可以在第 2 章中对编辑器和其他工具有更多的了解。

类型

我们已经了解了 TypeScript 是 JavaScript 的超集。TypeScript 通过向 JavaScript 增加可选的静态类型声明来把 JavaScript 变成强类型的程序语言。可选的静态类型声明可约束函数、变量、属性等程序实体，这样编译器和相应的开发工具就可以在开发过程中提供更好的正确性验证和帮助功能（比如 IntelliSense）。

强类型特性能让程序员对自己和其他开发团队人员在代码中表达他的意图。

TypeScript 的类型检测在编译期进行并且没有运行时开销。

可选的静态类型声明

TypeScript 非常擅长类型推导，但是在很多情况下没有办法自动侦测出一个对象或者变量的类型。在这些情况下，TypeScript 允许我们明确地声明一个变量的类型。这种允许声明变量类型的功能就是被大家所熟知的*可选的静态类型声明*（optional static type notation）。对于变量来说，类型声明在变量名后面并且前面有一个冒号：

```
var counter;                    // 未知(any)类型
var counter = 0;                // number类型（推断出的）
var counter : number;           // number类型
var counter : number = 0;       // number类型
```

可以看到，变量的类型声明在变量名后面这种风格是基于类型理论，且更强调类型是可选的。当没有类型声明的时候，TypeScript 会尝试检查赋值给变量的值来推测变量的类型。举例来说，在上面代码片段的第二行里，我们可以看到变量 counter 被识别出是 number 类型，因为它被赋值为 number 类型的 0。这种类型被自动推测出来的过程被称为*类型推导*（type inference），当一个变量的类型无法被推测时，一个特殊的类型 any 会作为它的类型。

变量、基本类型和运算符

基本类型有 boolean、number、string、array、void 和所有用户自定义的 enum 类型。所有这些类型在 TypeScript 中，都是一个唯一的顶层的 Any Type 类型的子类型，any 关键字代表这种类型。让我们看一下这些原始类型。

类型	描述
boolean	与 string 和 number 数据类型在理论上可以有无数个值不同，boolean 类型只可能是两种值。它们分别是 true 和 false。一个 boolean 值是一个真实性的值，它代表条件是否正确。 `var isDone: boolean = false;`
number	和在 JavaScript 中一样，所有的数字在 TypeScript 中都是浮点数。这些浮点数全部都是 number 类型。 `var height: number = 6;`

string	在 TypeScript 中，string 类型用来表示文本。在代码中使用字符串是将它们放在引号或者双引号中间。单引号中的字符串可以包含双引号，双引号中的字符串也可以包含单引号。
	`var name: string = "bob";`
	`name = 'smith';`

array	和 JavaScript 一样， TypeScript 允许使用数组。array 类型的声明有两种写法。第一种，可以在数组元素的类型后面跟着[]来表示包含这种类型元素的数组：
	`var list:number[] = [1, 2, 3];`
	第二种是使用范型数组类型 Array：
	`var list:Array<number> = [1, 2, 3];`

enum	enum 类型是为了给一个数字集合更友好地命名。enum 类型中的成员默认从 0 开始，但你也可以手动设置成员中的值来更改这种默认行为。
	`enum Color {Red, Green, Blue};`
	`var c: Color = Color.Green;`

any	any 类型可以表示任意 JavaScript 值。一个 any 类型的值支持所有在 JavaScript 中对它的操作，并且对一个 any 类型的值操作时仅进行最小化静态检查。
	`var notSure: any = 4;`
	`notSure = "maybe a string instead";`
	`notSure = false; //合法行为定义为一个布尔类型`
	使用 any 类型是与现存的 JavaScript 代码一起工作的一种高效的方式，你可以在编译期逐步添加或者去除类型检查。any 类型在你只知道一部分类型的情况下也很方便，比如你可能有一个混合了各种类型元素的数组：
	`var list:any[] = [1, true, "free"];`
	`list[1] = 100;`

续表

类型	描述

void	在某种程度上，any 的对立面就是 void，即所有的类型都不存在的时候。你会在一个函数没有返回值时看到它：
	`function warnUser(): void {`
	`  alert("This is my warning message");`
	`}`

JavaScript 的原始类型也包括了 undefined 和 null。在 JavaScript 中，undefined 是全局作用域的一个属性，它会赋值给那些被声明但未被初始化的变量。null 是一个字面量（不是全局对象的一个属性），它可以被赋值给那些表示没有值的变量。

```
var TestVar;              // 变量声名但未初化
alert(TestVar);           // 显示undefined
alert(typeof TestVar);    // 显示undefined

var TestVar = null;       // 变量被声名，并且被赋值为null
assigned as value
alert(TestVar);           // 显示null
alert(typeof TestVar);    // 显示objcect
```

在 TypeScript 中，我们不能把 null 或 undefined 当作类型使用:

```
var TestVar : null;       // 错误，类型错误
var TestVar : undefined; // 错误，找不到 undefined
```

因为 null 和 undefined 都不能被当作类型使用，所以上面这些代码都是不合法的。

var、let 和 const

在 TypeScript 中，当声明一个变量时，可以使用 var、let 和 const 关键字:

```
var mynum : number = 1;
let isValid : boolean = true;
const apiKey : string = "0E5CE8BD-6341-4CC2-904D-C4A94ACD276E";
```

使用 var 声明的变量保存在最近的函数作用域中（如果不在任何函数中则在全局作用域中）。

使用 let 声明的变量保存在最近的比函数作用域小的块作用域中（如果不在任何块中则在全局作用域中）。

const 关键字会创建一个保存在创建位置作用域中的常量，可以是全局作用域也可以是块作用域。这表明 const 是块作用的，可在第 5 章进一步了解作用域的知识。

let 与 const 关键字在 TypeScript 1.4 中就可以使用了，但只有在编译目标为 ECMAScript 6 时可用。然而，在 TypeScript 1.5 版本到来时，就可以在编译目标为 ECMAScript 5 与 ECMAScript 3 时使用它们了。

联合类型

TypeScript 允许声明联合类型：

```
var path : string[]|string;
path = '/temp/log.xml';
path = ['/temp/log.xml', '/temp/errors.xml'];
path = 1; // 错误
```

联合类型用来声明那些可以存储多种类型值的变量。在上面这个例子中，我们声明了一个可以存储一个（字符串）或者一组路径（字符串数组）的变量 path。在例子中，我们也对这个变量进行了赋值，将字符串和字符串的数组合法地赋值给了这个变量。然而，当试图将一个数字赋值给它时我们遇到了一个编译错误，因为这个联合类型并没有声明 number 为它的合法类型。

类型守护

可以在运行时使用 typeof 或者 instanceof 运算符对类型进行验证。TypeScript 语言服务会在 if 区域寻找这些运算符，然后对应地更改类型：

```
var x: any = { /* ... */ };
if(typeof x === 'string') {
  console.log(x.splice(3, 1)); // 错误，'string'上不存在'splice'方法
}
// x 依然是 any 类型
x.foo(); // 合法
```

在这段代码中，我们首先声明了一个 any 类型的变量 x，随后在运行时通过 typeof 运算符对 x 进行了类型检查。如果 x 的类型为 string 时，我们就会尝试调用被认为是 x 的一

个成员的 splice 方法。TypeScript 语言服务可以读懂在条件语句中使用 typeof 的用法。TypeScript 会自动推断出 x 一定是 string 类型，然后告诉我们 splice 方法不存于 string 类型上。这种特性被称为*类型守护*。

类型别名

TypeScript 允许使用 type 关键字声明类型别名：

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

类型别名实质上与原来的类型一样，它们仅仅是一个替代的名字。类型别名可以让代码的可读性更高，但是它也会导致一些问题。

如果你在一个很大的团队中工作，毫无约束地创建类型别名会导致可维护性的问题。在 *Maintainable JavaScript*（由 Nicholas C. Zaka 所著）一书中，作者建议应避免修改一个不属于你的对象。Nicholas 说的是，避免对那些不是你自己声明的对象（DOM 对象、BOM 对象、原始类型和第三方库）进行修改和覆盖，我们同样能将其应用到别名的使用上。

环境声明

环境声明允许在 TypeScript 代码中创建一个不会被编译到 JavaScript 中的变量。这个特性是用来促进与现有 JavaScript 代码、DOM（文档对象模型），还有 BOM（浏览器对象模型）结合而设计的。让我们看一个例子：

```
customConsole.log("A log entry!"); // 错误
```

如果你尝试调用 customConsole 对象上的 log 方法，TypeScript 会告诉我们 customConsole 对象未被声明：

```
// Cannot find name 'customConsole'
```

出现这种情况并不令人意外。但是，有时候我们希望调用一个未被定义的对象上的方法，比如 window 对象上的 console 方法。

```
console.log("Log Entry!");
var host = window.location.hostname;
```

当访问 DOM 或 BOM 对象时，我们没有遇到错误，是因为这些对象已经在一个特殊的 TypeScript 文件（被称为*声明文件*）中被声明了。可以使用 declare 操作符创建一个环境声明。

在下面这段代码中，我们会声明一个被 customConsole 对象实现的接口。然后使用 declare 操作符在作用域中增加一个 customConsole 对象：

```
interface ICustomConsole {
  log(arg : string) : void;
}
declare var customConsole : ICustomConsole;
```

稍后在本章中会解释大量关于接口的细节。

然后就可以在没有编译错误的情况下使用 customConsole：

```
customConsole.log("A log entry!"); // 成功
```

TypeScript 默认包含一个名为 lib.d.ts 的文件，它提供了像 DOM 这种 JavaScript 内置库的接口声明。

使用.d.ts 结尾的声明文件，是用来提高 TypeScript 对第三方库和像 Node.js 或浏览器这种运行时环境的兼容性的。

我们会在第 2 章介绍如何使用.d.ts 文件。

算术运算符

下表中列出的是 TypeScript 支持的算术运算符。为了便于理解下面的例子，设置变量 A 的值总是 10，变量 B 的值总是 20。

运算符	描述	例子
+	将两个数相加	A + B = 30
-	第一个数减去第二个数	A – B = 10
*	两个数相乘	A * B = 200
/	分子除以分母	B / A = 2

续表

运算符	描述	例子
%	求余数并保留整数	B % A = 0
++	自增操作，自加 1	A++ = 11
--	自减操作，自减 1	A-- = 9

比较运算符

下表中列出的是 TypeScript 支持的比较运算符。为了便于理解下面的例子，设置变量 A 的值总是 10，变量 B 的值总是 20。

运算符	描述	例子
==	比较两个运算元是否相等，如果相等则结果为 true	(A == B)为 false，A == '10' 为 true
===	比较两个运算元的值和类型是否都相等，如果都相等则结果为 true	(A === B)为 false，A === '10' 为 false
!=	比较两个运算元是否不等，如果不等则结果为 true	(A != B) 为 true
!==	比较两个运算元的类型和值是否都不等，如果都不等则结果为 true	(A !== '10') 为 true
>	比较左边的运算元是否大于右边的运算元，如果大于则为 true	(A > B) 为 false
<	比较左边的运算元是否小于右边的运算元，如果小于则为 true	(A < B) 为 true
>=	比较左边的运算元是否大于或等于右边的运算元，如果大于或者等于则为 true	(A >= B) 为 false
<=	比较左边的运算元是否小于或等于右边的运算元，如果小于或者等于则为 true	(A <= B) 为 false

逻辑运算符

下表中列出的是 TypeScript 支持的逻辑运算符。为了便于理解下面的例子，设置变量 A 的值总是 10，变量 B 的值总是 20。

运算符	描述	例子
&&	称为逻辑与操作符，如果两个运算元都为非零，则结果为 true	(A && B)为 true

续表

运算符	描述	例子
|	称为逻辑或操作，如果两个运算元任意一个为非零，则结果为 true	(A || B)为 true
!	称为逻辑非操作，用来对运算元取反。如果一个条件是 true，那么逻辑非操作会让它变为 false	!(A && B)为 false

位运算符

下表中列出的是 TypeScript 支持的位运算符。为了便于理解下面的例子，设置变量 A 的值总是 2，变量 B 的值总是 3。

运算符	描述	例子
&	称为按位与操作符。对两个运算元的每一个二进制位进行逻辑与操作	(A & B) 为 2
|	称为按位或操作符。对两个运算元的每一个二进制位进行逻辑或操作	(A | B) 为 3
^	称为按位异或操作符。对两个运算元的每一个二进制位进行异或操作。异或操作的意思是两个运算元不同时为 true，相同则为 false	(A ^ B) 为 1
~	称为按位取反操作符。这是一个一元操作符，它对操作元的每一个二进制位取反	(~B) 为 -4
<<	称为左移位操作符。将第一个操作元的二进制形式向左移第二个操作元个比特位，右边用 0 填充。移一位相当于乘以 2，移两位相当于乘以 4，依此类推	(A << 1) 为 4
>>	称为有符号右移位操作符。将第一个操作元的二进制形式向右移第二个操作元个比特位，左边用符号位填充	(A >> 1) 为 1
>>>	称为无符号右移操作符。与有符号右移位类似，除了左边一律使用 0 补位	(A >>> 1) 为 1

让我们像 C++、Java 或者 C#那样使用移位操作的一个主要原因是它非常快。但是通常认为，位操作在 TypeScript 和 JavaScript 中并没有那么高效。位操作在 JavaScript 中效率不如其他语言的原因是，它必须先将操作元从浮点型（JavaScript 存储数字的数据类型）转换成 32 位整型进行运算，然后再转换回浮点型。

赋值操作符

下表中列出的是 TypeScript 支持的赋值操作符。

运算符	描述	例子
=	这是最简单的等于操作符，将右边操作元的值赋给左边的操作元	C = A + B 会将 A + B 的结果赋值给 C
+=	这是加等于操作符，它将右边的操作元与左边的操作元相加后赋值给左边的操作元	C += A 等价于 C = A + C
- =	这是减等于操作符，它将左边的操作元减去右边的操作元再赋值给左边的操作元	C - = A 等价于 C = C - A
*=	这是乘等于操作符，它将右边的操作元乘以左边的操作元后赋值给左边的操作元	C *= A 等价于 C = C * A
/=	这是除等于操作符，它将左边的操作元除以右边的操作元后赋值给左边的操作元	C /= A 等价于 C = C / A
%=	这是余等于操作符，它将两个操作元取余后赋值给左边的操作元	C %= A 等价于 C = C % A

流程控制语句

这一节将描述 TypeScript 中的选择语句、循环语句和分支语句。

单一选择结构（if）

下面这段代码声明了一个 boolean 类型的变量 `isValid`。然后，一个 `if` 语句会判断 `isValid` 的值是否为 `true`。如果判断结果为 true，则在屏幕上会显示消息 `Is valid!`。

```
var isValid : boolean = true;

if(isValid) {
  alert("is valid!");
}
```

双选择结构（if…else）

下面这段代码声明了一个 boolean 类型的变量 `isValid`。然后，一个 `if` 语句会判断 `isValid` 的值是否为 `true`。如果判断结果为 true，则在屏幕上会显示消息 `Is valid!`。另一方面，如果判断结果为 false，在屏幕上会显示消息 `Is NOT valid!`。

```
var isValid : boolean = true;

if(isValid) {
  alert("Is valid!");
}
else {
  alert("Is NOT valid!");
}
```

三元操作符（?）

三元操作符是双选择结构的一种替代形式。

```
var isValid : boolean = true;
var message = isValid ? "Is valid!" : "Is NOT valid!";
alert(message);
```

上面这段代码声明了一个 boolean 类型的变量 isValid。然后它判断操作符 ? 左边的变量或表达式是否等于 true。

如果判断结果为 true，则会执行冒号左边的表达式，Is valid!会被赋值给变量 message。

另一方面，如果判断结果为 false，则会执行冒号右边的表达式，Is NOT valid!会被赋值给变量 message。

最后，变量 message 的值会显示在屏幕上。

多选结构（switch）

switch 语句接受一个表达式，将表达式的值与 case 语句进行匹配，然后执行关联到这种情况下的语句。switch 语句经常与枚举类型的变量一起使用来提高代码的可读性。

在下面这个例子中，我们声明了一个接受枚举类型参数 AlertLevel 的函数。我们在这个函数内部生成一个字符串数组存储 E-mail 地址然后执行 switch 语句。枚举变量中的每一个选项都对应着 switch 结构内的一个 case：

```
enum AlertLevel{
  info,
  warning,
```

```
    error
}

function getAlertSubscribers(level: AlertLevel) {
  var emails = new Array<string>();

  switch (level) {
    case AlertLevel.info:
      emails.push("cst@domain.com");
      break;
    case AlertLevel.warning:
      emails.push("development@domain.com");
      emails.push("sysadmin@domain.com");
      break;
    case AlertLevel.error:
      emails.push("development@domain.com");
      emails.push("sysadmin@domain.com");
      emails.push("management@domain.com");
      break;
    default:
      throw new Error("Invalid argument!");
  }
  return emails;
}
getAlertSubscribers(AlertLevel.info); // ["cst@domain.com"]
getAlertSubscribers(AlertLevel.warning); //
["development@domain.com", "sysadmin@domain.com"]
```

变量 level 的值会与 switch 中所有的 case 值进行匹配。如果其中一个值与其匹配，那么与这个 case 关联的语句将会被执行。一旦这个 case 语句执行完毕，这个变量的值就会与下一个 case 进行匹配。

当一个 case 中的语句执行完毕后，下一个满足条件的 case 语句就会接着执行。如果 break 关键字出现在 case 语句中，程序就不会继续匹配接下来的 case 语句了。

如果没有匹配到任何 case 语句，程序寻找可选的 default 语句，如果找到，它将控制程序进入这个语句并且执行其中的代码。

如果没有找到 default 语句，程序将会继续执行 switch 表达式后面的语句。按照惯

例，default 语句放在最后的位置，但这并不是一个强制性的写法。

语句在顶部进行判断的循环（while）

while 语句被用来在满足条件的情况下重复一个操作。比如下面这段代码，声明一个数字类型的变量 i，当条件（i 小于 5）满足时，将会执行一个操作（i 加 1 然后在浏览器的控制台中打印它的值）。当这个操作完成后，将会再次判断循环的条件。

```
var i : number = 0;
while (i < 5) {
  i += 1;
  console.log(i);
}
```

在 while 语句中，语句内的操作只在 while 条件满足时执行。

语句在底部进行判断的循环（do…while）

do...while 语句被用来重复一个操作直到条件不再被满足。比如下面这段代码，声明一个数字类型的变量 i，在条件（i 小于 5）满足时一直执行一个操作（i 加 1 然后在浏览器的控制台中打印它的值）。

```
var i: number = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

和 while 语句不一样，do...while 语句会在判断 while 条件是否满足之前至少执行一次，不管条件是否满足。

迭代对象的属性（for…in）

for...in 语句本身并不是一个坏的实践，然而它可能会被滥用。例如，迭代一个数组或者类数组对象。for...in 语句的原意是枚举对象的属性。

```
var obj: any = { a: 1, b: 2, c: 3 };
for (var key in obj) {
```

```
  console.log(key + " = " + obj[key]);
}

// 输出:
// "a = 1"
// "b = 2"
// "c = 3"
```

这段代码会沿着原型链，将继承的属性也进行枚举。for…in 语句会沿着对象的原型链迭代，枚举出包括继承的属性的所有属性。如果只想枚举对象自己的属性（非继承属性），可以使用 hasOwnProperty 方法：

```
for (var key in obj) {
  if (obj.hasOwnProperty(prop)) {
    // prop没有被继承
  }
}
```

计数器控制循环（for）

for 语句会创建一个包含三个可选表达式的循环，表达式在圆括号中用分号分隔，紧跟一个或者一些在循环中执行的语句：

```
for (var i: number = 0; i < 9; i++) {
  console.log(i);
}
```

上面这段代码包含一个 for 语句，它以声明一个变量 i 并初始化为 0 开始。第二个语句判断 i 是否小于 9，然后每次循环的时候将 i 加 1。

函数

就像 JavaScript 一样，TypeScript 的函数也可以通过具名或匿名的方式创建。这使我们可以根据应用中的具体情况，选择合适的方式，不论是在构建 API 时，或创建供其他函数调用的中间函数时。

```
// 具名函数
function greet(name?: string): string {
```

```
  if (name) {
    return "Hi! " + name;
  }
  else
  {
    return "Hi!";
  }
}

// 匿名函数
var greet = function(name?: string): string {
  if (name) {
    return "Hi! " + name;
  }
  else
  {
    return "Hi!";
  }
}
```

正如上述代码所示，在 TypeScript 中，不仅可以为函数的参数加上类型，也可以给函数的返回值指定类型。TypeScript 会通过查看函数里的 `return` 语句，来检查返回值的类型正确与否，并且它们都不是必需的。

如果不想使用函数语法，还有另一种语法可以选择，也可以在函数的返回值类型后加上箭头（=>）操作符并不使用 `function` 关键字：

```
var greet = (name: string): string => {
  if (name) {
    return "Hi! " + name;
  }
  else {
    return "Hi! my name is " + this.fullname;
  }
};
```

使用这种语法声明的函数通常都称作箭头函数。继续回到上述例子，还可以给 `greet` 变量添加上匹配匿名函数的类型。

```
var greet: (name: string) => string = function(name: string):
  string {
  if (name) {
    return "Hi! " + name;
  }
  else
  {
    return "Hi!";
  }
};
```

一个需要注意的地方是，当处于类的内部时，使用箭头函数（=>）语法将会改变 this 操作符的工作机制。在后面的章节里，我们将会详细讨论它。

现在我们已经学习了如何将一个变量强制描述为指定形式的函数。这在我们使用回调函数（作为另一个函数的参数）时，十分有用。

```
function sume(a: number, b: number, callback: (result: number)
  => void) {
  callback(a + b);
}
```

在上述例子里，我们声明了一个名为 sume 的函数，并且指定了两个 number 类型的参数和第三个函数类型的 callback 参数。回调函数上的类型声明将会限制 callback 参数为一个仅接受一个 number 类型的参数，且无返回值的函数。

我们将会在第 3 章详细学习函数。

类

在 ECMAScript 6（即最新版本的 JavaScript）中，添加了基于类的面向对象编程语法。由于 TypeScript 是基于 ES6 的，所以开发者如今就已经可以开始使用基于类的面向对象的语法了。TypeScript 的编译器会负责将 TypeScript 代码编译为兼容主流浏览器和平台的

JavaScript 代码。

让我们来看一个在 TypeScript 中定义类的例子：

```
class Character {
  fullname: string;
  constructor(firstname: string, lastname: string) {
    this.fullname = firstname + " " + lastname;
  }
  greet(name?: string) {
    if (name) {
      return "Hi! " + name + "! my name is " + this.fullname;
    } else {
      return "Hi! my name is " + this.fullname;
    }
  }
}

var spark = new Character("Jacob", "Keyes");
var msg = spark.greet();
alert(msg); // "Hi! my name is Jacob Keyes"
var msg1 = spark.greet("Dr. Halsey");
alert(msg1); // "Hi! Dr. Halsey! my name is Jacob Keyes"
```

在上面的例子里，我们定义了一个名为 Character 的新类。这个类有三个成员：一个名为 fullname 的属性，一个构造函数 constructor，和一个 greet 方法。当我们在 TypeScript 中声明类时，所有的属性和方法默认都是公共的。

你可能已经留意到，当（在对象内部）访问对象内的成员时，我们都在前面加上了 this 操作符，this 操作符表明了这是一个成员访问操作。我们使用 new 操作符构造了 Character 类的一个实例，这会调用类的构造函数，按照定义对实例进行初始化。

为了兼容 ECMAScript 3 和 ECMAScript 5，TypeScript 中的类会被编译为 JavaScript 中的函数。

关于类和其他面向对象编程的概念，我们将在第 4 章中进行深入学习。

接口

在 TypeScript 中，可以使用接口来确保类拥有指定的结构。

```
interface LoggerInterface {
  log(arg: any): void;
}
class Logger implements LoggerInterface {
  log(arg) {
    if (typeof console.log === "function") {
      console.log(arg);
    } else {
      alert(arg);
    }
  }
}
```

在上面的例子里，我们定义了一个名为 loggerInterface 的接口，和一个实现了它的 Logger 类。TypeScript 也允许使用接口来约束对象。这使我们可以避免很多潜在的小错误，尤其是在写对象字面量时：

```
interface UserInterface{
  name : string;
  password : string;
}
var user : UserInterface = {
  name : "",
  pasword : "" // password 遗漏错误属性
};
```

关于接口和相关的其他面向对象编程的概念，我们将在第 4 章中进行深入学习。

命名空间

命名空间，又称内部模块，被用于组织一些具有某些内在联系的特性和对象。命名空

间能够使代码结构更清晰，可以使用 namespace 和 export 关键字，在 TypeScript 中声明命名空间。

```
namespace Geometry{
  interface VectorInterface {
  /* ... /
  }
  export interface Vector2dInterface {
    /* ... */
  }
  export interface Vector3dInterface {
    /* ... /
  }
  export class Vector2d implements VectorInterface, Vector2dInterface {
    /* ... /
  }
  export class Vector3d implements VectorInterface, Vector3dInterface {
    /* ... /
  }
}

var vector2dInstance: Geometry.Vector2dInterface = new Geometry.Vector2d();
var vector3dInstance: Geometry.Vector3dInterface = new Geometry.Vector3d();
```

在上面的例子里，我们声明了一个包含了 Vector2d、Vector3d 类和 VectorInterface、Vector2dInterface、Vector3dInterface 接口的命名空间。注意，命名空间内的第一个接口声明前并没有 export 关键字。所以，在命名空间的外部，我们访问不到它。

关于命名空间（内部模块）和外部模块，以及各自适用的场景，我们将在第 4 章继续探讨。

综合运用

我们已经逐个学习了如何使用 TypeScript 语言中的各个特性。在接下来的例子里，我们将会同时使用模块、类、函数和类型注解：

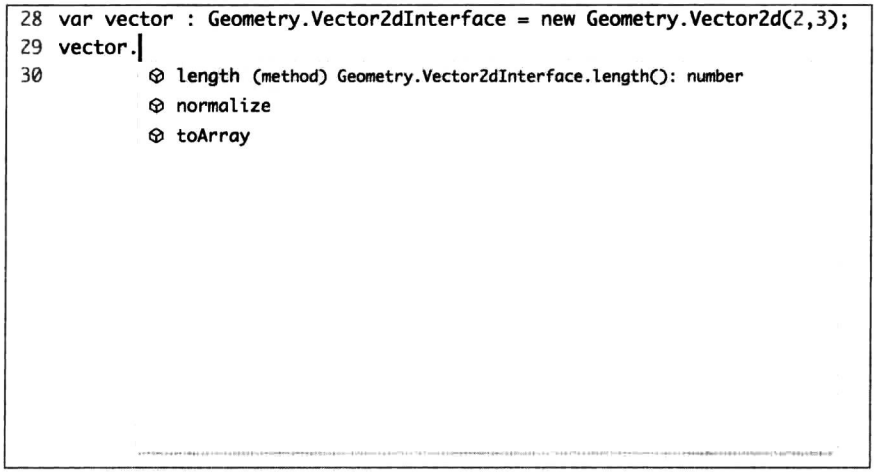
```
module Geometry {
  export interface Vector2dInterface {
    toArray(callback: (x: number[]) => void): void;
    length(): number;
    normalize();
  }
  export class Vector2d implements Vector2dInterface {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
      this._x = x;
      this._y = y;
    }
    toArray(callback: (x: number[]) => void): void {
      callback([this._x, this._y]);
    }
    length(): number {
      return Math.sqrt(this._x * this._x + this._y * this._y);
    }
    normalize() {
      var len = 1 / this.length();
      this._x *= len;
      this._y *= len;
    }
  }
}
```

上面的例子是一个简单的 JavaScript 3D 引擎代码中的一小部分。在 3D 引擎中，有大量关于矩阵和矢量的计算。我们先定义了一个包含许多成员的 `Geometry` 模块。为了保证例子的简短，我们只在其中添加了一个 `Vector2d` 类。这个 2D 空间类中存储了两个坐标（`x` 和 `y`），并且包含了一些相关操作和计算。对于大多数矢量类来说，最重要的操作就是单位化（**normalize**），即 `Vector2d` 类中的一个方法。

3D 引擎是一个复杂的软件，作为一个开发者，你可能更倾向于直接使用第三方的 3D 引擎，而不是自己写一个。不过不用担心，TypeScript 不仅可以让你在开发大型应用时更轻松，还能够让你在使用一些大型应用时更为轻松。在下面的例子中，我们将使用之前定义好的模块来创建一个 `Vector2d` 类的实例：

```
var vector : Geometry.Vector2dInterface = new Geometry.Vector2d(2,3);
vector.normalize();
vector.toArray(function(vectorAsArray : number[]){
  alert(' x :' + vectorAsArray[0] + ' y : '+ vectorAsArray[1]);
});
```

类型检查和智能提示将会帮助我们创建 `Vector2d` 类的实例，单位化它的值，将其转化为数组，然后在屏幕上展示出来。



小结

在本章中，我们学习了 TypeScript 想要解决的问题，以及微软的 TypeScript 工程师创建它时，使用的一些设计理念。

在本章的后半段，我们写了很多 TypeScript 代码片段。你开始第一次书写 TypeScript 代码，并且使用类型注解、变量、原始数据类型、操作符、流程控制语句、函数和类。在下一章中，我们将学习如何自动化开发流程。

2

自动化工作流程

在学习了 TypeScript 的主要语言特性之后，我们现在将学习如何使用一些工具来自动化开发流程。这些工具会将我们从重复劳动中解放出来。

在本章，我们将会涵盖以下话题：

- 开发工作流概览
- 版本控制工具
- 包管理工具
- 自动化任务工具
- 自动化测试工具
- 持续集成工具
- 脚手架工具

一个现代化的开发工作流程

如今，开发一个高质量的 Web 应用是一件非常耗时的工作。如果想让应用有较好的用户体验，就需要让应用在不同的浏览器、设备、网络连接速度和屏幕分辨率下，都有相对一致的表现。此外，质量审查和性能优化也都将花费不少时间。

作为开发者，我们需要尽可能地减少花在重复劳动上的时间。这几年中，我们已经做了许多尝试。从最开始的写构建脚本（如 Makefile），到后来自动化测试。如今，已经有了大量的可用工具，来完成大多数的自动化任务。这些工具可以被归类为：

- 版本控制工具

- 包管理工具
- 自动化任务工具
- 自动化测试工具
- 持续集成工具
- 脚手架工具

准备工作

在学习编写自动化工作流程的脚本之前，我们需要在自己的开发环境中安装好一些必要的工具。

Node.js

Node.js 是一个基于 V8（谷歌的开源 JavaScript 引擎）的运行平台，它使我们能够在浏览器之外运行 JavaScript 代码。有了 Node.js，我们便可以开始使用 JavaScript 来编写服务器端代码和桌面应用了。

虽然我们将要做的并不是编写服务端代码，但是由于在本章使用的众多工具都是用 Node.js 编写的应用，所以仍需要它。

如果你还未安装 Node.js，可以访问 `https://nodejs.org`，然后根据你的操作系统，下载对应的安装包。

Atom

Atom 是一个由 GitHub 团队开发的开源编辑器。它的开源社区非常活跃，已经有了大量的插件和主题，可以访问 `https://atom.io/`来下载它。

安装完毕后，打开编辑器，进入设置窗口。如下图所示，你将会看见一个管理插件的侧栏按钮，和一个管理主题的侧栏按钮：

根据操作系统的不同，Atom 的用户界面也会略有不同。如果你需要更多管理插件和主题的信息，请参阅 Atom 的文档：`https://atom.io/docs`。

我们需要在插件管理页中搜索 `atom-typescript` 插件，然后安装它。另外，也可以在主题管理页中，为我们的编辑器选择并安装一个漂亮的主题风格。

我们之所以使用 Atom，而不是 Visual Studio，是因为 Atom 在 Linux、OS X 和 Windows 上都可用，它能适用于更多的读者。

不幸的是，由于在本书即将出版时，Visual Studio Code 方才出现，所以在本书里我们将不会讨论它。Visual Studio Code 是一款由微软开发的跨平台的轻量级集成开发环境。如果你想要学习更多有关 Visual Studio Code 的内容，可以参阅 `https://code.visualstudio.com/`。如果你想使用 Visual Studio，那么可以从 `https://visualstudiogallery.msdn.microsoft.com/2d42d8dc-e085-45eb-a30b-3f7d50d55304` 下载 Visual Studio 的 TypeScript 插件。

`seti-ui` 是 Atom 中评价最高的主题之一，它为应用中的每一类文件都提供了不同的显示图标。例如，`gulpfile.js` 或 `bower.json`（我们将会在后文中学习它们）是 JavaScript 和 JSON 文件，`seti-ui` 主题会识别出它们分别是 Gulp 和 Bower 的配置文件，然后以不同的图标进行展示。

我们可以通过在操作系统的控制台内，输入以下命令来安装它:

```
cd ~/.atom/packages
git clone https://github.com/jesseweed/seti-ui --depth=1
```

在运行上述命令之前，你需要安装好 Git。在本章的后面，将会有关于安装 Git 的内容。

安装好了主题和 TypeScript 插件后，需要重启 Atom 编辑器，来使它们生效。如果一切正常，将会在编辑器的右上角看到如下提示：

Git 和 GitHub

在本章的最后，我们将会学习如何配置一个持续集成服务器。这个构建服务器将会观察应用代码中的变化，然后确保这些变化不会影响应用中的已有代码。

为了能够观察到代码的变化，我们需要一个版本控制系统。如今市面上有不少版本控制系统可供选择，其中使用最广泛的有 Subversion、Mercurial 和 Git。

使用版本控制系统有许多好处。首先，它使多个开发者能够修改同一个源文件，并且所有的修改都不会被覆盖。

其次，还可以使用版本控制系统来保存每一次修改后代码的备份。这些特性十分有用，比如，可以用它来查找某个 bug 第一次被写入代码的时间。

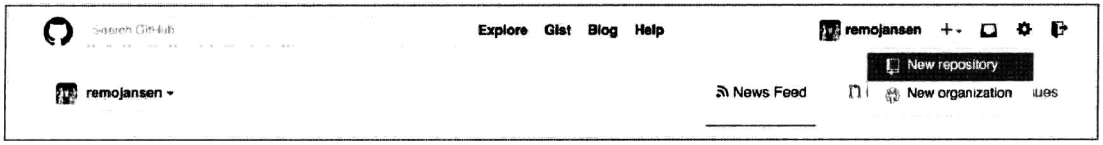
在下文的例子里，我们将会对源代码做一些改变，然后使用 Git 和 GitHub 来管理这些改变。你可以访问 `http://git-scm.com/downloads`，选择并下载适合你操作系统的 Git 安装包。然后访问 `https://github.com/`，注册一个 GitHub 账号，GitHub 提供了多种付费计划可供选择，其实对于下文中的例子来说，免费版本就已经足够了。

版本控制工具

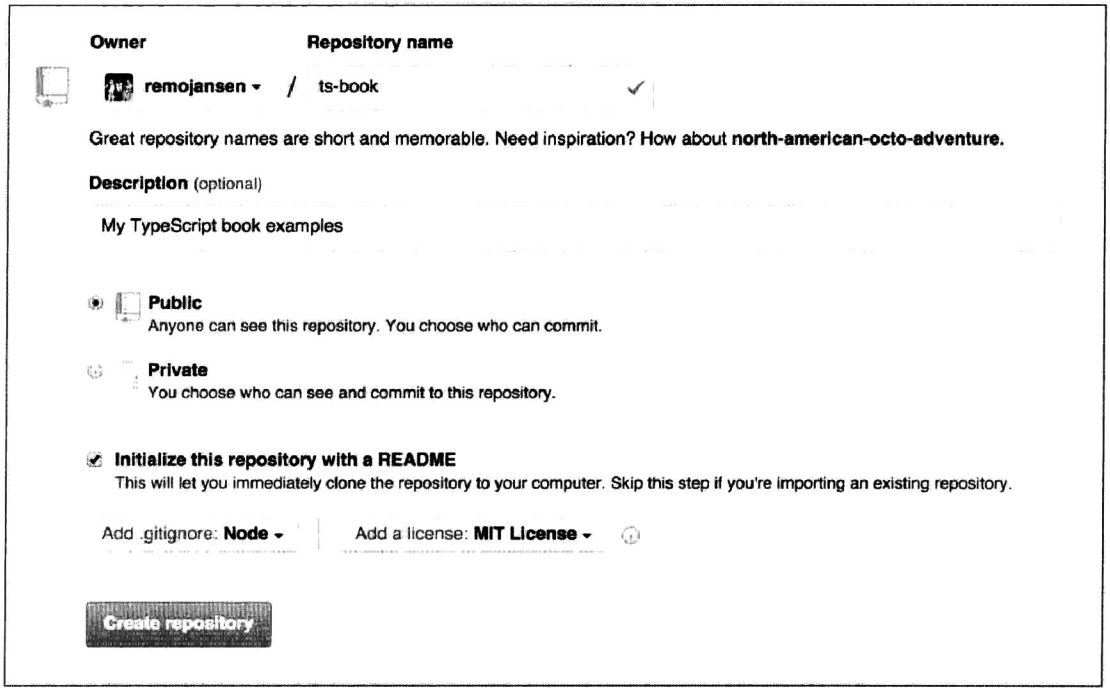
现在，已经成功安装了 Git，并注册好 GitHub 账号，我们将在 GitHub 里创建一个新的代码仓库。一个代码仓库即是一个存储源代码的地方，它被版本控制工具用于管理不同版本的代码。虽然单一用户在一台计算机上也可以创建一个仓库，但它常常会被放于服务器上，这样可以被许多用户所访问。

GitHub 为开源项目提供了免费的版本控制。在开源社区，GitHub 已经十分流行，不少知名开源项目已托管在了 GitHub 上（包括 TypeScript）。但是，GitHub 并不是唯一的选择，你仍可以使用本地 Git 仓库，或其他版本控制服务，如 Bitbucket。如果你想了解更多，可以访问 Git 的官方文档：`https://git- scm.com/doc`，或 BitBucket 的官方网站：`https://bitbucket.org/`。

为了在 GitHub 上创建一个新的仓库，首先需要登录 GitHub，并且单击页面右上角的加号按钮。



然后将会出现一个和下图类似的表单。可以在里面填入如仓库名、仓库描述、隐私策略等仓库信息。



可以为仓库添加一个 `README.md` 文件，它使用 markdown 语法，用于添加想在仓库首页展示的信息。另外，还可以添加一个默认的 `.gitignore` 文件，它用于指定我们想让 Git 忽略的文件，这些文件将不会被保存在仓库中。

最后，需要为源代码选择一个开源证书。在创建好仓库之后，进入仓库首页，在页面的右下角找到用于克隆该仓库的 URL。

复制这个 URL，打开控制台，将这个 URL 作为 `git clone` 命令的参数输入：

```
git clone https://github.com/user-name/repository-name.git
```

有时，在 Windows 的命令行界面中找不到 Git 和 Node.js 命令。
最简单的解决办法是使用 Git 控制台（在安装时 Git 会附带安装）来替代 Windows 命令行。
如果仍想使用 Windows 控制台，你需要手动地将 Git 和 Node.js 安装目录添加到 Windows 的 PATH 环境变量中。
另外，我们所有的例子都将使用 UNIX 路径语法。
如果你的操作系统是 OS X 或 Linux，那么使用自带的默认命令行界面是没有问题的。

该命令将会有类似下面的输出：

```
Cloning into 'repository-name'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

然后，可以使用变更目录命令（cd）进入仓库，接着使用 `git status` 命令检查本地仓库的状态：

```
cd repository-name
```

```
git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

在本书接下来的部分中，我们都将使用 GitHub 来管理源代码。如果你只想使用本地仓库，那么可以使用 Git 的 `init` 命令创建一个空的本地仓库。更多关于 `git init` 命令和使用本地仓库的信息，请参阅 Git 文档：`http://git-scm.com/docs/git-init`。

`git status` 命令的输出告诉我们，当前的工作目录并没有任何改变。在 Atom 中打开仓库目录，然后创建一个 `gulpfile.js` 文件。现在，再次运行 `git status` 命令，会看到一些新的未追踪的文件：

```
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  gulpfile.js

nothing added to commit but untracked files present (use "git add" to
track)
```

不同状态的文件，在 Atom 中会以不同的颜色显示。我们可以非常轻易地分辨出，哪些文件是新创建的，哪些是在克隆完仓库后有过修改的。

当做了一些修改后（如创建新文件或修改已有文件），需要执行 `git add` 命令来表明我们希望保存这些修改：

```
git add gulpfile.js
```

```
git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  new file:   gulpfile.js
```

这样一来，就已经指明了我们想要保存的内容，需要执行 `git commit` 来真正地将它们保存下来。我们需要使用`-m`参数来对修改添加备注信息：

```
git commit -m "added the new gulpfile.js"
```

如果一切正常，那么将会有类似以下的输出：

```
[master 2a62321] added the new file gulpfile.js
1 file changed, 1 insertions(+)
create mode 100644 gulpfile.js
```

如果想与其他开发者共享此次提交，需要将代码推送至远程仓库，可通过 `git push` 命令进行：

```
git push
```

`git push` 命令将会要求输入 GitHub 的用户名和密码，然后修改后的代码就被推送至远程仓库。如果我们进入 GitHub 访问仓库首页，那么将能看到新创建的那个文件。在后面关于持续集成服务的章节里，我们将继续回到 GitHub 上。

如果你在一个大型团队中从事开发工作，那么在试图推送代码至远程仓库的过程中，可能会遇到代码冲突。如何解决代码冲突已经超出本书的讨论范围。可以从 `https://www.kernel.org/pub/software/scm/git/docs/user-manual.html` 获取用户手册，以便得到更详细的指导。

包管理工具

包管理工具用于管理应用的依赖。我们将会学习三种包管理工具：Bower、npm 和 tsd。

npm

npm 最初被设计为 Node.js 的默认包管理工具，但如今它已经有了更广泛的用途。npm 使用一个名为 `package.json` 的文件作为配置文件，来管理应用的依赖。在安装服务端程序、桌面应用或开发工具的依赖时，我们往往选择 npm。

在安装依赖前，需要为项目添加一个 `package.json` 文件。可以通过以下命令做到：

```
npm init
```

`npm init` 命令将会询问我们一些有关项目的基本信息，包括项目名、版本号、简述、入口文件、测试命令、Git 仓库、关键字、作者名和证书类型。

如果你对上述 `package.json` 文件中的属性有任何不明确的地方，可以参阅 npm 官方文档：`https://docs.npmjs.com/files/package.json`。

接着 `npm` 命令会展示即将生成的 `package.json` 文件的预览，并寻求我们的最终确认。

注意，在使用 `npm` 命令之前，必须确保已经安装了 Node.js。

创建好 `package.json` 文件后，就可以使用 `npm install` 命令来安装依赖了。`npm install` 命令的第一个参数为以空格分隔的一个或多个依赖名，第二个参数则是安装范围。

安装范围可以是：

- 开发时的依赖（测试框架、编译器等）。
- 运行时的依赖（Web 框架、数据库 ORM 等）。

我们将使用 `gulp-typescript` 包来编译 TypeScript 代码。所以，我们将其作为开发时的依赖（使用`--save-dev` 参数）进行安装。

```
npm install gulp-typescript --save-dev
```

如果要全局安装一个依赖，可以使用`-g` 参数：

```
npm install typescript -g
```

在开发环境中进行全局安装时，可能会需要管理者权限。
另外还需要注意的是，全局安装的包并不会出现在 package.json 中。但在持续集成时，我们需要确保应用中所有的依赖都被安装，所以需要手动将全局安装的包添加进 package.json 的 devDependencies 或 peerDependencies 中。我们将会在本章后面学习持续集成。

运行时的依赖，需要使用--save 参数来进行安装：

```
npm install jquery --save
```

jQuery 可能是史上最流行的 JavaScript 库，它简化了不少浏览器端的 API，并且让使用者在调用时无须考虑跨浏览器兼容问题。同时，它也简化了我们选择 HTML 文档的节点树中节点的方式。
我们假设本书的读者已经对 jQuery 有了充分的了解。如果你想阅读更多关于 jQuery 的资料，请参考它的官方文档：https://api.jquery.com/。

在安装完一些依赖后，我们的 package.json 文件看上去像下面这样：

```
{
  "name": "repository-name",
  "version": "1.0.0",
  "description": "example",
  "main": "index.html",
  "scripts": {
    "test": "test"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/username/repository-name.git"
  },
  "keywords": [
    "typescript",
    "demo",
```

```
    "example"
  ],
  "author": "Name Surname",
  "contributors": [],
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/username/repository-name/issues"
  },
  "homepage": "https://github.com/username/repository-name",
  "engines": {},
  "dependencies": {
    "jquery" : "^2.1.4"
  },
  "devDependencies": {
    "gulp-typescript": "^2.8.0"
  }
}
```

package.json 中的某些字段必须手动进行配置。更多 package.json 中可供配置的字段信息，请参阅 https://docs.npmjs.com/files/package.json。

在本书出版发行时，本书中使用的各个 npm 包的版本可能都有所更新，请到 https://npmjs.com 搜索各个包的文档，查询它们的新特性和潜在的不兼容改变。

所有的 npm 包都将保存在 node_modules 目录下。推荐将 node_modules 目录加入我们的.gitignore 文件中，这样一来，版本控制系统就会忽略它。只需打开.gitignore 文件，并在新的一行中写入目录名 node_modules 就可办到。

下一次克隆仓库到本地时，需要重新安装应用的依赖。不需添加任何参数，执行 npm install 命令即可：

```
npm install
```

npm 会查找 package.json 中记录的依赖列表，并全部安装。

如果我们需要知道一个 npm 包的名字，可以访问 https://www.npmjs.com 并进行搜索。

Bower

Bower 也是一个包管理工具。它与 npm 非常类似，只是它通常只用于管理前端的依赖。因此，Bower 中的许多包也都针对前端做过优化。

我们可以使用 npm 来安装 Bower：

```
npm install -g bower
```

Bower 使用的配置文件名为 bower.json，而不是 package.json。我们在 npm 中使用的大多数参数都可以在 Bower 中使用，例如，可以通过 bower init 命令来初始化一个 Bower 配置文件：

```
bower init
```

初始化后的 Bower 配置文件与 package.json 十分类似。如果你想更多地了解 bower.json 配置文件，请参阅官方文档：http://bower.io/docs/ config/。

同样也可以使用 bower install 命令来安装一个包：

```
bower install jquery
```

指定安装范围的参数也是相同的：

```
bower install jquery --save
bower install jasmine --save-dev
```

所有的 Bower 包都保存在 bower_components 目录下。和 node_modules 一样，我们同样也推荐你将该目录加入.gitignore 文件中，来避免它被推送至远程仓库。

tsd

在上一章里，我们了解到 TypeScript 默认包含了一个 lib.d.ts 文件，里面描述了

JavaScript 内建对象、文档对象模型（DOM）和浏览器对象模型（BOM）的 API。一个扩展名为.d.ts 的文件是一种特殊的 TypeScript 文件，我们称它们为类型定义文件或描述文件。

一个类型描述文件通常包含对第三方库 API 的类型声明。这些库使现存的 JavaScript 库能与 TypeScript 集成在一起。例如，如果在 TypeScript 文件中调用 jQuery，我们会得到一个报错：

```
$.ajax({ / **/ }); // cannot find name '$'
```

为了解决这个问题，需要在 TypeScript 代码中添加 jQuery 描述文件的引用。就像下面的例子那样：

```
///<reference path="jquery.d.ts">
```

幸运的是，我们不必亲自写一个 jQuery 的描述文件，因为有一个名为 DefinitelyTyped 的开源项目已经包含了众多 JavaScript 库的描述文件。在 TypeScript 出现的早期，开发者们需要手动从 DefinitelyTyped 项目的网站上下载并安装声明文件，但是如今，我们已经有了 tsd。

tsd 这个名字源于 TypeScript Definitions 的首字母缩写，它是一个管理 TypeScript 应用中描述文件的工具。和 npm 和 bower 一样，它也有自己的配置文件，名为 tsd.json，并且将所有下载下来的描述文件都保存在 typings 目录下。

执行以下命令来安装 tsd：

```
npm install tsd -g
```

我们使用 tsd init 命令来初始化 tsd.json 配置文件，并且使用 tsd install 命令来下载和安装依赖：

```
tsd init // 生成tsd.json
tsd install jquery --save // 安装jQuery描述文件
```

你可以访问 DefinitelyTyped 项目的网站：https://github.com/borisyankov/DefinitelyTyped，来搜索 tsd 包。

自动化任务工具

自动化任务工具用来自动化地执行开发过程中需要重复进行的任务。这些任务包括编

译 TypeScript 文件、压缩 JavaScript 文件，等等。如今，最流行的两个 JavaScript 自动化任务工具分别是 Grunt 和 Gulp。

Grunt 在 2012 年的上半年开始流行，此后，开源社区开发了大量兼容 Grunt 的插件。

而 Gulp 的流行则始于 2013 年的下半年。因此，Gulp 的插件数量略少于 Grunt，但它目前大有赶超的趋势。

除了可用插件数量上的差别，Gulp 和 Grunt 的主要差别在于，在 Grunt 中，我们使用文件作为任务的输入和输出，而在 Gulp 中，我们使用的是流。Grunt 插件大多使用键值对来进行配置，Gulp 插件则更倾向于使用代码来描述任务，而不是配置，这使得 Gulp 任务的可读性通常比 Grunt 任务来得高。

在本书中，我们将使用 Gulp。如果你想更多地学习 Grunt，可以参阅 http://gruntjs.com/。

为了能够更好地理解 Gulp，我们将使用一个现成的项目，然后为其添加一些额外的文件夹和文件。我们将会创建一些会多次使用项目中的路径、文件夹和文件的任务。项目的目录结构如下所示：

```
├── LICENSE
├── README.md
├── index.html
├── gulpfile.js
├── karma.conf.js
├── tsd.json
├── package.json
├── bower.json
├── source
|   └── ts
|       └── *.ts
├── test
|   └── main.test.ts
├── data
|   └── *.json
├── node_modules
|   └── ...
├── bower_components
```

```
|       └── ...
└── typings
        └── ...
```

本书的源代码中提供了一份该示例项目的最终版副本，它可以帮助我们学习下文中的例子。你还可以通过它来加深对下文中即将提到的概念的理解。

让我们先使用 npm 全局安装 `gulp`：

```
npm install -g gulp
```

然后再把它作为开发时依赖进行安装：

```
npm install --save-dev gulp
```

接着在项目的根目录创建一个名为 `gulpfile.js` 的 JavaScript 文件，并添加如下代码：

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

gulp.task('default', function() {
  console.log('Hello Gulp!');
});
```

最后，执行 `gulp` 命令（必须和 `gulpfile.js` 在同一目录下）：

```
gulp
```

我们已经成功创建了第一个 Gulp 任务，名为 `default`。当在命令行执行 `gulp` 命令后，它将在所有当前目录下搜索 `gulpfile.js`，找到后执行 `default` 任务。

检查 TypeScript 代码的质量

之前例子中的 `default` 任务并没有做任何实事，通常在每个任务中，都会使用一个 Gulp 插件。现在来添加我们的第二个任务，它将使用到 `gulp-tslint` 插件，用来检查我们的 TypeScript 代码格式是否符合一系列推荐的实践。

首先通过 npm 安装这个插件：

```
npm install gulp-tslint --save-dev
```

然后在 gulpfile.js 中添加一个新任务：

```
var tslint = require('gulp-tslint');
gulp.task('lint', function() {
  return gulp.src([
    './source/ts/**/**.ts', './test/**/**.test.ts'
  ]).pipe(tslint())
    .pipe(tslint.report('verbose'));
});
```

我们创建了一个名为 lint 的新任务。下面来分析一下这个任务所有的执行步骤：

1. Gulp 的 src 函数将会查找./source/ts 目录及其子目录下所有扩展名为.ts 的文件。同样它还会查找./test 目录及其子目录下所有扩展名为.test.ts 的文件。
2. src 函数的输出流将会被 pipe 函数重定向为 tslint 函数的输入。
3. 最后，我们将 tslint 函数的输出重定向为 tslint.report 函数的输入。

现在，lint 任务已经添加完毕。我们将要修改 gulpfile.js 文件，来指明 lint 任务为 default 任务的一个子任务：

```
gulp.task('default', ['lint']);
```

许多插件允许我们在路径前添加一个英文感叹号（！）来忽略一些文件。例如，!path/*.d.ts 会忽略 path 目录下所有扩展名为.d.ts 的文件。这个语法在源文件和目标文件都在同一目录下时，十分有用。

编译 TypeScript 代码

现在我们将添加两个编译 TypeScript 代码的任务（一个编译应用的逻辑的代码，另一个编译单元测试的代码）。

我们将使用 gulp-typescript 插件。和上面的例子一样，将其作为开发时依赖安装：

```
npm install gulp-typescript --save-dev
```

然后创建一个新的 gulp-typescript 项目对象：

```
var ts = require('gulp-typescript');
```

```
var tsProject = ts.createProject({
  removeComments : true,
  noImplicitAny : true,
  target : 'ES3',
  module : 'commonjs',
  declarationFiles : false
});
```

gulp-typescript 插件已经宣布它支持通过一个名为 tsconfig.json 的特殊 JSON 文件进行配置。这个文件用于保存 TypeScript 编译器的配置参数，这样我们就能够在编译过程中使用它了。

当在控制台界面里编译 TypeScript 代码时，tsconfig.json 可以让我们不必每一次都必须输入所有的编译参数。关于这个特性的更多信息，请参阅 gulp-typescript 文档：https://www.npmjs.com/package/gulp-typescript。

在上面的例子中，我们将 TypeScript 编译器作为依赖加载进来，然后创建了一个名为 tsProject 的对象，它包含了 TypeScript 编译器在编译我们的代码时需要带上的参数。现在来编译应用的源代码：

```
gulp.task('tsc', function() {
return gulp.src('./source/ts/**/**.ts')
           .pipe(ts(tsProject))
           .js.pipe(gulp.dest('./temp/source/js'));
});
```

tsc 任务会去查找./source/ts 目录及其子目录下所有的.ts 文件，然后将它们作为一个输入流传递给 TypeScript 编译器。编译器将会使用 tsProject 对象的属性作为编译时的参数，并将编译后的 JavaScript 代码保存在./temp/sources/js 目录下。

我们还需要编译项目里用 TypeScript 书写的单元测试。它们都在 test 目录下，我们想让编译后的 JavaScript 文件保存在 temp/test 目录下。对于不同的任务，不同的输入文件，使用同一个配置对象并不是一个推荐的做法。所以我们需要初始化另一个 gulp-typescript 项目对象，将其命名为 tsTestProject：

```
    var tsTestProject = ts.createProject({
        removeComments : true,
        noImplicitAny : true,
        target : 'ES3',
        module : 'commonjs',
        declarationFiles : false
    });
```

tsc-test 任务和 tsc 任务非常类似，不过它们编译的文件不同。由于应用源代码和测试代码在不同的目录下，所以我们在任务里使用不同的路径：

```
gulp.task('tsc-tests', function() {
  return gulp.src('./test/**/**.test.ts')
             .pipe(ts(tsTestProject ))
             .js.pipe(gulp.dest('./temp/test/'));
});
```

再次修改 default 任务，让它包含我们的新任务：

```
gulp.task('default', ['lint', 'tsc', 'tsc-tests']);
```

优化 TypeScript 应用

当编译完 TypeScript 代码后，编译器会为每个 TypeScript 文件生成对应的 JavaScript 文件。但是它们现在还不能运行在浏览器中，因为在浏览器中运行 JavaScript 的唯一方法是，为每一个 JavaScript 文件都在 HTML 中添加 <script> 标签来引入它。

或者，还有以下两种方案：

- 可以使用如 RequireJS 这样的库，通过 AJAX 来按需加载各个文件。这种做法被称为*异步模块加载*。我们需要将 TypeScript 编译器的模块参数配置成使用*异步模块定义*（AMD）。
- 还可以将 TypeScript 的模块参数配置成 CommonJS，然后使用诸如 Browserify 这样的工具，解析应用的模块和依赖，最后生成一个包含应用里的所有模块且高度优化的 JavaScript 文件。

在本书中，我们将使用 CommonJS 方案，因为 Browserify 和 Gulp 是高度集成的。

如果此前你没有了解过 AMD 或 CommonJS，不必担心，我们将会在第 4 章中详细讨论它们。

在我们的应用的根模块（例子中是 `main.ts`）里，包含如下代码：

```
///<reference path="./references.d.ts" />

import { headerView }  from './header_view';
import { footerView }  from './footer_view';
import { loadingView } from './loading_view';

headerView.render();
footerView.render();
loadingView.render();
```

上述例子中的 `import` 声明用于访问一些外部模块中的内容。我们将在第 4 章学习外部模块。

编译完毕后（配置为使用 CommonJS），输出的 JavaScript 代码会是这样的：

```
var headerView = require('./header_view');
var footerView = require('./footer_view');
var loadingView = require('./loading_view');
headerView.render();
footerView.render();
loadingView.render();
```

正如代码的前三行所描述的，`main.js` 文件依赖于 `header_view.js`、`footer_view.js` 和 `loading_view.js` 这三个 JavaScript 文件。如果继续去查看这三个文件，会发现这些文件又有它们各自的依赖。

我们将这些依赖视为模块。导入一个模块让我们能够在另一个模块里使用此模块的公开部分（或称为导出部分）。

Browserify 将会追踪这三个依赖各自的依赖树，然后生成一个包含应用中所有依赖的高度优化的文件。

现在，我们将在 gulpfile.js 中添加两个新的任务。在第一个任务里，我们将配置 Browserify 来追踪应用中所有模块的依赖。而在第二个任务里，我们则配置 Browserify 来追踪应用里所有单元测试的依赖。

在实现这些任务之前，需要先安装一些包：

```
npm install browserify vinyl-transform  gulp-uglify gulp-sourcemaps
```

导入这些包，然后初始化它们：

```
var browserify    = require('browserify'),
    transform     = require('vinyl-transform'),
    uglify        = require('gulp-uglify'),
    sourcemaps    = require('gulp-sourcemaps');
var browserified  = transform(function(filename) {
  var b = browserify({ entries: filename, debug: true });
  return b.bundle();
});
```

在上述例子里，我们已经加载了必要的插件，然后由于兼容性问题，声明了一个名为 browserified 的函数。browserified 函数将会把一个普通的 Node.js 流转换为一个 Gulp 流。

现在我们来实现实际的任务：

```
gulp.task('bundle-js', function () {
  return gulp.src('./temp/source/js/main.js')
    .pipe(browserified)
    .pipe(sourcemaps.init({ loadMaps: true }))
    .pipe(uglify())
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('./dist/source/js/'));
});
```

在这个任务中，我们只需将 main.js 作为入口，Browserify 会从这个入口出发，追踪应用里所有的模块和依赖，然后生成一个包含了高度优化的 JavaScript 文件的流。

接着我们会使用 uglify 插件来最小化输出。这将会减少应用的加载时间，但这也会让 bug 难以被追踪。我们将会生成一个 source map 文件来简化追踪 bug 的过程。

uglify 将会移除代码里所有的空行和空格，并且缩减变量名的长度。Source map 使得我们在追踪 bug 时能够将最小化后的代码映射到源文件中对应的地方。

正由于这样，我们可以轻松地追踪一个最小化后的代码文件中的 bug。在 Chrome 和 Firefox 浏览器的开发者工具里，已经内建了对 source map 的支持。

bundle-test 与之前所写的十分类似。这次，我们没有使用 uglify 和 source map，因为不需要优化单元测试代码的下载时间。正如你所看到的，这次我们允许多个入口的存在（每一个入口都将与一组称为测试套件的自动化测试关联，如果你还不熟悉这个概念，不要担心，我们将在第 7 章中学习）：

```
gulp.task('bundle-test', function () {
  return gulp.src('./temp/test/**/**.test.js')
    .pipe(browserified)
    .pipe(gulp.dest('./dist/test/'));
});
```

最后，在 default 任务里添加上我们的新任务：

```
gulp.task('default', ['lint', 'tsc', 'tsc-tests', 'bundle-js',
'bundle-test']);
```

我们已经创建了一个把 TypeScript 文件编译为 JavaScript 文件的任务。这些 JavaScript 文件保存在一个临时文件夹里，接着后面的任务把这些临时文件夹下的 JavaScript 文件打包成一个文件。在实际的环境中，使用临时文件夹并不是一个推荐的做法。应该直接把编译的输出流作为打包的输入流。但是在本书中，为了更好地让读者理解每一个任务，我们选择了将这些步骤分开。

如果现在尝试去执行 default 任务，那么可能会遇到一些问题，因为它里面的任务现在是并行执行的。下面我们将学习如何控制任务的执行顺序，来避免这些问题。

管理 Gulp 任务的执行顺序

有些时候，需要以特定的顺序来执行我们的任务（例如，在执行单元测试之前，需要先将 TypeScript 代码编译成 JavaScript 代码）。由于 Gulp 默认是异步执行所有任务的，所以控制任务的执行顺序非常有挑战性。

有三种让一个任务同步执行的办法：

- 传递一个回调函数
- 返回一个流
- 返回一个 promise

在第 3 章中，我们将深入学习回调函数和 promise 的用法。

让我们先看看前两种办法（本章中将不会提到 promise）：

```
// 传递一个回调函数
gulp.task('sync', function (cb) { // 注意cb参数
  // setTimeout可以是任意的异步函数
  setTimeout(function () {
    cb(); // cb在这里执行
  }, 1000);
});

// 返回一个流
gulp.task('sync', function () {
  return gulp.src('js/*.js') // 注意return关键字在此处
    .pipe(concat('script.min.js')
    .pipe(uglify())
    .pipe(gulp.dest('../dist/js');
});
```

现在已经有了一个同步的任务，将其加入任务依赖，来管理执行顺序：

```
gulp.task('secondTask', ['sync'], function () {
  // 这个任务在sync任务完成前都不会执行
});
```

在上述代码里，secondTask 任务在 sync 任务完成前都不会执行。假设现在有第三个名为 thirdTask 的任务，我们希望 default 任务在 sync 和 thirdTask 这两个任务都完成后再执行，并且 sync 和 thirdTask 这两个任务是并行执行的：

```
gulp.task('default', ['sync', 'thirdTask'], function () {
  // 在这里执行一些任务
});
```

幸运的是，可以通过 npm 安装一个名为 run-sequence 的 Gulp 插件，它将使我们能更好地控制任务执行顺序：

```
var runSequence = require('run-sequence');
gulp.task('default', function(cb) {
  runSequence(
    'lint',                      // lint
    ['tsc', 'tsc-tests'],        // compile
    ['bundle-js','bundle-test'], // optimize
    'karma'                      // test
    'browser-sync',              // serve
    cb                           // callback
  );
});
```

上述代码中的任务将会有以下的执行顺序：

1. lint。
2. 并行执行 tsc 和 tsc-tests。
3. 并行执行 bundle-js 和 bundle-test。
4. karma。
5. browser-sync。

在本书即将出版前，Gulp 的开发团队宣布了改进任务执行顺序控制的计划，而不再需要依赖额外的插件。更多关于这个特性的说明，请参阅 Gulp 文档：https://github.com/gulpjs/gulp/blob/master/docs/README.md。

自动化测试工具

自动化测试工具让我们能够自动化地执行应用里的单元测试。

单元测试是指针对代码中的某个函数或某个部分（单元）进行的测试。通过单元测试，可以保证函数按照预期在工作。这里假设读者已经对单元测试的过程有所了解，不过，更多关于单元测试的高级话题我们将在第 7 章中讨论。

通过使用自动化测试工具，可以自动在多个浏览器内执行应用的测试套件，而不必手动打开浏览器运行测试。

我们将使用的自动化测试工具叫作 Karma。Karma 可以和多个流行的单元测试框架兼容，本书中使用的测试框架为 Mocha，并且会连同另外两个库一同使用：Chai（一个断言库）和 Sinon（一个数据模拟框架）。

如果对这些库还不是非常熟悉，不必担心，我们将在第 7 章继续学习它们。

使用 npm 来安装这些将要使用的测试框架：

```
npm install mocha chai sinon --save-dev
```

然后把 Karma 作为开发时依赖安装：

```
npm install karma karma-mocha karma-chai karma-sinon karma-coverage
karma-phantomjs-launcher gulp-karma --save-dev
```

在安装完所有需要的包后，在 `gulpfile.js` 中添加一个新任务。这个任务将会使用 Karma 运行应用的单元测试：

```
var karma = require("gulp-karma");
gulp.task('karma', function(cb) {
  gulp.src('./dist/test/**/**.test.js')
    .pipe(karma({
      configFile: 'karma.conf.js',
      action: 'run'
```

```
    }))
    .on('end', cb)
    .on('error', function(err) {
      // 确保测试失败后让gulp以非0的状态码退出
      throw err;
    });
});
```

在上述例子里，我们获取了./dist/test/及其子目录下所有扩展名为.test.js 的文件，然后把它们连同 karma.conf.js（包含了 Karma 配置信息）文件一同传递给了 Karma 插件。我们需要在根目录中创建一个名为 karma.conf.js 的文件，然后把以下代码复制进来：

```
module.exports = function (config) {
  'use strict';
  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
    plugins : [
      'karma-coverage',
      'karma-mocha',
      'karma-chai',
      'karma-sinon',
      'karma-phantomjs-launcher'
    ],
    preprocessors: {
      './dist/test/*.test.js' : ['coverage']
    },
    port: 9876,
    colors: true,
    autoWatch: false,
    singleRun: false,
    logLevel: config.LOG_INFO
  });
};
```

这个配置文件告诉了 Karma 应用的根目录、框架（Mocha、Chai 和 Sinon.js）、浏览器

（PhantomJS）、插件和需要报告的测试执行期间的信息。PhantomJS 是一个无界面的 Web 浏览器，使用它就无须打开一个真正的 Web 浏览器，而可以执行单元测试代码。

在将应用部署到生产环境之前，我们还需要在真实的浏览器中运行这些测试。不过，已有一些如 `karma-firefox-launcher` 和 `karma-chrome-launcher` 这样的插件可让我们在特定浏览器中运行测试。

Karma 默认会报告一些在测试执行期间的状态。我们将测试覆盖率也加入了报告，因为想要知道应用中到底有百分之几的代码被测试到。在加上覆盖率报告后，运行我们的单元测试，在 `karma.conf.js` 所处的同一目录下就能看到一个 `coverage` 文件夹。

如果你在 `http://karma-runner.github.io/0.8/config/configuration-file.html` 上查看 Karma 配置文档，将会注意到，我们并没有在 `karma.conf.js` 里添加 files 字段。我们没有用这个属性来指明单元测试文件的位置，是因为 Gulp 任务已经把它作为输入流传递给了 Karma。

使跨设备测试同步

现在在 `gulpfile.js` 中添加最后一个任务，用于在浏览器中运行我们的应用。我们先通过 npm 安装 `browser-sync` 包：

```
npm install -g browser-sync
```

我们先创建两个任务，这两个任务的主要用途是将多个任务合并成一个主任务。这么做主要是因为，很多时候，在修改了部分 TypeScript 代码后，总是要刷新浏览器来看看效果，并且在看到实际效果前，需要先执行一系列任务（编译、打包等）。在把这一系列任务打包成一个任务后，可以使我们的配置文件可读性更高：

```
gulp.task('bundle', function(cb) {
  runSequence('build', [
    'bundle-js', 'bundle-test'
  ], cb);
});
gulp.task('test', function(cb) {
```

```
  runSequence('bundle', ['karma'], cb);
});
```

上面例子中的两个任务用于将所有的任务合并成一个主任务（名为 bundle），以及把所有测试相关的任务合并成一个主任务（名为 test）。

在安装完所有包并且实现了上述两个任务后，我们为 gulpfile.js 添加一个新任务：

```
var browserSync = require('browser-sync');
gulp.task('browser-sync', ['test'], function() {
  browserSync({
    server: {
      baseDir: "./dist"
    }
    });
  return gulp.watch([
    "./dist/source/js/**/*.js",
    "./dist/source/css/**.css",
    "./dist/test/**/**.test.js",
    "./dist/data/**/**",
    "./index.html"
  ], [browserSync.reload]);
});
```

在上面的任务里，我们在 BrowserSync 里把应用的静态文件目录配置为./dist。然后为这个目录添加了一个监视任务，一旦这个目录下的文件有所改变，那么 BrowserSync 就会自动刷新浏览器。

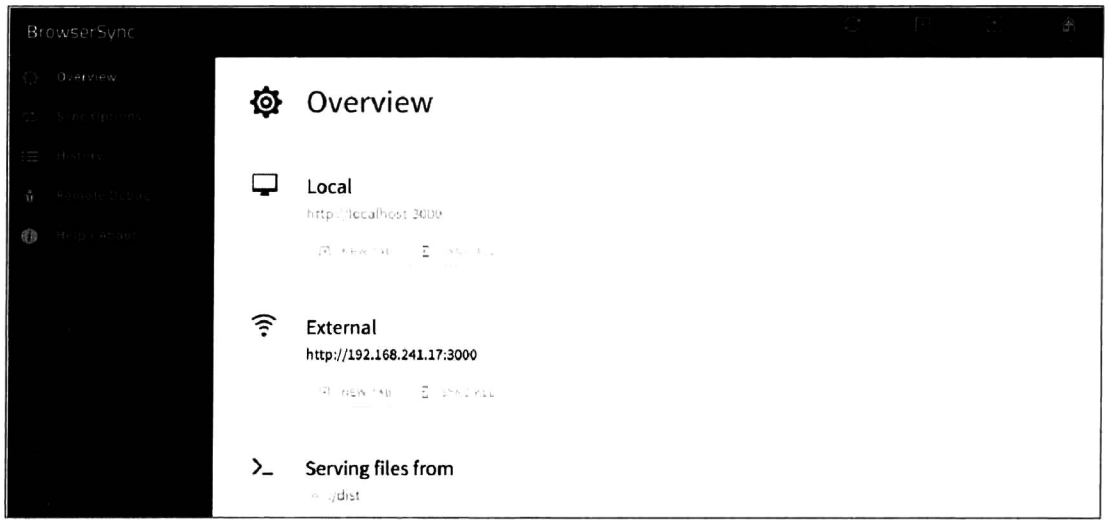
当代码有变化时，test 任务就会自动执行。然后因为 test 任务会调用 bundle 任务，所以在自动刷新浏览器之前，所有的任务（test 和 bundle）都会执行。

BrowserSync 着实是一个非常强大的工具，它能够使我们在一台设备上执行测试，然后自动在其他多台设备上重复一系列动作（单击、滑动，等等）。它让我们能够远程调试应用，这在移动端设备上调试应用时十分有用。

跨设备同步其实非常简单。如果我们执行了 browser-sync 任务，那么它就会替我们在系统的默认浏览器中打开应用。如果我们观察控制台里的输出，能看到应用已经运行在了一个地址（http://localhost:3000）上，而 BrowserSync 工具则已经在另一个地址（http://localhost:3001）上可用：

```
[BS] Access URLs:
------------------------------------
       Local: http://localhost:3000
    External: http://192.168.241.17:3000
------------------------------------
         UI: http://localhost:3001
  UI External: http://192.168.241.17:3001
------------------------------------
[BS] Serving files from: ./dist
```

如果我们在浏览器的一个选项卡中打开 BrowserSync 工具的 URL（例子里为 http://localhost:3001），就能看到如下界面：



可使用 BrowserSync 工具的界面配置远程调试参数和设备同步参数。如果要同步一个新设备，只需将我们的手机或平板电脑连接到本地的网络环境，然后在设备的浏览器中打开外部 URL 即可。

如果你想更深入地学习 BrowserSync，请参阅它的官方文档：http://www.browsersync.io/docs/。

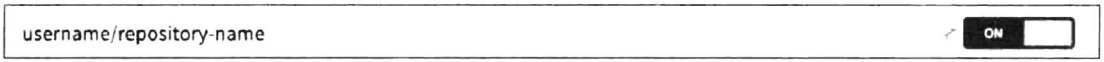
持续集成工具

持续集成工具（CI）是一种帮助我们避免潜在的代码集成问题的开发实践。在多人开发时，我们往往需要把多个不同人开发的子系统进行合并。在这个把每个人各自经过测试的代码合并在一起的过程中，往往会出现软件集成方面的问题。

在使用“瀑布”式开发或“大爆炸”式开发时，组件代码在实现完毕并经过测试后会被合并。与它们不同的是，持续集成要求开发者每天都向远程代码仓库提交代码。每次提交，系统都会进行自动构建，这使得集成问题会尽早被团队发现。

在本章里，我们已经创建了一个远程代码仓库和它的自动化构建流程，但是还没有配置一个监控我们的提交，并且自动运行自动化构建流程的工具。我们需要一个持续集成服务器。市面上有许多可选的持续集成服务器提供商，但是仔细分析这些选择的优劣已经超出了本书的范围。在本书中，我们将使用 Travis CI，因为它与 GitHub 高度集成，并且对开源项目免费。

为了能够配置 Travis CI，我们需要使用自己的 GitHub 身份来登录 `https://travis-ci.org`。在登录后，将可以看到自己 GitHub 中的公开项目列表，如下图所示，并且能够让其启动持续集成功能。



为了完成配置，我们需要在应用的根目录创建一个名为 `travis.yml` 的文件，它包含了 Travis CI 配置：

```
language: node_js
node_js:
- "0.10"
```

TravisCI 的配置文件中还有很多可选的属性，详情请参阅 `http://docs.travis-ci.com/`。

在完成了这些简单的配置步骤后，Travis CI 将会开始监控我们对远程仓库的每一次提交。

如果在本地开发环境中应用可以成功构建，但是在持续集成服务器中失败的话，我们就需要去查阅构建日志，并找出到底是哪里出了问题。有可能是我们本地的软件版本高于持续集成环境中的版本，或者持续集成环境中安装的依赖不全。关于如何解决这些问题，请参阅 Travis CI 文档：`http://docs.travis-ci.com/user/build-configuration/`。

脚手架工具

脚手架工具可以用来自动生成项目的文件结构、构建脚本，等等。如今最流行的脚手架工具是 Yeoman。Yeoman 使用的内建命令为 `yo`，同时它自己也是一个包管理工具和自动化任务工具，它基于模板生成项目。

生成项目的模板在 Yeoman 中被称为生成器，在开源社区中，人们已经发布了许许多多的生成器，所以我们总能找到符合要求的生成器。或者，也可以自己写一个 Yeoman 生成器，然后发布。

我们将展示如何用 Yeoman 来创建一个新项目，以节省时间。Yeoman 将会替我们创建 `package.json` 和 `bower.json` 文件，然后自动安装依赖。

`yo` 命令可以通过 npm 安装：

```
npm install -g yo
```

在安装完毕后，至少需要安装一个生成器。我们需要根据项目的需求，找到一个合适的生成器。

我们的新项目将使用 TypeScript 和 Gulp，所以可以使用一个名为 `generator-typescript` 的生成器。所有可用的生成器都可以在 `http://yeoman.io/generators/` 上找到。

可以通过 npm 安装一个生成器：

```
npm install -g generator-typescript
```

安装完毕后，我们需要结合 `yo` 命令一同使用：

```
yo typescript
```

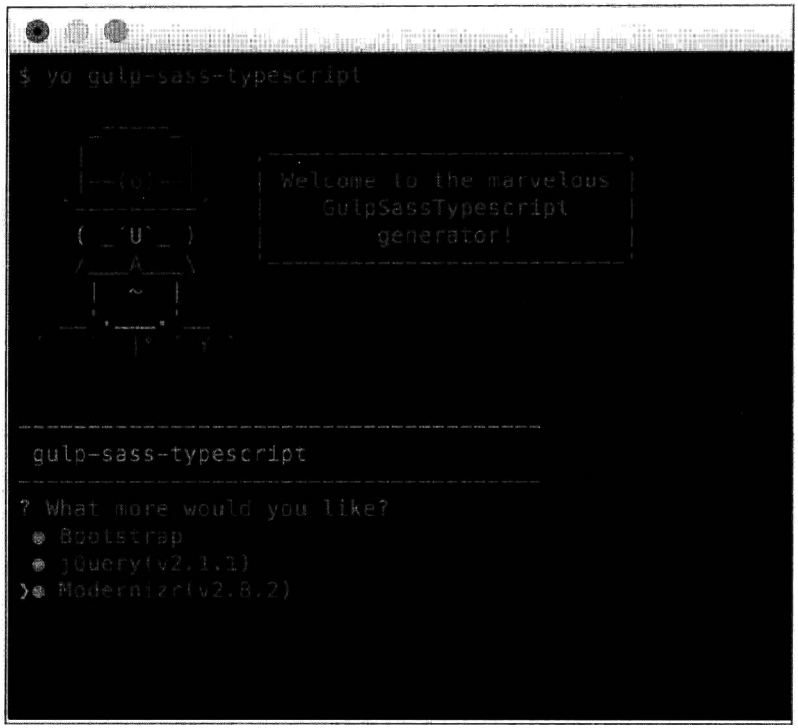
如果想在项目中使用 Sass，那么可以使用 `generator-gulp-sass-typescript` 生成器替代：

```
npm install -g generator-gulp-sass-typescript
```

很多生成器在运行时都是交互式的，允许我们为项目选择一些可选的第三方库。让我们运行一下这个生成器：

```
yo generator-gulp-sass-typescript
```

屏幕上将会出现一系列引导，一步步地帮助我们创建新项目，包括选择 Gulp 作为自动化任务工具，Sass 作为 CSS 预处理器和 TypeScript 作为编程语言：



下载示例代码

可以通过你的账号，在 http://www.packtpub.com 上下载所有已经购买过的书籍中的示例代码。如果你在别处购买了本书，那么可以访问 http://www.packtpub.com/support，并进行注册，示例代码文件将会以邮件的形式直接发送给你。

生成器运行完毕后，项目模板将会为你的项目生成如下结构：

```
├── app
│   ├── index.html
│   ├── sass
│   │   └── styles.scss
│   ├── scripts
│   │   └── main.js
│   ├── styles
│   │   └── styles.css
│   └── ts
│       └── main.ts
├── bower.json
├── bower_components
│   └── ...
├── gulpfile.js
├── node_modules
│   └── ...
└── package.json
```

bower.json、package.json 和 gulpfile.js（Gulp 的自动化任务配置文件）文件都是自动生成的，这将节省不少时间。

在不理解生成器生成的代码前，盲目使用它并不是一个好主意。在以后我们使用 Yeoman 这样的脚手架工具来生成新项目时，推荐的做法是先对它生成的代码和任务做一个充分了解。

小结

在本章中，我们学习了如何使用一个版本控制系统，如何使用 Gulp 任务来使构建任务自动化。自动化构建使我们能够先检查 TypeScript 代码的质量，然后进行编译、测试和优化。还学习了如何安装第三方包以及它们的 TypeScript 描述文件。

在本章的最后，我们学习了如何使用持续集成服务器和自动化构建来减少潜在的代码集成问题。

下一章，我们将学习 TypeScript 中的函数。

3
使用函数

在第 1 章中，我们第一次认识了函数的用法。函数是任何 TypeScript 应用程序中的基础功能块，它们非常强大，值得用一整章去探索它们的潜力。

在这一章，我们将深入学习如何使用函数。这一章分成两个主要部分，在第一部分中，我们将快速回顾一些基本概念，然后学习一些不太常见的函数特性和用例。第一部分包括以下概念：

- 函数声明和函数表达式
- 函数类型
- 有可选参数的函数
- 有默认参数的函数
- 有剩余参数的函数
- 函数重载
- 特殊重载签名
- 函数作用域
- 立即调用函数
- 泛型
- tag 函数和标签模板

第二部分主要学习 TypeScript 异步编程能力和以下概念：

- 回调函数和高阶函数
- 箭头函数

- 回调地狱
- promise
- 生成器
- 异步调用函数（async 和 await）

在 TypeScript 中使用函数

在这一部分中，我们将主要聚焦于函数、形参和实参的声明和使用。也会介绍 TypeScript 中一个强大的特性：泛型。

函数声明和函数表达式

在第 1 章中，我们介绍了显式指定函数名称（命名函数）和不显式指定（匿名函数），但并未提及我们会使用两种不同类型的函数。

在下面这个例子中，命名函数 `greetNamed` 是一个函数声明，而 `greetUnnamed` 是一个函数表达式。现在先忽略前两行代码，它包含了 `console.log` 的打印日志语句：

```
console.log(greetNamed("John"));
console.log(greetUnnamed("John"));
function greetNamed(name : string) : string {
  if(name) {
    return "Hi! " + name;
  }
}
var greetUnnamed = function(name : string) : string {
  if(name){
    return "Hi! " + name;
  }
}
```

你可能会认为这两种函数非常相似，但是它们的行为并不一样。解释器会首先在代码解析阶段执行函数声明，而另一方面，除非函数表达式被赋值，否则就不会被执行。

这两种行为的主要区别在于一个被称为“变量提升”的过程，稍后我们将学习变量提升。

如果编译上述 TypeScript 代码片段到 JavaScript 中，并且尝试在浏览器中执行。很明显的是，由于 JavaScript 认识到这是一个函数声明，并且在程序执行之前对其进行解析，所以第一个 alert 代码段会执行。

但第二个 alert 会抛出一个异常，提示 greetUnnamed 不是一个函数。这个异常之所以被抛出，是因为在函数执行完之后 greetUnnamed 才被赋值。

函数类型

我们已经知道，可以在应用程序中通过使用可选的类型声明注解来显式声明一个元素的类型：

```
function greetNamed(name : string) : string {
  if(name) {
    return "Hi! " + name;
  }
}
```

在上述函数中，我们定义了参数 name 的类型（string）和返回值类型（string）。有时候我们不只需要定义函数中元素的类型，还需要定义函数本身的类型。让我们看下面这个例子：

```
var greetUnnamed : (name : string) => string;

greetUnnamed = function (name : string) : string {
  if(name){
    return "Hi! " + name;
  }
}
```

在上述例子中，我们声明了变量 greetUnnamed 及其类型。greetUnnamed 的类型是一个只包含一个名为 name 的 string 类型参数、在调用后会返回类型为 string 的函数。在声明了这个变量之后，一个完全符合变量类型的函数被赋值给了它。

我们也可以声明 greetUnnamed 的类型，并在同一行中将一个函数赋值给它，而不是分割成两行，比如下面这个例子：

```
var greetUnnamed : (name : string) => string = function(name : string)
: string {
```

```
  if(name){
    return "Hi! " + name;
  }
}
```

就像上面这个例子，之前的代码片段中同样声明了变量 greetUnnamed 和它的类型。我们也可以在同一行中把一个函数赋值给这个变量，被赋值的函数类型必须与变量类型相同。

在上面这个例子中，我们声明了 greetUnnamed 变量的类型并且把一个函数赋值给了它。这个函数的类型可以从被赋值的函数推断出来。因此，添加一个冗余的类型声明不是必需的，只是为了方便对类型的理解，但是这种冗余的类型注解会让代码难于阅读，这并非一个好的实践。

有可选参数的函数

与 JavaScript 不同，调用函数时传的参数的数量或类型不符合函数中定义的参数要求时，TypeScript 编译器会报错。让我们通过一个代码示例来证明这点：

```
function add(foo : number, bar : number, foobar : number) : number {
  return foo + bar + foobar;
}
```

上述函数名为 add 并包含三个 number 类型的参数：foo、bar 和 foobar。如果调用这个函数时没有完整地传入三个参数，会得到一个编译错误，提示提供的参数与函数声明中的参数无法匹配。

```
add();          // 提供的参数不匹配函数的签名。
add(2, 2);      // 提供的参数不匹配函数的签名。
add(2, 2, 2);   // 返回 6
```

在一些场景下，我们也许想调用这个函数且不提供所有的参数。TypeScript 提供了一个函数可选参数的特性，可以帮助增加这个函数的灵活性。可以在 TypeScript 中通过在函数参数后追加一个字符?，指定函数参数是可选的。更新一下前面的函数，将 foobar 参数从必选参数修改为可选参数。

```
function add(foo : number, bar : number, foobar? : number) : number {
```

```
    var result = foo + bar;
    if(foobar !== undefined) {
        result += foobar;
    }
    return result;
}
```

注意，我们将参数 foobar 的名称更改为了 foobar?，并在函数内部检测 foobar 参数是否被提供。修改后，在我们提供两个或者三个参数调用这个函数时，TypeScript 编译器不再抛出错误。

```
add();          // 提供的参数不匹配函数的签名。
add(2, 2);      // 返回 4
add(2, 2, 2);   // 返回 6
```

值得注意的是，可选参数必须位于必选参数列表的后面。

有默认参数的函数

当函数有可选参数时，我们必须检测参数是否被传递了（就像上一个例子中那样）。

在一些场景中，应为一个可选参数设置默认值。可以使用内联 if 结构重写 add 函数（从上一个代码片段中）：

```
function add(foo : number, bar : number, foobar? : number) : number {
  return foo + bar + (foobar !== undefined ? foobar : 0);
}
```

这个函数并没有错误，但是可以通过提供 foobar 参数的默认值，来代替标记其为可选参数，以改善其可读性。

```
function add(foo : number, bar : number, foobar : number = 0) :
number {
  return foo + bar + foobar;
}
```

我们只需要在声明函数签名时使用 = 操作符提供一个默认值，即可指定函数参数是可选的。TypeScript 编译器会在 JavaScript 输出结果中生成一个 if 结构，在 foobar 参数没有传递给函数时设置一个默认值。

```
function add(foo, bar, foobar) {
  if (foobar === void 0) {
    foobar = 0;
  }
  return foo + bar + foobar;
}
```

`void 0` 是 TypeScript 编译器检测一个变量是否为 undefined 的用法。几乎所有的开发者都使用 undefined 变量，几乎所有编译器都使用 `void 0`。

和可选参数一样，默认参数必须位于所有必选参数列表的后面。

有剩余参数的函数

我们已经了解了如何使用可选和默认参数来改善函数调用。让我们再回到上一个例子中：

```
function add(foo : number, bar : number, foobar : number = 0) :
number {
  return foo + bar + foobar;
}
```

我们已经学习了如何调用 `add` 函数并传递两个或者三个参数，但是如果希望允许其他开发者传递四个或者五个参数呢？不得不再添加两个额外的默认或者可选参数。那如果希望允许开发者传递任意数量的参数呢？解决方案是使用剩余参数。剩余参数语法允许把不限量的参数表示为一个数组：

```
function add(...foo : number[]) : number {
  var result = 0;
  for(var i = 0; i < foo.length; i++){
    result += foo[i];
  }
  return result;
}
```

看一下上述代码片段，我们用一个参数 `foo` 替换了参数 `foo`、`bar` 和 `foobar`。你会注意到，参数 `foo` 前面有一个三个点的省略号。一个剩余参数必须包含一个数组类型，否则就会出现编译错误。我们现在可以以任意数量的参数调用 `add` 函数：

```
add();              // returns 0
```

```
add(2);              // returns 2
add(2,2);            // returns 4
add(2,2,2);          // returns 6
add(2,2,2,2);        // returns 8
add(2,2,2,2,2);      // returns 10
add(2,2,2,2,2,2);    // returns 12
```

虽然没有具体的参数数量限制，理论上可以取数字类型的最大值。但实际上，这依赖于如何调用这个函数。

JavaScript 函数有一个被称为 arguments 的内建对象，这个对象可以通过 arguments 局部变量取到。arguments 变量是一个非常像数组的对象，包含了调用函数时的所有参数。

arguments 对象暴露出一些标准数组中的属性和方法，但不是全部。你可以通过完整的参考文档 https://developer.mozilla.org/en-US/docs/Web/ JavaScript/Reference/Functions/arguments 来学习它的特性。

如果我们检查 JavaScript 的输出会发现，TypeScript 遍历 arguments 参数，以便将所有参数添加到 foo 变量中：

```
function add() {
  var foo = [];
  for (var _i = 0; _i < arguments.length; _i++) {
    foo[_i - 0] = arguments[_i];
  }
  var result = 0;
  for (var i = 0; i < foo.length; i++) {
    result += foo[i];
  }
  return result;
}
```

对这个额外的针对函数参数的遍历存在一些争议。虽然难以想象这个额外的遍历是否会成为一个性能瓶颈，但是如果你认为这可能会对应用程序带来性能问题，应考虑不使用剩余参数而是只使用一个数组作为函数参数。

```
function add(foo : number[]) : number {
```

```
  var result = 0;
  for(var i = 0; i < foo.length; i++){
    result += foo[i];
  }
  return result;
}
```

上述代码只有一个包含了 number 类型的数组。调用 API 会和剩余参数有一些不同，但是可以去除对函数参数列表进行遍历的操作：

```
add();          // 提供的参数不匹配函数的签名。
add(2);         // 提供的参数不匹配函数的签名。
add(2,2);       // 提供的参数不匹配函数的签名。
add(2,2,2);     // 提供的参数不匹配函数的签名。

add([]);        // 返回0
add([2]);       // 返回2
add([2,2]);     // 返回4
add([2,2,2]);   // 返回6
```

函数重载

函数重载或方法重载是使用相同名称和不同参数数量或类型创建多个方法的一种能力。

在 TypeScript 中，我们可以通过声明一个函数的所有的函数签名，然后再将一个签名作为实现。让我们看一下这个例子：

```
function test(name: string) : string;       // 重载签名
function test(age: number) : string;        // 重载签名
function test(single: boolean) : string;    // 重载签名
function test(value: (string | number | boolean) : string { // 实现签名
  switch(typeof value){
  case "string":
    return `My name is ${value}.`;
  case "number":
    return `I'm ${value} years old.`;
  case "boolean":
    return value ? "I'm single." : "I'm not single.";
```

```
    default:
      console.log("Invalid Operation!");
    }
}
```

你可能对上面代码中的字符串语法不是很熟悉。这种语法被称为模板字符串。模板字符串被反引号（``）而不是单引号或双引号包围。模板字符串可以包含占位符，它被美元符号和花括号指定（${expression}）。占位符中的表达式和字符被传入一个函数中，这个函数默认只是将各个部分连接为一个字符串。

正如在上面的例子中所见到的，我们重载了 test 函数三次。第一个接受一个字符串作为其唯一参数，第二个接受一个数字，最后一个则接受一个布尔值作为其唯一参数。所有重载签名都必须兼容，如果一个函数试图返回一个数字而另一个试图返回字符串，那么将会得到一个编译错误。

实现签名必须兼容所有的重载签名，总是在参数列表的最后，接受一个 any 类型或联合类型的参数作为它的参数。

直接执行实现签名将会得到一个编译错误。

```
test("Remo");                    // 返回 "My name is Remo."
test(26);                        // 返回 "I'm 26 years old.";
test(false);                     // 返回 "I'm not single.";
test({ custom : "custom" });     // 报销
```

特定重载签名

我们可以使用一个特定的签名来创建具有同样名称、参数数量但是有不同返回类型的多个函数。为了创建一个特定签名，必须将函数的参数类型指定为一个字符串。这个字符串被用于定义哪个函数重载被调用：

```
interface Document {
  createElement(tagName: "div"): HTMLDivElement;         // 特定重载签名
  createElement(tagName: "span"): HTMLSpanElement;       // 特定重载签名
  createElement(tagName: "canvas"): HTMLCanvasElement;   // 特定重载签名
  createElement(tagName: string): HTMLElement;           // 非特定重载签名
}
```

在这个例子中，我们为函数 createElement 声明了三个特定重载签名和一个非特定重载签名。

当在一个对象中声明特定签名时，这个对象中必须被赋予至少一个非特定重载签名。从上面的例子中可以发现，createElement 属性属于一个包含了三种特定重载签名的类型，并被赋予到非特定重载签名中。

当编写重载声明时，必须在最后列出非重载签名。

回顾第 1 章，我们也可以通过使用联合类型来创建有同样名称和参数数量但是参数类型不一样的函数。

函数作用域

一些低级语言（比如 C 语言）包含了底层内存管理特性。在拥有更高层抽象的编程语言（比如 TypeScript）中，当变量被创建时，内存就已经被分配，并且在它们不被使用时会被清理掉。这个清理内存的过程被称为*垃圾回收*，由 JavaScript 运行时的垃圾回收器实现。

垃圾回收器通常很高效，但希望它能帮我们处理所有内存泄露时就会有问题。垃圾回收器会在变量脱离作用域时清理掉它们，所以理解 TypeScript 的作用域的工作机制对我们来说是非常重要的，这样我们就能理解变量的生命周期。

一些编程语言使用程序的源代码结构来指定哪些变量被引用（词法作用域），另一些编程语言使用程序的运行时堆栈状态来指定哪些变量被引用（动态作用域）。主要的现代编程语言使用词法作用域（包括 TypeScript），词法作用域往往比动态作用域更易被人和分析工具理解。

在绝大多数词法作用域编程语言中，变量的作用域为代码块（一段被花括号{}括起来的代码）。在 TypeScript（和 JavaScript）中，变量的作用域在一个函数中：

```
function foo() : void {
  if(true){
    var bar : number = 0;
  }
  alert(bar);
}
foo(); // shows 0
```

上面这个 foo 函数包含了一个 if 结构。我们在 if 结构中声明了一个名为 bar 的数字类型的变量，之后试图通过 alert 函数显示这个变量的值。

我们可能会认为这段代码会在第 5 行报错，因为 bar 变量应该在函数调用时超出作用域。但是，当调用这个 foo 函数的时候，alert 函数会显示变量的值，并且不报告任何错误，因为所有函数中的变量都在整个函数体的作用域内，即使是在另一个代码块中（除了函数代码块之外）。

这看起来真的非常奇怪，但是一旦我们了解了底层原理，就会很容易理解。在运行时中，所有的变量声明都会在函数执行前移动到函数的顶端，这种行为被称为变量提升。

TypeScript 先被编译为 JavaScript 代码然后再执行，这意味着 TypeScript 应用在运行时是一个 JavaScript 应用。因为这个原因，当提及 TypeScript 运行时的时候，我们就是在说 JavaScript 运行时。我们会在第 5 章深入学习运行时。

所以，在上述代码被执行之前，运行时会将变量声明提升到函数的顶部。

```
function foo() : void {
  var bar :number;
  if(true){
    bar= 0;
  }
  alert(bar);
}
```

这意味着可以在变量声明前直接使用它，来看下面这个例子：

```
function foo2() : void {
  bar = 0;
  var bar : number;
  alert(bar);
}

foo2();
```

在上述代码片段中，我们定义了一个函数 foo2，并且在它的函数体中，先给一个名为 bar 的变量赋值为 0。此时，变量还没有被声明。在第二行，我们才实际声明了变量 bar

和它的类型。在最后一行，我们使用 alert 函数显示变量 bar 的值。

因为在函数中的任何地方（除了另一个函数）声明变量都和在函数顶端声明一样，所以 foo2 函数在运行时被转换为以下的样子：

```
function foo2(): void {
  var bar : number;
  bar = 0;
  alert(bar);
}

foo2();
```

因为有 Java 或者 C# 背景的开发者不习惯使用函数作用域，这是 JavaScript 最饱受批评的特性之一。负责 ECMAScript 6 标准的人已经了解到这一点，所以他们引入了新的关键字 let 和 const。

let 关键字允许我们将作用域设置在代码段（if、while、for 等）而不是函数中。我们可以更新第一个示例的代码来演示 let 是如何工作的。

```
function foo() : void {
  if(true){
    let bar : number = 0;
    bar = 1;
  }
  alert(bar); // error
}
```

bar 变量现在使用 let 关键字进行声明，所以它只会在 if 代码块中被访问。变量不会被提升到函数 foo 的顶端，并且无法被脱离 if 代码段的 alert 函数访问。

当函数使用 const 进行定义时，它和 let 拥有同样的作用域规则，并且不能被重新赋值。

```
function foo() : void {
  if(true){
    const bar : number = 0;
    bar = 1; // error
  }
  alert(bar); // error
}
```

如果试图编译这一段代码，会得到一个编译错误，因为 bar 变量不能在 if 代码段之外被访问（就像使用 let 关键字一样），并且当尝试给 bar 变量赋一个新值时会出现一个新的错误，第二个错误是因为常量不能在初始化之后重新赋值。

立即调用函数

立即调用函数表达式（IIFE）是一种设计模式，使用函数作用域作为一个词法作用域。IIFE 可以被用于防止全局作用域中的变量提升导致的污染。举例如下：

```
var bar = 0; // 全局的

(function() {
  var foo : number = 0; // 在函数作用域中
  bar = 1; // 在全局作用域中
  console.log(bar); // 1
  console.log(foo); // 0
})();

console.log(bar); // 1
console.log(foo); // 错误
```

在上述例子中，我们使用 IIFE 包装了两个变量的声明（foo 和 bar）。foo 变量作用于 IIFE 函数中并且在全局作用域中不可用，这解释了为什么当我们尝试在最后一行访问它的时候会产生一个错误。

我们也可以给 IIFE 传递一个变量，以便更好地控制在作用域之外创建的变量。

```
var bar = 0; // 全局的

(function(global) {
  var foo : number = 0; // 在函数作用域中
  bar = 1; // 在全局作用域中
  console.log(global.bar); // 1
  console.log(foo); // 0
})(this);
console.log(bar); // 1
console.log(foo); // 错误
```

这一次，因为我们没有在 IIFE 中调用 this 操作符，所以把指向全局作用域的 this 操作符作为唯一的参数传递给了 IIFE。在 IIFE 中，this 操作符传递来的参数称为 global，我们可以更好地控制全局作用域中声明的 bar 变量，而不是 foo 变量。

此外，IIFE 允许我们访问公开方法，隐藏函数内的私有变量，让我们看一下这个例子：

```
class Counter {
  private _i : number;
  constructor() {
    this._i = 0;
  }
  get() : number {
    return this._i;
  }
  set(val : number) : void {
    this._i = val;
  }
  increment() : void {
    this._i++;
  }
}
var counter = new Counter();
console.log(counter.get()); // 0
counter.set(2);
console.log(counter.get()); // 2
counter.increment();
console.log(counter.get()); // 3
console.log(counter._i); // 错误：'_i'为私有属性
```

通常约定，TypeScript 和 JavaScript 开发者使用下画线（_）开始的变量名作为私有变量名。

我们定义了一个名为 Counter 的类，它有一个数字类型的私有属性 _i。这个类也有两个 get 和 set 方法，用来设置和获取这个私有属性_i 的值。我们还创建了 Counter 类的实例并调用了 set、get 和 increment 方法来观察代码是否如设想般正常工作。如果我们尝试访问 Counter 类实例的_i 属性，将得到一个错误，因为这个属性是私有的。

如果我们编译上面的 TypeScript 代码（仅编译类定义的部分），并且观察生成的 JavaScript 代码，可以看到：

```
var Counter = (function () {
    function Counter() {
      this._i = 0;
    }
    Counter.prototype.get = function () {
      return this._i;
    };
    Counter.prototype.set = function (val) {
      this._i = val;
    };
    Counter.prototype.increment = function () {
      this._i++;
    };
    return Counter;
})();
```

生成的 JavaScript 代码在大部分场景下都能完美运行，但如果在浏览器内执行这段代码，创建一个 Counter 的实例并且访问它的_i 属性，也不会遇到任何错误，因为 TypeScript 不会生成运行时的私有属性。有时候我们需要编写在运行时拥有私有变量的函数，比如，假设要发布一个 JavaScript 开发者使用的库，可以使用 IIFE 来模拟允许公共地访问一个方法，但在函数内部有一个私有的值：

```
var Counter = (function () {
    var _i : number = 0;
    function Counter() {
    }
    Counter.prototype.get = function () {
      return _i;
    };
    Counter.prototype.set = function (val : number) {
      _i = val;
    };
    Counter.prototype.increment = function () {
      _i++;
```

```
    };
    return Counter;
})();
```

在上面的例子中，所有的代码都跟 TypeScript 生成的 JavaScript 代码类似，除了_i 并不是 Counter 类的属性，而是一个在 Counter 闭包内的变量。

闭包的职责是引用自由变量。换句话说，函数定义的位置所在的闭包会记住它所在的环境（作用域内的变量）。我们会在第 5 章继续深入探索闭包。

如果现在我们在浏览器内运行编译后的代码并直接访问_i 属性，会发现这个属性在运行时也是私有的：

```
var counter = new Counter();
console.log(counter.get()); // 0
counter.set(2);
console.log(counter.get()); // 2
counter.increment();
console.log(counter.get()); // 3
console.log(counter._i); // undefined
```

在一些场景下，对作用域和闭包进行精确地控制，代码会变得跟编译后的 JavaScript 很像。但记住，只要我们编写的组件（类、模块等）是被其他 TypeScript 组件使用的，就不需要为实现运行时私有属性而烦恼。我们将在第 5 章继续深入学习 TypeScript 的运行时。

范型

Andy Hunt 和 Dave Thomas 在 *The Pragmatic Programmer* 一书中提出了 don't repeat yourself（DRY）原则，DRY 原则旨在减少各种类型的信息重复。现在来看一下什么是范型函数，并理解它是如何帮助我们遵循 DRY 原则的。

我们从定义一个简单的 User 类开始：

```
class User {
  name : string;
```

```
  age : number;
}
```

现在我们已经有了 User 类，再来编写一个名为 getUsers 的函数，它会通过 AJAX 请求一个用户列表：

```
function getUsers(cb : (users : User[]) => void) : void {
  $.ajax({
    url: "/api/users",
    method: "GET",
    success: function(data) {
      cb(data.items);
    },
    error : function(error) {
      cb(null);
    }
  });
}
```

在这个例子中，我们使用了 jQuery。记住，要先新建一个 package.json 文件并使用 npm 安装 jQuery。还需要使用 tsd 安装 jQuery 的类型定义文件。可以在第 1 章和第 2 章中得到更多关于这些操作的帮助。

getUsers 函数接受一个函数作为参数，并在 AJAX 请求成功后调用它。可以这样调用它：

```
getUsers(function(users : User[]){
  for(var i; users.length; i++){
    console.log(users[i].name);
  }
});
```

想象一下，我们需要一个功能几乎完全相同的函数。但是这次我们将会使用 Order 实例取代 User：

```
class Order {
```

```
  id : number;
  total : number;
  items : any[]
}
```

getOrders 函数与 getUsers 函数几乎完全相同，它在请求中使用了一个不一样的 URL，它将一个 Orders 数组传入回调函数而不是 User 数组：

```
function getOrders(cb : (orders : Order[]) => void) : void {
  $.ajax({
    url: "/api/orders",
    method: "GET",
    success: function(data) {
      cb(data.items);
    },
    error : function(error) {
      cb(null);
    }
  });
}
getOrders(function(orders : Orders[]){
  for(var i; orders.length; i++){
    console.log(orders[i].total);
  }
});
```

我们可以使用范型来避免这种类型的代码重复。范型编程是一种程序语言的风格，它允许程序员使用以后才会定义的类型，并在实例化时作为参数指定这些类型。我们将编写一个名为 getEntities 的范型函数，它接受两个参数：

```
function getEntities<T>(url : string, cb : (list : T[]) => void) : void {
  $.ajax({
    url: url,
    method: "GET",
    success: function(data) {
      cb(data.items);
    },
    error : function(error) {
```

```
        cb(null);
      }
    });
}
```

在函数名后面增加了一对角括号（<>）来表明这是一个范型函数。如果角括号内是一个字符 T，它表示一个类型。函数的第一个参数是字符串类型的 url，第二个参数是函数类型的 cb，它接受一个 T 类型的参数作为唯一的参数。

现在我们可以使用范型方法并指明类型 T 的具体类型：

```
getEntities<User>("/api/users",function(users : Users[]) {
  for(var i; users.length; i++) {
    console.log(users[i].name);
  }
});

getEntities<Order>("/api/orders", function(orders : Orders[]) {
  for(var i; orders.length; i++) {
    console.log(orders[i].total);
  }
});
```

tag 函数和标签模板

我们已经知道了如何像下面这样使用模板字符串了：

```
var name = 'remo';
var surname = jansen;
var html = `<h1>${name} ${surname}</h1>`;
```

但是有一个模板字符串的用法我们之前故意跳过了，因为它紧密地与我们要讲的一种叫 tag 的函数联系在一起。

可以使用 tag 函数扩展和修改模板字符串的行为。当我们在模板字符串上应用一个 tag 函数时，这个模板字符串就变成了*标签模板*。

我们将实现一个名为 htmlEscape 的 tag 函数。要使用 tag 函数，必须在 tag 函数后紧跟一个模板字符串：

```
var html = htmlEscape `<h1>${name} ${surname}</h1>`;
```

一个标签模板必须返回一个字符串，并接受下面的参数：

- 第一个参数是一个数组，它包含了模板字符串中所有的静态字面量（在上面的例子中是<h1>和</h1>）。
- 剩余的参数是模板字符串中所有的变量（在上面的例子中是 name 和 surname）。

我们现在知道了一个 tag 函数的定义：

```
tag(literals : string[], ...values : any[]) : string
```

接下来，让我们实现 htmlEscape tag 函数：

```
function htmlEscape(literals, ...placeholders) {
     let result = "";
     for (let i = 0; i < placeholders.length; i++) {
          result += literals[i];
          result += placeholders[i]
                 .replace(/&/g, '&amp;')
                 .replace(/"/g, '&quot;')
                 .replace(/'/g, '&#39;')
                 .replace(/</g, '&lt;')
                 .replace(/>/g, '&gt;');
     }
     result += literals[literals.length - 1];
     return result;
}
```

上面的函数逐字迭代字符串和值来确保所有的 HTML 代码被正确转义，防止代码注入攻击。

使用 tag 函数最大的好处是它允许我们创建一个自定义的模板字符串处理器。

这个特性在 TypeScript 1.6 发行时可用。

TypeScript 中的异步编程

现在我们已经看到如何使用函数，将会继续探索如何将它与一些原生对象组合在一起

编写异步程序。

回调和高阶函数

在 TypeScript 中，函数可以作为参数传给其他函数。被传递给其他函数的函数叫作回调。函数也可以被另一个函数返回。那些接受函数为参数（回调）或返回另一个函数的函数被称为高阶函数。回调通常被用在异步函数中：

```
var foo = function() { // 回调
  console.log('foo');
}
function bar(cb : () => void) { // 高阶函数
  console.log('bar');
  cb();
}
bar(foo); // 输出 'bar' 然后输出 'foo'
```

箭头函数

在 TypeScript 中，我们可以使用 function 表达式或者箭头函数定义一个函数。箭头函数是 function 表达式的缩写，并且这种词法会在其作用域内绑定 this 操作符。

在 TypeScript 中，this 操作符的行为和其他语言有一点不一样。当在 TypeScript 中定义一个类的时候，可以使用 this 指向这个类自身的属性。让我们看一个例子：

```
class Person {
  name : string;
  constructor(name : string) {
    this.name = name;
  }
  greet() {
    alert(`Hi! My name is ${this.name}`);
  }
}
var remo = new Person("Remo");
remo.greet(); // "Hi! My name is Remo"
```

我们定义了一个 Person 类，并包含了一个名为 name 的字符串类型的属性。这个类有

一个构造函数和一个叫作 greet 的方法。我们可以新建一个名为 remo 的实例，并且调用 greet 方法，它在内部使用 this 操作符访问 remo 的 name 属性。在 greet 方法内部，this 操作符指向装载了 greet 方法的对象。

我们必须谨慎使用 this 操作符，因为在一些场景下它将会指向错误的值。我们在上一个例子中加入一个方法：

```
class Person {
  name : string;
  constructor(name : string) {
    this.name = name;
  }
  greet() {
    alert(`Hi! My name is ${this.name}`);
  }
  greetDelay(time : number) {
    setTimeout(function() {
      alert(`Hi! My name is ${this.name}`);
    }, time);
  }
}
var remo = new Person("remo");
remo.greet(); // "Hi! My name is remo"
remo.greetDelay(1000); // "Hi! My name is "
```

在 greetDelay 方法中，我们实现了一个和 greet 方法几乎相同的功能。但这次这个方法接受一个名为 time 的参数，它用来延迟问候信息的发出。

为了延迟问候消息，我们使用了 setTimeout 函数和一个回调函数。当定义了一个异步函数时（包含回调），this 关键字就会改变它指向的值，指向匿名函数。这就解释了为什么 remo 没有通过 greetDelay 方法显示出来。

提醒一下，箭头函数表达式的词法会绑定 this 操作符。这就意味着，我们可以增加函数而不用担心 this 的指向。现在将上面例子中的 function 表达式替换成箭头函数：

```
class Person {
    name : string;
  constructor(name : string) {
    this.name = name;
```

```
    }
    greet() {
      alert(`Hi! My name is ${this.name}`);
    }
    greetDelay(time : number) {
      setTimeout(() => {
        alert(`Hi! My name is ${this.name}`);
      }, time);
    }
}

var remo = new Person("remo");
remo.greet(); // "Hi! My name is remo"
remo.greetDelay(1000); // "Hi! My name is remo"
```

通过使用箭头函数，我们可以保证 `this` 操作符指向的是 `Person` 的实例而不是 `setTimeout` 的回调函数。如果我们执行 `greetDelay` 方法，`name` 属性会按预期正常显示。

下面的代码是 TypeScript 编译生成的。当编译箭头函数时，TypeScript 编译器会生成一个 `this` 的别名，名为 `_this`。这个别名用来确保 `this` 指向的是正确的对象。

```
Person.prototype.greetDelay = function (time) {
  var _this = this;
  setTimeout(function () {
    alert("Hi! My name is " + _this.name);
  }, time);
};
```

回调地狱

我们已经看到了回调函数和高阶函数是两个强有力且灵活的 TypeScript 功能。然而，回调的使用可能会导致称为*回调地狱*的维护性问题。我们现在将编写一个实际场景中出现的例子来看一下什么是回调地狱以及如何轻松解决这个问题。

记住，你可以在随书的例子中找到完整的代码。

我们将使用 handlebars 和 jQuery 库，所以先通过 npm 和 tsd 安装这两个库和它们的类型描述文件：

```
///<reference path="../typings/handlebars/handlebars.d.ts" />
///<reference path="../typings/jquery/jquery.d.ts" />
```

为了让代码更易读，我们会设置一个回调类型的别名：

```
type cb = (json : any) => void;
```

现在我们需要声明一个 View 类。View 类允许设置以下属性。

- **容器**：一个 DOM 选择器，是 View 被插入的地方。
- **模板 URL**：会返回一个 Handlebars 模板的 URL。
- **服务 URL**：一个会返回 JSON 数据的网络服务 URL。
- **请求参数**：被发送到网络服务的数据。

这个 View 类是这样实现的：

```
class View {
  private _container : string;
  private _templateUrl : string;
  private _serviceUrl : string;
  private _args : any;
  constructor(config){
    this._container = config.container;
    this._templateUrl = config.templateUrl;
    this._serviceUrl = config.serviceUrl;
    this._args = config.args;
  } //...
```

在定义了这个类的构造函数和属性之后，我们将增加一个名为 _loadJson 的私有方法。这个方法接受服务 URL、请求参数、请求成功后的回调和请求失败后的回调作为它的参数。在方法内部，使用服务 URL 和请求参数发送一个 jQuery AJAX 请求：

```
private _loadJson(url : string, args : any, cb : cb, errorCb :
cb) {
  $.ajax({
    url: url,
```

```
    type: "GET",
    dataType: "json",
    data: args,
    success: (json) => {
      cb(json);
    },
    error: (e) => {
      errorCb(e);
    }
  });
} //...
```

handlebars 是一个能在浏览器中编译并渲染 HTML 模板的库。这些模板可以帮我们将 JSON 转成 HTML，将在后面的部分中多次提到这个库，但如果你从未使用过 handlebars 也不用担心，这个部分不是关于 handlebars 的。

这个部分是关于如何使用回调控制一组任务的执行流程的。如果你想了解更多关于handlebars的信息，可访问http://handlebarsjs.com/。

下面这个方法和前一个几乎一样，但不是加载 JSON 而是加载一个 handlebars 模板：

```
private _loadHbs(url : string, cb : cb, errorCb : cb) {
  $.ajax({
    url: url,
    type: "GET",
    dataType: "text",
    success: (hbs) => {
      cb(hbs);
    },
    error: (e) => {
      errorCb(e);
    }
  });
} //...
```

下面这个函数接受 handlebars 模板字符串作为参数，并尝试使用 handlebars 编译函

数去编译它。类似上面的例子，我们使用回调函数，它在编译成功或失败后被调用：

```
private _compileHbs(hbs : string, cb : cb, errorCb : cb) {
  try
  {
    var template = Handlebars.compile(hbs);
    cb(template);
  }
  catch(e) {
    errorCb(e);
  }
} //...
```

在下面这个函数中，我们将编译好的模板和已加载的 JSON 数据作为参数传入，并遵循模板语法规则，将它们放在一起使 JSON 转成 HTML。和上一个例子一样，使用回调并在操作成功或失败后调用它：

```
private _jsonToHtml(template : any, json : any, cb : cb, errorCb: cb) {
  try {
    var html = template(json);
    cb(html);
  }
  catch(e) {
    errorCb(e);
  }
}
//...
```

下面这个函数使用_jsonToHtml 生成的 HTML 作为参数，并将它插入 DOM 元素中：

```
private _appendHtml = function (html : string, cb : cb, errorCb : cb) {
  try {
    if($(this._container).length === 0) {
      throw new Error("Container not found!");
    }
    $(this._container).html(html);
    cb($(this._container));
  }
  catch(e) {
```

```
    errorCb(e);
  }
}
//...
```

现在我们已经拥有了一些使用回调的函数，将在一个名为 render 的方法中同时使用它们。render 方法控制这些任务的执行流程，并用下面的顺序执行它：

1. 加载 JSON 数据
2. 加载模板
3. 编译模板
4. 将 JSON 转化成 HTML
5. 将 HTML 插入到 DOM 元素中

每一个任务都接受一个异步任务成功时的回调，和一个异步任务出错时的回调:

```
public render (cb : cb, errorCb : cb) {
  try
  {
    this._loadJson(this._serviceUrl, this._args, (json) => {
      this._loadHbs(this._templateUrl, (hbs) => {
        this._compileHbs(hbs, (template) => {
          this._jsonToHtml(template, json, (html) => {
            this._appendHtml(html, cb);
          }, errorCb);
        }, errorCb);
      }, errorCb);
    }, errorCb);
  }
  catch(e){
     errorCb(e);
  }
}
```

通常情况下，需要避免像上面这样的嵌套的回调，因为这会导致以下问题：

- 让代码难以理解
- 让代码难以维护（重构、重用等）
- 让异常处理更加困难

promise

在看了如何使用回调以及它如何导致一些维护性问题后，我们现在来了解一下 promise 以及如何使用 promise 编写更好的异步代码。promise 背后最主要的思想是对异步操作结果的一个承诺。一个 promise 一定是以下几种状态之一。

- **pending**：promise 的初始化状态。
- **fulfilled**：代表异步操作成功的 promise 状态。
- **rejected**：代表异步操作失败的 promise 状态。

当一个 promise 处于 fulfilled 或 rejected 状态后，它的状态就永远不可更改了。让我们看一下 promise 的基本语法：

```
function foo() {
  return new Promise((fulfill, reject) => {
    try {
      // do something
      fulfill(value);
    }
    catch(e){
      reject(reason);
    }
  });
}
foo().then(function(value){ console.log(value); })
     .catch(function(e){ console.log(e); });
```

这里的 try...catch 语句只是为了展示如何 fulfill 和 reject 一个 promise。在 Promise 函数内，并不需要 try...catch 语句，因为 promise 会在有异常抛出的时候自动进入 rejected 状态。

上面的代码中声明了一个名为 foo 的函数，它返回一个 promise。一个 promise 包含了一个名为 then 的方法，它接受一个函数，在 promise 被 fulfilled 后调用。promise 还提供了一个名为 catch 的方法，它在 promise 被 rejected 之后调用。

现在我们将回到回调地狱的例子，然后对代码做一些改动并使用 promise 代替回调。

和之前一样，需要 handlebars 和 jQuery，让我们先载入它们的定义文件。这次还需要额外地引入一个叫作 Q 的库的类型定义文件：

```
///<reference path="../typings/handlebars/handlebars.d.ts" />
///<reference path="../typings/jquery/jquery.d.ts" />
///<reference path="../typings/q/q.d.ts" />
```

我们将使用库提供的 Promise 对象而不是原生的，因为库会兼容旧版本的浏览器。将在这个例子中使用名为 Q (v1.0.1) 的 promise 库。可以访问 https://github.com/kriskowal/q 了解更多关于它的信息。

这个类的名字已经从 View 变成了 ViewAsync，但其他部分都与之前的相同:

```
class ViewAsync {
  private _container : string;
  private _templateUrl : string;
  private _serviceUrl : string;
  private _args : any;
  constructor(config) {
    this._container = config.container;
    this._templateUrl = config.templateUrl;
    this._serviceUrl = config.serviceUrl;
    this._args = config.args;
  }
//...
```

作为惯例，很多开发者将 Async 放在函数名的后面，来指明一个函数是异步的。

我们将在 _loadJsonAsync 中使用第一个 promise。这个函数在回调例子中是 _loadJson。我们从函数的签名中移除了回调的定义，最后将这个函数用一个 promise 对象包裹起来，并在异步任务成功或失败的时候调用 resolve 或 reject。

```
private _loadJsonAsync(url : string, args : any) {
```

```
    return Q.Promise(function(resolve, reject) {
      $.ajax({
        url: url,
        type: "GET",
        dataType: "json",
        data: args,
        success: (json) => {
          resolve(json);
        },
        error: (e) => {
          reject(e);
        }
      });
    });
  }
  //...
```

我们随后重构（重命名、移除回调、使用 promise 包裹逻辑部分等）这个类的所有方法（_loadHbsAsync、compileHbsAsync 和_appendHtmlAsync）：

```
  private _loadHbsAsync(url : string) {
    return Q.Promise(function(resolve, reject) {
      $.ajax({
        url: url,
        type: "GET",
        dataType: "text",
        success: (hbs) => {
          resolve(hbs);
        },
        error: (e) => {
          reject(e);
        }
      });
    });
  }
  private _compileHbsAsync(hbs : string) {
    return Q.Promise(function(resolve, reject) {
      try
```

```
    {
      var template : any = Handlebars.compile(hbs);
      resolve(template);
    }
    catch(e) {
      reject(e);
    }
  });
}
private _jsonToHtmlAsync(template : any, json : any) {
  return Q.Promise(function(resolve, reject) {
    try
    {
      var html = template(json);
      resolve(html);
    }
    catch(e) {
      reject(e);
    }
  });
}
private _appendHtmlAsync(html : string, container : string) {
  return Q.Promise((resolve, reject) => {
    try
    {
      var $container : any = $(container);
      if($container.length === 0) {
        throw new Error("Container not found!");
      }
      $container.html(html);
      resolve($container);
    }
    catch(e) {
      reject(e);
    }
  });
}
//...
```

renderAsync 方法（之前叫作 render）和以前有一些区别。

下面这个函数将会被 promise 包裹函数逻辑，调用_loadJsonAsync，并将结果传入 getJson。回到_loadJsonAsync 方法，我们会注意到它的返回值也是一个 promise。因此，getJson 变量也是一个 promise，它在我们的视图需要的数据返回后马上进入 fulfilled 状态。

将调用_loadHbsAsync 返回的 promise 的 then 方法，它使我们可以在 promise 的状态变为 fulfilled 时将_loadHbsAsync 函数的返回结果传入_compileHbsAsync 中。

```
public renderAsync() {
  return Q.Promise((resolve, reject) => {
    try {
      // 将getJson赋值为promise
      var getJson = this._loadJsonAsync(this._serviceUrl, this._args);

      // 将getTemplate赋值为promise
      var getTemplate = this._loadHbsAsync(this._templateUrl)
      .then(this._compileHbsAsync);

      // 并行执行这两个promises
      Q.all([getJson, getTemplate]).then((results) => {
        var json = results[0];
        var template = results[1];

        this._jsonToHtmlAsync(template, json)
       .then((html : string) => {
          return this._appendHtmlAsync(html, this._container);
       })
       .then(($container : any) => { resolve($container); });
      });
    }
    catch(error) {
      reject(error);
    }
  });
}
```

一旦定义了 getJson 和 getTemplate 变量（它们的值都是 promise），我们就可以

使用 Q 提供的 all 方法并行执行 getJson 和 getTemplate。

Q 的 all 方法接受一个 promise 数组和一个回调作为参数。一旦数组中的 promise 全部 fulfilled，回调函数就会被调用，并且名为 results 的数组会被传入 fulfilled 回调中。这个数组包含了传入的所有 promise fulfilled 之后的值。

在 Q 的 all 方法内部，我们将使用加载好的 JSON 和编译好的模板作为参数调用 _jsonToHtmlAsync。最后用 then 方法调用_appendHtmlAsync 并 resolve 这个 promise。

观察这个例子，可以发现，使用 promise 可以在 render 方法中更好地控制执行流程。记住你可以使用以下 4 种不同类型的异步流程控制。

- **并行**：异步任务将会并行执行。在例子中我们看到了对 getJson 和 getTemplate 使用了 all 方法。
- **串行**：一组任务串行，前一个任务完成后不会传参给后一个任务。
- **瀑布流**：一组任务串行，每一个任务会将结果传到下一个任务中。这个实现在任务互相依赖时非常有用。在上面的例子中，可以看到，_loadHbsAsync promise 将它的结果传入到_compileHbsAsync promise 中使用了这种流程控制。
- **混合**：这是一种并行、串行和瀑布流的任意组合实现。例子中的 render 方法使用了混合类型的异步流程控制。

生成器

如果在 TypeScript 中调用一个函数，我们可以肯定一旦这个函数开始运行，在它运行完成之前其他代码都不能运行。然而，一种新的函数可能会在函数执行的过程中将这个函数暂停一次或多次，并在随后恢复它的运行，而且可以让其他代码在暂停的过程中运行，在 TypeScript 和 ES6 中即将实现这个功能。这种新型函数被称为*生成器*。

一个生成器代表了一个值的序列。生成器对象的接口只是一个迭代器，可以调用 next() 函数使它产出结果。

可以使用 function 关键字后面跟着一个星号(*)定义一个生成器的构造函数。yield 关键字被用来暂停函数的执行并返回一个值。让我们来看一个例子：

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
```

```
  return 5;
}

var bar = new foo();
bar.next(); // Object {value: 1, done: false}
bar.next(); // Object {value: 2, done: false}
bar.next(); // Object {value: 3, done: false}
bar.next(); // Object {value: 4, done: false}
bar.next(); // Object {value: 5, done: true}
bar.next(); // Object { done: true }
```

你可以看到，这个迭代器有 5 个步骤。第一次调用 `next()` 的时候，函数会执行到第一个 `yield` 的位置，并且它会返回值 1 并且停止运行，直到 `next()` 再次被调用。可以看到，我们现在可以终止函数的执行了。可以像下面的例子这样编写一个无限循环而不会导致栈溢出：

```
function* foo() {
  var i = 1;
  while (true) {
    yield i++;
  }
}

var bar = new foo();
bar.next(); // Object {value: 1, done: false}
bar.next(); // Object {value: 2, done: false}
bar.next(); // Object {value: 3, done: false}
bar.next(); // Object {value: 4, done: false}
bar.next(); // Object {value: 5, done: false}
bar.next(); // Object {value: 6, done: false}
bar.next(); // Object {value: 7, done: false}
// ...
```

生成器给了我们以同步的方式编写异步代码的可能性，只要我们在异步事件发生的时候调用生成器的 `next()` 方法就能做到这一点。

异步函数——async 和 await

*异步函数*是一个即将到来的 TypeScript 的特性。一个异步函数是在异步操作中被调用的函数。开发者可以使用 `await` 关键字等待异步结果的到来而不会阻塞程序的执行。

当编译目标是 ES6 时，异步函数将会被 promise 实现，编译目标是 ES5 和 ES3 时会使用 promise 的兼容版本实现。

与使用 promise 相比，使用异步函数可以显著提高程序的可读性，在技术上使用 promise 可以达到和同步函数同样的效果：

```
var p: Promise<number> = /* ... */;
async function fn(): Promise<number> {
  var i = await p;
  return 1 + i;
}
```

上面的代码片段中我们声明了一个名为 `p` 的 promise，这个 promise 将会等待被执行。在等待期间，程序并不会被阻塞，因为我们是在一个名为 `fn` 的异步函数中等待它。可以看到，`fn` 函数前面有一个 `async` 关键字，这将会对编译器指明这是一个异步函数。

在函数内部，`await` 关键字被用来暂停代码执行，直到 `p` 被 fulfilled。可以看到，这个语法更有语义并更加简洁。

`async` 和 `await` 在最新的 TypeScript 中已经可用。

小结

本章中，我们深入地学习了如何使用函数，了解了回调、promise 和生成器的用法，并探索了 TypeScript 异步编程的能力。

在下一章中，我们将会看到如何使用类、接口和其他 TypeScript 中的面向对象的编程特性。

4

TypeScript 中的面向对象编程

在上一章中，我们浏览了函数和一些异步控制技术的用法。在这一章中，我们会看到怎样在像类或者模块这种可复用的组件中组织函数。本章分为两个部分，第一部分包括以下几个要点：

- SOLID 原则
- 类
- 关联、聚合和组合
- 继承
- 混合
- 范型类
- 范型约束
- 接口

在第二部分，我们将了解如何声明并使用命名空间和外部模块：

- 命名空间（内部模块）
- 外部模块
- 异步模块定义
- CommonJS 模块
- ES6 模块
- Browserify 和通用模块定义
- 循环依赖

SOLID 原则

在早期的软件开发中，开发者通常使用过程式的程序语言编写代码。在过程式的语言中，程序遵循自顶向下的原则开发，并且逻辑都包裹在函数中。

当程序员意识到过程式的语言无法提供他们需要的抽象层次、可维护性和复用性时，出现了像模块式或结构式这类新型的程序语言。

开发者社区创建了一系列的最佳实践和设计模式来提高过程式语言的抽象层级和复用性，但其中部分指南要求使用者具有相当的专业知识。为了促进编程人员更好地遵循这些指南，一种新型的程序语言被创造出来了，这就是大家所熟知的*面向对象编程语言*（OOP）。

随着时间的推移，开发者们迅速认识到一些常见的使用 OOP 时会犯的错误，并总结了 5 个每个 OOP 开发者都需要遵守的准则，以便更容易地创建出可维护和可扩展的系统。这 5 个准则就是 SOLID 原则。SOLID 是 Michael Feathers 提出的一组首字母缩写，它代表着下面的原则。

- **单一职责原则（SRP）**：表明软件组件（函数、类、模块）必须专注于单一的任务（只有单一的职责）。
- **开/闭原则（OCP）**：表明软件设计时必须时刻考虑到（代码）可能的发展（具有扩展性），但是程序的发展必须最少地修改已有的代码（对已有的修改封闭）。
- **里氏替换原则（LSP）**：表明只要继承的是同一个接口，程序里任意一个类都可以被其他的类替换。在替换完成后，不需要其他额外的工作程序就能像原来一样运行。
- **接口隔离原则（ISP）**：表明我们应该将那些非常大的接口（大而全的接口）拆分成一些小的更具体的接口（特定客户端接口），这样客户端就只需关心它们需要用到的接口。
- **依赖反转原则（DIP）**：表明一个方法应该遵从依赖于抽象（接口）而不是一个实例（类）的概念。

在本章，会看到如何遵循这些原则书写 TypeScript 代码来让我们的程序长久地易于维护和扩展。

类

我们应该已经熟悉了 TypeScript 类（Classes）的基础，在之前的章节中已经声明过一些类。那么让我们跟随例子来看一下关于类的细节和 OOP 的概念。从声明一个简单的类

开始：

```
class Person {
  public name : string;
  public surname : string;
  public email : string;
  constructor(name : string, surname : string, email : string){
    this.email = email;
    this.name = name;
    this.surname = surname;
  }
  greet() {
   alert("Hi!");
  }
}
var me : Person = new Person("Remo", "Jansen",
"remo.jansen@wolksoftware.com");
```

我们用类去描述一个对象或者实例。一个类由名字、属性和方法组成。上面例子中的类名为 Person，包含三个属性（name、surname 和 email）和两个方法（constructor 和 greet）。类的属性通常用来描述对象的特征，方法通常用来描述对象的行为。

constructor 是一个特殊的方法，在用 new 关键字创建一个类的实例（或者叫对象）的时候被用到。我们声明了一个变量 me，用来存储 Person 类的实例。new 关键字使用了 Person 类的 constructor 方法返回一个类型为 Person 的对象。

一个类需要遵循单一职责原则（SRP）。在上面的代码中，Person 类包含了它所有的特性和行为。现在，让我们加一些验证 email 的逻辑：

```
class Person {
  public name : string;
  public surname : string;
  public email : string;
  constructor(name : string, surname : string, email : string) {
    this.surname = surname;
    this.name = name;
    if(this.validateEmail(email)) {
      this.email = email;
    }
```

```
    else {
      throw new Error("Invalid email!");
    }
  }
  validateEmail() {
    var re = /\S+@\S+\.\S+/;
     return re.test(this.email);
  }
  greet() {
    alert("Hi! I'm " + this.name + ". You can reach me at " + this.email);
  }
}
```

当一个类不遵循 SRP，知道了太多的事情（拥有太多属性）或做了太多事情（拥有太多方法），我们称这种对象为 God 对象。这里的 Person 对象就是一个 God 对象，因为我们增加了一个和 Person 类的行为并无关联的 validateEmail 方法。

决定一个属性或方法能不能成为一个类的一部分，在某种意义上是一种主观的抉择。如果我们花一点时间分析和选择，就可以找到更好的类的设计方法。

可以通过声明一个 Email 类重构 Person 类，负责 E-mail 验证并作为 Person 类的一个属性。

```
class Email {
  public email : string;
  constructor(email : string){
    if(this.validateEmail(email)) {
      this.email = email;
    }
    else {
      throw new Error("Invalid email!");
    }
  }
  validateEmail(email : string) {
    var re = /\S+@\S+\.\S+/;
    return re.test(email);
  }
}
```

现在我们有了一个 Email 类，可以把验证 E-mail 的代码从 Person 类中移除，然后更新 email 属性，将 string 替换为 Email 对象：

```
class Person {
  public name : string;
  public surname : string;
  public email : Email;
  constructor(name : string, surname : string, email : Email){
    this.email = email;
    this.name = name;
    this.surname = surname;
  }
  greet() {
    alert("Hi!");
  }
}
```

确保每个类都只有单一的职责，可以让我们更容易地看出一个类做了哪些事情、如何去扩展/改进它，将来可以通过提高 Person 和 Email 类的抽象等级来改进它们。例如，当使用 Email 类时，我们不需要意识到其中 validateEmail 方法的存在，那么这个方法在 Email 外部就是不可见的，这样的话，Email 类就更容易理解。

当我们提升一个对象的抽象等级时，可是说成是在封装这个对象的数据和行为，封装也被称为信息隐藏。比如，Email 类允许我们使用 E-mail 而不需要关心 E-mail 验证，因为这个类已经处理了这些事情。我们可以通过访问修饰符（private 或者 public），将 Email 类上所有需要抽象的属性和方法标记为私有：

```
class Email {
  private email : string;
  constructor(email : string){
    if(this.validateEmail(email)) {
      this.email = email;
    }
    else {
      throw new Error("Invalid email!");
    }
  }
  private validateEmail(email : string) {
```

```
    var re = /\S+@\S+\.\S+/;
    return re.test(email);
  }
  get():string {
    return this.email;
  }
}
```

然后可以直接使用 Email 类而不需要考虑任何形式的验证：

```
var email = new Email("remo.jansen@wolksoftware.com");
```

接口

使用 JavaScript 开发大规模 Web 应用的时候，我们最怀念的功能就是接口（Interfaces）。我们已经看到了遵循 SOLID 原则可以帮助提高代码质量，可在大型项目中编写出更好的代码。但是问题在于，如果使用 JavaScript 时试图遵循 SOLID 原则，很快会意识到没有接口我们永远无法编写出遵循 SOLID 原则的面向对象的代码。幸运的是，TypeScript 为我们提供了接口特性。

在传统的面向对象概念中，一个类可以扩展另一个类，也可以实现一个或多个接口。一个接口可以实现一个或多个接口但是不能扩展另一个类或接口。维基百科对 OOP 中接口的定义是：

> *在面向对象的语言中，术语 interface 经常被用来定义一个不包含数据和逻辑代码但用函数签名定义了行为的抽象类型。*

实现一个接口可以被看作是签署了一份协议。接口好比是协议，当我们签署它（实现它）时，必须遵守它的规则。接口的规则是方法和属性的签名，我们必须实现它们。

接下来将在本章看到很多接口的示例。

在 TypeScript 中，接口并不是严格遵守上面提到的定义。在 TypeScript 中主要有两点不同：

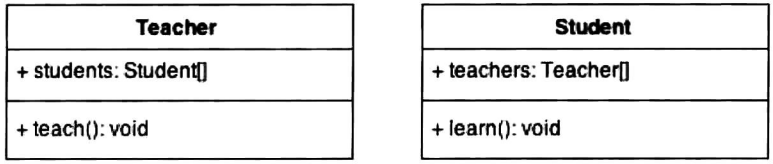
- 接口可以扩展其他接口或者类。
- 接口可以定义数据和行为而不只是行为。

关联、聚合和组合

在面向对象的概念中，类与类之间可以有一些关系。让我们看一下类与类之间的三种不同的关系。

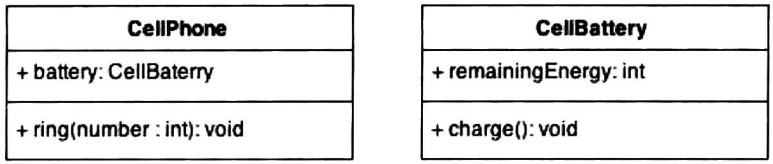
关联

那些有联系但它们的对象有独立的生命周期，并且没有从属关系的类之间的关系，我们称之为关联（Association）。让我们看一下老师与学生的例子。多个学生可以与一个老师有关联关系，一个学生也可以与多个老师有关联关系，但他们都有自己的生命周期（都可以被独立地创建和删除）。所以当老师离开学校时，不必删除任何学生，学生们离开学校时也不必删除任何老师。



聚合

我们将那些拥有独立生命周期，但是有从属关系，并且子对象不能从属于其他对象的关系称为聚合（Aggregation）。让我们看一下手机与手机电池的例子。一块单独的手机电池可以属于一台手机，但是假如一台手机停止工作了，我们将它从数据库中删掉，但电池并不需要删除，因为它可能还是可以用的。所以在聚合关系中，即使是有从属关系，但对象间依然有独立的生命周期。

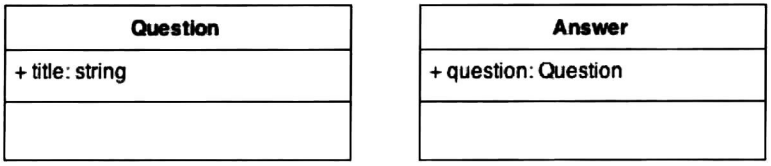


组合

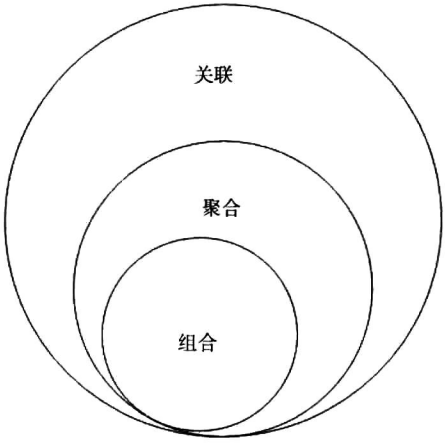
组合（Composition）是指那些没有独立生命周期，父对象被删除后子对象也被删除的对象间的关系。

让我们看一下问题与答案的例子。一个问题可以有多个答案，并且一个答案不可以属于多个问题。如果我们删除了问题，答案将会被自动删除。

生命周期依赖其他对象的对象（例子中的答案）也被称作**弱实体**。



有时候决定使用关联、聚合还是组合是一个复杂的过程。导致这一困难的原因是聚合和组合是关联的子集，这意味着它们属于关联的特殊情况。



继承

面向对象编程最重要的基本功能之一，就是可以扩展已有的类，这个功能被称为继承（Inheritance）。它允许我们创建一个类（子类），从已有的类（父类）上继承所有的属性和方法，子类可以包含父类中没有的属性和方法。回到上一个声明的类 `Person`，我们将使用 `Person` 类作为 `Teacher` 子类的父类：

```
class Person {
  public name : string;
  public surname : string;
  public email : Email;
```

```
    constructor(name : string, surname : string, email : Email){
      this.name = name;
      this.surname = surname;
      this.email = email;
    }
    greet() {
      alert("Hi!");
    }
}
```

这个示例包含在随书的代码中。

一旦有了一个父类，我们就可以使用保留字 extends 扩展它了。在接下来的例子中，我们会声明一个扩展之前已经定义了的 Person 类的 Teacher 类，这意味着 Teacher 类会继承 Person 类中所有的方法和属性：

```
class Teacher extends Person {
  teach() {
    alert("Welcome to class!");
  }
}
```

注意，我们还向 Teacher 类添加了一个新的 teach 方法。如果创建 Person 和 Teacher 的实例，会看到它们将会共享所有的属性和方法。除了 Teacher 类中的 teach 方法，这个方法只在 Teacher 类的实例中存在：

```
var teacher = new Teacher("remo", "jansen", new
Email("remo.jansen@wolksoftware.com"));

var me = new Person("remo", "jansen", new
Email("remo.jansen@wolksoftware.com"));

me.greet();
teacher.greet();
me.teach(); // 错误：'Person'不存在'teach'属性
teacher.teach();
```

有时我们希望子类能提供父类中同名方法的特殊实现。可以使用保留字 super 达到这个目的。试想一下，我们需要增加一个属性列出老师教的科目并且想通过 Teacher 类的构造函数初始化这个属性。我们可以在子类的构造函数中使用 super 关键字引用父类的构造函数，也可以在扩展已有方法的时候使用 super 关键字，比如 greet 方法。在面向对象语言中，这种允许子类提供父类已有方法的特殊实现的功能被称作*方法重写*。

```
class Teacher extends Person {
  public subjects : string[];
  constructor(name : string, surname : string, email : Email, subjects:
string[]) {
    super(name, surname, email);
    this.subjects = subjects;
  }
  greet() {
    super.greet();
    alert("I teach " + this.subjects);
  }
  teach() {
    alert("Welcome to Maths class!");
  }
}
var teacher = new Teacher("remo", "jansen", new
Email("remo.jansen@wolksoftware.com"), ["math", "physics"]);
```

可以声明一个新的类继承一个已经继承了别的类的类。在下面的代码片段中，我们将声明一个 SchoolPrincipal 类继承从 Person 类扩展而来的 Teacher 类。

```
class SchoolPrincipal extends Teacher {
   manageTeachers() {
     alert("We need to help students to get better results!");
   }
}
```

当创建一个 SchoolPrincipal 类的实例时，我们将可以访问它的父类的所有方法和属性（SchoolPrincipal、Teacher 和 Person）：

```
var principal = new SchoolPrincipal("remo", "jansen", new
```

```
Email("remo.jansen@wolksoftware.com"), ["math", "physics"]);
principal.greet();
principal.teach();
principal.manageTeachers();
```

但是，不推荐有过多层级的继承。一个位于继承树上很深层级的类开发、测试和维护，在某种程度上非常复杂。不幸的是，在不确定是否应该增加*继承树深度*（DIT）的时候，并没有一个具体的规则指导我们。

我们应该将继承用来减少程序的复杂度而不是起到相反的效果。应该保证 DIT 在 0 到 4 之间，因为大于 4 时将损害封装性并增加复杂度。

混合

有时候，我们会认为声明一个同时继承两个或多个类（被称作*多重继承*）的类是一个好想法。

让我们看一个例子，在这个例子中，我们不会为方法加入任何代码，避免因为这些方法而分心，而是将注意力集中在继承树上：

```
class Animal {
  eat() {
    // ...
  }
}
```

我们从声明一个 `Animal` 类开始，它只有一个 `eat` 方法。现在，声明两个新的类：

```
class Mammal extends Animal {
  breathe() {
    // ...
  }
}

class WingedAnimal extends Animal {
  fly(){
    // ...
  }
}
```

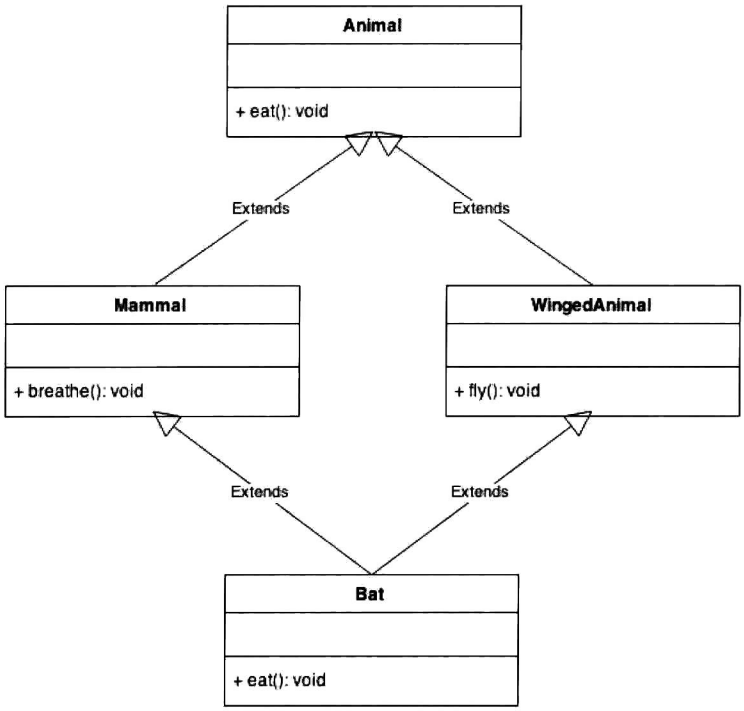
我们已经声明了两个新类，分别叫 `WingedAnimal`（飞行动物）和 `Mammal`（哺乳动物），这两个类都继承自 `Animal` 类。

现在类已经准备好了，我们将尝试着实现一个名为 `Bat`（蝙蝠）的类，蝙蝠既是哺乳动物又是飞行动物——创建一个既继承 `Mammal` 类又继承 `WingedAnimal` 类的 `Bat` 类看起来顺理成章。然而，当我们试图这么做的时候，会遇到一个编译错误：

```
// 错误：子类只能继承一个父类
class Bat extends WingedAnimal, Mammal {
  // ...
}
```

出现这个错误是因为 TypeScript 不支持多重继承，这意味着一个类只能继承自一个类。像 C# 或者 TypeScript 这样的程序语言的设计者决定不支持多重继承，是因为这个特性会潜在地增加程序的复杂性。

在一些情况下，一个类的继承图会呈现为钻石状，如下图所示。这种类型的类的继承图可能会导致我们遇到被称作*钻石问题*的设计问题。



假如我们调用只存在于继承树上其中一个类中的方法的时候，不会遇到问题:

```
var bat = new Bat();
bat.fly();
bat.eat();
bat.breathe();
```

钻石问题出现在当调用 Bat 类中的父类中都存在的方法的时候，哪一个父类的方法被调用是不清楚的或者说是存在歧义的。当增加一个 move 方法到 Mammal 类和 WingedAnimal 类上并试图在 Bat 类的实例上调用它时，我们就会遇到一个歧义调用错误。

现在我们知道了多重继承为什么会导致潜在的危险，所以我们引入了混合（Mixin）特性。混合是多重继承的替代，但是这个功能有一些限制。

让我们回到 Bat 类的例子来展示混合的用法：

```
class Mammal {
  breathe() : string {
    return "I'm alive!";
  }
}

class WingedAnimal {
  fly() : string{
    return "I can fly!";
  }
}
```

这个示例包含在随书的代码中。

这两个类在上面的代码片断中的形式和上一个例子没有太多差别。我们为 fly 和 breathe 方法加了一些逻辑，这样就可以得到一些输出帮助我们理解这个示范。还有，需要注意，这些类都不再继承 Animal 类了：

```
class Bat implements Mammal, WingedAnimal {
  breathe : () => string;
  fly : () => string;
}
```

Bat 类增加了一些重要的东西。我们使用了保留关键字 implements（与 extends 相反）表明 Bat 会实现所有 Mammal 和 WingedAnimal 类中声明的功能。还增加了所有 Bat 类会实现的函数的签名。

我们需要将下面的函数复制到代码中以便应用混合的效果：

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name =>
{
      if (name !== 'constructor') {
        derivedCtor.prototype[name] = baseCtor.prototype[name];
      }
    });
  });
}
```

上面这个函数是一个非常有名的范式，可以在很多图书和线上资料中找到，包括 TypeScript 的官方手册。

这个函数会迭代所有父类的属性（存储到 baseCtors 数组中），然后，将它们的实现复制到子类中（derivedCtor）。

只需要在程序中声明一次这个函数，然后就可以像下面这样使用它：

```
applyMixins(Bat, [Mammal, WingedAnimal]);
```

随后子类（Bat）会包含它的两个父类（WingedAnimal 和 Mammal）的所有属性和实现。

```
var bat = new Bat();
bat.breathe(); // I'm alive!
bat.fly();     // I can fly!
```

我们在本章开头说过，使用混合有一些限制。第一个限制就是只能在继承树上继承一级的方法和属性。现在可以理解我们要在应用混合前移除 Animal 类的原因了。第二个限制是，如果两个或更多的父类包含了同名的方法，那么只会继承传入 applyMixins 方法中的 baseCtors 数组中最后一个类中的该方法。我们现在来看一个呈现了这两个限制的例子。

```
class Animal {
  eat() : string {
    return "Delicious!";
  }
}
```

然后声明 Mammal 和 WingedAnimal 类，这次我们会继承 Animal 类：

```
class Mammal extends Animal {
  breathe() : string {
    return "I'm alive!";
  }
  move() : string {
    return "I can move like a mammal!";
  }
}

class WingedAnimal extends Animal {
  fly() : string{
    return "I can fly!";
  }
  move() : string {
    return "I can move like a bird!";
  }
}
```

随后我们声明 Bat 类但命名为 Bat1。这个类会实现 Mammal 和 WingedAnimal 类：

```
class Bat1 implements Mammal, WingedAnimal {
  eat : () => string;
  breathe : () => string;
  fly : () => string;
  move : () => string;
}
```

我们已经准备好调用 applyMixins 函数了。注意，在我们传入的数组里，Mammal 在 WingedAnimal 前面：

```
applyMixins(Bat1, [Mammal, WingedAnimal]);
```

现在可以创建一个 Bat1 的实例，然后可观察到，由于第一个限制，eat 方法并没有从 Animal 类上继承下来：

```
var bat1 = new Bat();
bat1.eat();     // 错误：不是一个函数
```

所有父类的方法都被正确地继承：

```
bat1.breathe(); // I'm alive!
bat1.fly();     // I can fly!"
```

除了 move 方法，由于第二条限制，只有最后一个传给 applyMixins 的父类的同名实现被继承。在这个例子中，move 的实现是继承自 WingedAnimal 类的：

```
bat1.move();    // I can move like a bird
```

为了确定这一点，我们看一下调用 applyMixins 的时候调换父类的顺序之后的结果：

```
class Bat2 implements WingedAnimal, Mammal {
  eat : () => string;
  breathe : () => string;
  fly : () => string;
  move : () => string;
}
```

注意，我们之前是在数组中将 WingedAnimal 放在 Mammal 前面的：

```
applyMixins(Bat2, [WingedAnimal, Mammal]);
var bat2 = new Bat2();
bat2.eat();     // Error: not a function
bat2.breathe(); // I'm alive!
bat2.fly();     // I can fly!
bat2.move()     // I can move like a mammal
```

范型类

在上一章中，我们看到了怎样使用范型函数，现在来看一下如何使用范型类。

如同范型函数一样，范型类能够帮助我们避免重复代码，让我们看一个例子：

```
class User {
  public name : string;
  public password : string;
}
```

这个示例包含在随书的代码中。

我们已经声明了一个 User 类，它包含了两个属性：name 和 password。我们将声明一个非范型的 NotGenericUserRepository 类。这个类通过它的构造函数接受一个 URL 并且有一个名为 getAsync 的方法。getAsync 方法通过 AJAX 请求一个用户列表并存储在一个 JSON 文件中：

```
class NotGenericUserRepository {
  private _url : string;
  constructor(url : string) {
    this._url = url;
  }

  public getAsync() {
    return Q.Promise((resolve : (users : User[]) => void, reject) => {
      $.ajax({
        url: this._url,
        type: "GET",
        dataType: "json",
        success: (data) => {
          var users = <User[]>data.items;
          resolve(users);
        },
        error: (e) => {
          reject(e);
        }
      });
    });
  }}
```

一旦声明了 NotGenericUserRepository，我们就可以创建它的实例并调用 getAsync

方法：

```
var notGenericUserRepository = new NotGenericUserRepository("./demos/
shared/users.json");
notGenericUserRepository.getAsync()
    .then(function(users : User[]){
      console.log('notGenericUserRepository => ', users);
    });
```

如果还需要请求一个不同于 User 的其他类型实例的列表，我们最终会重复写很多代码。想象一下，我们还需要请求一个会议的对话列表，会创建一个名为 Talk 的实例和一个几乎完全相同的类：

```
class Talk {
  public title : string;
  public description : string;
  public language : string;
  public url : string;
  public year : string;
}

class NotGenericTalkRepository {
  private _url : string;
  constructor(url : string) {
    this._url = url;
  }
  public getAsync() {
    return Q.Promise((resolve : (talks : Talk[]) => void, reject) => {
      $.ajax({
        url: this._url,
        type: "GET",
        dataType: "json",
        success: (data) => {
          var users talks = <Talk[]>data.items;
          resolve(userstalks);
        },
        error: (e) => {
          reject(e);
```

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

如果实例种类的数量增加，我们将会继续写重复代码。可能会想到用 any 类型避免这个问题，但是这样的话就失去了 TypeScript 在编译器中提供的类型检查的保护。更好的方式是创建一个范型类：

```
class GenericRepository<T> {
  private _url : string;
  constructor(url : string){
    this._url = url;
  }
  public getAsync() {
    return Q.Promise((resolve : (entities : T[]) => void, reject) => {
      $.ajax({
        url: this._url,
        type: "GET",
        dataType: "json",
        success: (data) => {
          var list = <T[]>data.items;
          resolve(list);
        },
        error: (e) => {
          reject(e);
        }
      });
    });
  }
}
```

这些代码除了范型以外与 NotGenericUserRepository 完全相同。我们移除了具体的 User 和 Talk 类型，用范型 T 取代它。现在可以使用一行代码并且不需要重复地声明任意多的实例：

```
var userRepository = new GenericRepository<User>("./demos/shared/
```

```
users.json");
userRepository.getAsync()
  .then((users : User[]) => {
    console.log('userRepository => ', users);
  });

var talkRepository = new GenericRepository<Talk>("./demos/shared/
talks.json");
talkRepository.getAsync()
  .then((talks : Talk[]) => {
    console.log('talkRepository => ', talks);
  });
```

范型约束

有时候，我们可能会需要约束范型类。比如上一部分例子中的范型类，我们有一个新的需求：需要增加一些变更来验证通过 AJAX 请求到的数据并且只有在验证有效后返回。

一个可行的解决方案是在范型类或函数内使用 `typeof` 操作符来验证参数范型 `T` 的类型：

```
// ...
success: (data) => {
  var list : T[];
  var items = <T[]>data.items;
  for(var i = 0; i < items.length; i++){
    if(items[i] instanceof User) {
      // validate user
    }
    if(items[i] instanceof Talk) {
      // validate talk
    }
  }
  resolve(list);
}
// ...
```

问题是我们每增加一个新的有效实例，就必须修改 `GenericRepository` 类增加额外

的逻辑。我们不会将验证逻辑加入到 GenericRepository 类中，因为一个范型类不应知道范型的类型。

一个更好的解决方案是给要获取的实例增加一个 isValid 方法，它在实例验证通过的时候返回 true：

```
// ...
success: (data) => {
  var list : T[];
  var items = <T[]>data.items;
  for(var i = 0; i < items.length; i++){
    if(items[i].isValid()) { // error
      // ...
    }
  }
  resolve(list);
}
// ...
```

第二种实现方式遵循 SOLID 原则中的开/闭原则。我们可以在 GenericRepository 正常工作的情况下增加一个新的实例（对扩展开放），但是不需要额外地修改代码（对已有的修改封闭）。这种实现的唯一问题是，如果在 GenericRepository 中尝试调用实例的 isValid 方法，我们会遇到一个编译错误。

会遇到这个错误是因为我们允许 GenericRepository 与任意类型的实体一起使用，但不是所有的类型都有 isValid 方法。幸运的是，这个问题可以通过范型约束轻易解决。范型约束会约束允许作为范型参数中的 T 的类型。我们将声明一个范型约束，这样的话只有实现了 ValidatableInterface 接口的类型可以用于这个范型方法。

让我们从声明一个接口开始：

```
interface ValidatableInterface {
  isValid() : boolean;
}
```

这个示例包含在随书的代码中。

接下来我们可以实现这个接口。在这个例子中，我们必须实现 isValid 方法：

```
class User implements ValidatableInterface {
  public name : string;
  public password : string;
  public isValid() : boolean {
    // user validation...
    return true;
  }
}

class Talk implements ValidatableInterface {
  public title : string;
  public description : string;
  public language : string;
  public url : string;
  public year : string;
  public isValid() : boolean {
    // talk validation...
    return true;
  }
}
```

现在，我们声明一个范型仓库并且为它加上范型约束，然后只有实现了 ValidatableInterface 的类型可以作为范型参数 T：

```
class GenericRepositoryWithConstraint<T extends ValidatableInterface> {
  private _url : string;
  constructor(url : string) {
    this._url = url;
  }
  public getAsync() {
    return Q.Promise((resolve : (talks : T[]) => void, reject) => {
      $.ajax({
        url: this._url,
        type: "GET",
        dataType: "json",
```

```
        success: (data) => {
          var items = <T[]>data.items;
          for(var i = 0; i < items.length; i++) {
            if(items[i].isValid()) {
              list.push(items[i]);
            }
          }
          resolve(list);
        },
        error: (e) => {
          reject(e);
        }
      });
    });
  }
}
```

即使在使用接口的时候，在上面的例子中也可以用 extends 关键字而不是 implements 关键字去声明范型约束。这样做并无特殊原因，这就是 TypeScript 范型约束语法的工作方式。

我们可以创建想要的任意多的仓库：

```
var userRepository = new
  GenericRepositoryWithConstraint<User>("./users.json");

userRepository.getAsync()
  .then(function(users : User[]){
    console.log(users);
  });

var talkRepository = new
  GenericRepositoryWithConstraint<Talk>("./talks.json");

talkRepository.getAsync()
  .then(function(talks : Talk[]){
    console.log(talks);
  });
```

如果尝试使用一个没有实现 ValidatableInterface 的类作为范型参数 T，就会得到一个编译错误。

在范型约束中使用多重类型

当声明范型约束的时候，我们只能关联一种类型。想象一下，有一个范型类需要被约束，它只允许使用实现了以下两个接口的类型：

```
interface IMyInterface {
  doSomething();
};
interface IMySecondInterface {
  doSomethingElse();
};
```

可能会想到像下面这样定义范型约束:

```
class Example<T extends IMyInterface, IMySecondInterface> {
  private genericProperty : T;
  useT() {
    this.genericProperty.doSomething();
    this.genericProperty.doSomethingElse(); // 错误
  }
}
```

然而，这段代码会抛出一个编译错误。我们不能在定义范型约束的时候指定多个类型。然而，可以通过将 IMyInterface、IMySecondInterface 转变成一个超接口来解决这个问题:

```
interface IChildInterface extends IMyInterface, IMySecondInterface {
}
```

IMyInterface 和 IMySecondInterface 现在都是超接口，因为它们都是 IChildInterface 接口的父接口，可以使用 IChildInterface 来定义范型约束:

```
class Example<T extends IChildInterface> {
  private genericProperty : T;
  useT() {
```

```
    this.genericProperty.doSomething();
    this.genericProperty.doSomethingElse();
  }
}
```

范型中的 new 操作

要通过范型代码来创建新的对象，我们需要声明范型 T 拥有构造函数。这意味着我们需要像下面这样用 type: { new(): T;}替代 type: T:

```
function factoryNotWorking<T>(): T {
  return new T(); // 找不到标识 T，编译错误
}
function factory<T>(): T {
  var type: { new(): T ;};
  return new type();
}

var myClass: MyClass = factory<MyClass>();
```

遵循 SOLID 原则

如我们之前提到过的，如果要遵循 SOLID 原则，接口是最基础的功能之一，而我们已经将 SOLID 原则的前两条付诸实践。

我们已经讨论过单一职责原则。现在，看一下其余三个原则的实例。

里氏替换原则

里氏替换原则（LSP）表示派生类对象能够替换其基类对象被使用。让我们通过一个例子进行理解。

我们将声明一个名为 PersistanceService 的类，它的作用是将一些对象持久化到某种存储中。我们以声明下面的接口开始：

```
interface PersistanceServiceInterface {
  save(entity : any) : number;
}
```

我们将在声明 PersistanceServiceInterface 后实现它，使用 cookie 作为存储程序数据的介质：

```
class CookiePersitanceService implements PersistanceServiceInterface{
  save(entity : any) : number {
    var id =  Math.floor((Math.random() * 100) + 1);
    // Cookie持久化逻辑...
    return id;
  }
}
```

我们继续声明一个名为 FavouritesController 的类，它有一个基于 PersistanceServiceInterface 的依赖：

```
class FavouritesController {
  private _persistanceService : PersistanceServiceInterface;
  constructor(persistanceService : PersistanceServiceInterface) {
    this._persistanceService = persistanceService;
  }
  public saveAsFavourite(articleId : number) {
    return this._persistanceService.save(articleId);
  }
}
```

最后可以通过向构造函数传入 CookiePersitanceService 的实例来创建一个 FavouritesController 的实例。

```
var favController = new FavouritesController(new
CookiePersitanceService());
```

LSP 允许将依赖替换成其他的实现，只要这个实现是基于同一个基类的。所以，假如我们决定不用 cookie 作为存储介质，而使用 HTML5 本地存储取代它，可以声明一个新的实现：

```
class LocalStoragePersitanceService implements
```

```
PersistanceServiceInterface{
  save(entity : any) : number {
    var id =  Math.floor((Math.random() * 100) + 1);
    // 本地存储持久化逻辑...
    return id;
  }
}
```

我们可以在不需要对 FavouritesController 控制类做任何修改的情况下用它替换：

```
var favController = new FavouritesController(new
LocalStoragePersitanceService());
```

接口隔离原则

接口被用来声明两个或更多的应用组件间是如何互相操作和交换信息的。这种声明也被称作*应用程序接口*（API）。在上一个例子中，接口是 PersistanceServiceInterface，它被 LocalStoragePersitanceService 和 CookiePersitanceService 类实现。这个接口也被 FavouritesController 类消费，因此我们说这个类是 PersistanceServiceInterface API 的客户端。

接口隔离原则（ISP）代表着任何客户端不应强制依赖于它没有使用到的方法。为了遵循 ISP，我们需要记住，在应用组件内声明 API 时，声明多个针对特定客户端的接口，要好于声明一个大而全的接口。让我们看一个例子。

如果设计一个 API，控制车辆中所有的零件（引擎、收音机、暖气、导航、灯等），可以只用一个通用的接口，它可以控制车辆中的每一个零件：

```
interface VehicleInterface {
  getSpeed() : number;
  getVehicleType: string;
  isTaxPayed() : boolean;
  isLightsOn() : boolean;
  isLightsOff() : boolean;
  startEngine() : void;
  acelerate() : number;
  stopEngine() : void;
  startRadio() : void;
```

```
  playCd : void;
  stopRadio() : void;
}
```

这个示例包含在随书的代码中。

如果一个类有一个基于 VehicleInterface 的依赖（客户端），但只想使用它的 radio 方法，我们就会违背 ISP。因为如我们已经看到的，没有任何客户端会被强制依赖它没有使用到的方法。

解决方案是将 VehicleInterface 分离成多个针对客户端的接口，这样我们的类就可以通过只依赖 RadioInterface 接口来遵循 ISP：

```
interface VehicleInterface {
  getSpeed() : number;
  getVehicleType: string;
  isTaxPayed() : boolean;
  isLightsOn() : boolean;
}

interface LightsInterface {
  isLightsOn() : boolean;
  isLightsOff() : boolean;
}

interface RadioInterface {
  startRadio() : void;
  playCd : void;
  stopRadio() : void;
}

interface EngineInterface {
  startEngine() : void;
  acelerate() : number;
  stopEngine() : void;
}
```

依赖反转原则

依赖反转原则（DIP）表示一个方法应该遵从依赖于抽象而不是一个实例。前面我们实现了 FavouritesController，并且可以不需要对 FavouritesController 做任何修改就能替换 PersistanceServiceInterface 的实现。这种行为成为可能是因为代码遵循 DIP，即：FavouritesController 依赖的是 PersistanceServiceInterface（抽象），而不是 LocalStoragePersitanceService 或 CookiePersitanceService（实现）。

如果你有相关背景，可能会想到 TypeScript 中有没有包含一些“控制反转（IoC）”的相关部分。我们确实可以在网上找到一些支持 IoC 的包。但是，由于 TypeScript 的运行时环境不支持反射或接口，这些包可以明确地被认为是冒牌的 IoC 而非真正的 IoC。如果你想了解更多控制反转的内容，强烈推荐这篇文章，Martin Fowler 写的控制反转模式，可以在 http://martinfowler.com/articles/injection.html 获取。

命名空间

TypeScript 提供了命名空间（之前已经知道了它是内部模块）特性。命名空间主要用于组织代码。

如果在写一个大型应用，在代码量增加的时候需要引入某种代码组织方案避免命名冲突，并使代码更加容易跟踪和理解。

这时可以用命名空间包裹那些有联系的接口、类和对象。比如，可以将所有的程序数据模型包含在名为 model 的内部模块中：

```
namespace app {
  export class UserModel {
    // ...
  }
}
```

当声明一个命名空间时，所有实体部分默认都是私有的，可以使用 export 关键字导出公共部分。

也可以在命名空间内声明另一个命名空间。下面来创建一个名为 models.ts 的文件，

并将这些代码加入其中：

```
namespace app {
  export namespace models {
    export class UserModel {
      // ...
      export class TalkModel {
      // ...
      }
    }
  }
}
```

在上面的例子中，我们声明了一个名为 app 的命名空间，在它的内部，又声明了一个名为 models 的公共命名空间，它有两个公共类：UserModel 和 TalkModel。随后可以在其他的 TypeScript 文件中通过指明命名空间名字进行访问：

```
var user = new app.models.UserModel();
var talk = new app.models.TalkModel();
```

如果一个内部模块变得太大，它应被分成多个文件来增加可维护性。如果我们继续上一个例子，可以通过在其他文件中引用 app 给内部模块 app 增加更多内容。

让我们新建一个名为 validation.ts 的文件并将下面这些代码加入其中：

```
namespace app {
  export namespace validation {
    export class UserValidator{
      // ...
    }
    export class TalkValidator {
      // ...
    }
  }
}
```

新建一个名为 main.ts 的文件并将下面这些代码加入其中：

```
var user = new app.models.UserModel();
var talk = new app.models.TalkModel();
```

```
var userValidator = new app.validation.UserValidator();
var talkValidator = new app.validation.TalkValidator();
```

即使命名空间 models 和 validation 在两个不同的文件中，也可以在第三个文件中同时访问到它们。

命名空间可以包含点号。比如，替代在 app 模块中使用嵌套命名空间（validation 和 models），我们可以在 validation 和 models 内部模块名上使用点号：

```
namespace app.validation {
  // ...
}
namespace app.models {
  // ...
}
```

import 关键词可以用在内部模块中，为另一个模块提供别名：

```
import TalkValidatorAlias = app.validation.TalkValidator;
var talkValidator = new TalkValidatorAlias();
```

一旦声明了命名空间，就可以决定是否将它们分别编译成 JavaScript 文件或者编译并合并成单一的 JavaScript 文件。

可以使用--out 标志将所有的输入文件编译成一个单一的 JavaScript 文件：

```
tsc --out output.js input.ts
```

编译器会根据文件内的引用在输出文件中自动整理好顺序，随后可以使用 HTML 中的<script>标签引入。

模块

TypeScript 也有外部模块或者模块的概念。模块与命名空间相比较最主要的区别是，在声明了所有的模块之后，我们不会使用<script>标签引入它们，它们通过模块加载器来加载。

*模块加载器*是在模块加载过程中为我们提供更好控制能力的工具，它能优化加载任务，比如异步加载文件或者轻松合并多个模块到单一的高度优化的文件中。

使用<script>标签的方式并不被推荐，因为当浏览器发现一个<script>标签时，它

不会使用异步请求加载这个文件。应该尽可能地尝试异步加载文件，因为这样能显著提高 Web 程序的网络性能。

我们会在第 6 章中介绍更多关于网络性能的内容。

目前的 JavaScript 版本（ES6 之前的版本）并不支持模块。开发者们被迫开发各种模块加载器，开源社区近几年正在寻找更优的解决方案。如今有好几个模块加载器可以使用，每一个都有不一样的模块定义语法，最流行的几个如下所示。

- RequireJS：RequireJS 使用了一个被称作异步模块定义的语法（AMD）。
- Browserify：Browserify 使用的语法被称为 CommonJS。
- SystemJS：SystemJS 是一个通用模块加载器，这意味着它支持所有的模块定义语法（ES6、CommonJS、AMD 和 UMD）。

Node.js 程序也使用 CommonJS 语法。

幸运的是，TypeScript 允许我们选择在运行环境中使用哪一种模块定义语法（ES6、CommonJS、AMD、SystemJS 或 UMD）。

可以在编译期使用`--module`标志来表明选择哪一种：

```
tsc --module commonjs main.ts    // 使用CommonJS
tsc --module amd main.ts         // 使用AMD
tsc --module umd main.ts         // 使用UMD
tsc --module system main.ts      // 使用SystemJS
```

与可以选择 4 种不同的运行时模块定义语法不同，在程序设计阶段只能选择两种模块定义语法：

- 外部模块语法（版本低于 1.5 的 TypeScript 的默认的模块定义语法）。
- ES6 模块语法（TypeScript 1.5 或更高版本中推荐的模块定义语法）。

重要的是，我们必须了解，可以在程序设计阶段使用一种模块定义语法（外部模块语法或 ES6），在运行时使用另一种模块定义语法（ES6、CommonJS、AMD、SystemJS 或

UMD)。

在 TypeScript 1.5 发布后，推荐使用 ES6 模块定义语法，因为它基于最新标准，并且将来可以在设计阶段和运行时都使用这种模块定义语法。

现在让我们看一下每一种模块定义语法。

ES6 模块——运行时与程序设计时

TypeScript 1.5 宣布支持 ES6 模块语法。让我们定义一个外部模块:

```
class UserModel {
  // ...
}
export { UserModel };
```

我们已经定义了一个外部模块，不需要使用 namespace 关键字，但依然要使用 export 关键字。我们在模块底部使用了 export 关键字，也可以在 class 前使用这个关键字，就像在内部模块的例子中一样：

```
export class UserModel {
  // ...
}
```

也可以通过别名输出：

```
class UserModel {
  // ...
}
export { UserModel as User }; // UserModel 输出为 User
```

一个 export 声明输出所有的同名定义：

```
interface UserModel {
  // ...
}

class UserModel {
  // ...
}
export { UserModel }; // 输出接口和函数
```

引入一个模块，我们必须这样使用 import 关键字：

```
import { UserModel } from "./models";
```

import 关键字给每一个导入的组件创建一个变量。在上面的代码中，变量 UserModel 被声明，并且值为在 models.ts 文件中声明并导出的 UserModel 类的一个引用。

可以使用 export 关键字在一个模块中导出多个实体：

```
class UserValidator {
  // ...
}

class TalkValidator {
  // ...
}

export { UserValidator, TalkValidator };
```

并且，可以使用 import 关键字从一个模块导入多个实体：

```
import { UserValidator, TalkValidator } from "./validation.ts"
```

在本书的剩余部分，我们将会在程序设计阶段使用 ES6 模块语法，在运行时使用 CommonJS 语法。

外部模块语法——仅在程序设计阶段可用

在 TypeScript 1.5 以前，模块必须使用一种叫外部模块语法的方式声明。这种语法只在程序设计阶段使用（TypeScript 代码中）。然而，一旦被编译成 JavaScript，它将会被转换成 AMD、CommonJS、UMD 或 SystemJS 模块执行。

应避免使用这种语法而应使用新的 ES6 语法来代替。但是，我们来简要看一下外部模块语法，因为可能会处理一些老的程序，或者查看一些老旧的文档。

可以使用 import 关键字来引入一个模块：

```
import User = require("./user_class");
```

上面的代码片段中声明了一个新的变量 User。User 变量将使用 user_class 模块输

出的内容作为自身的值。

要输出一个模块，需要使用 export 关键字。可以在类和接口上直接使用 export：

```
export class User {
  // ...
}
```

也可以直接使用 export 关键字自身，只需将想输出的值赋给 export：

```
class User {
  // ...
}
export = User;
```

外部模块定义可以被编译成任意可用的模块定义语法（AMD、CommonJS、SystemJS 或 UMD）。

AMD 模块定义语法——仅在运行时使用

如果将初始的外部模块语法编译成 AMD 模块（使用标识--compile amd），将生成下面的 AMD 模块：

```
define(["require", "exports"], function (require, exports) {
  var UserModel = (function () {
    function UserModel() {
    }
    return UserModel;
  })();
  return UserModel;
});
```

define 函数的第一个参数是一个数组，这个数组包含了依赖的模块名列表。第二个参数是一个回调函数，这个函数将在所有依赖全部加载完成时执行一次。回调函数将所有的依赖模块作为它的参数并且包含所有来自于 TypeScript 组件中的逻辑。注意，回调的返回类型是如何与通过 export 关键字声明为公共的组件相匹配的。AMD 模块随后可以被 RequireJS 模块加载器加载。

我们不会在本书的后面部分讨论 AMD 和 RequireJS。如果你想了解关于它们的更多信息，可以访问 http://requirejs.org/docs/start.html。

CommonJS 模块定义语法——仅在运行时使用

从将外部模块编译成 CommonJS 模块开始（使用标识--compile commonjs），我们将编译下面的代码片段：

```
class User {
  // ...
}
export = User;
```

生成了下面的 CommonJS 模块：

```
var UserModel = (function () {
  function UserModel() {
    //...
  }
  return UserModel;
})();
module.exports = UserModel;
```

正如上面这个代码片段所示，CommonJS 模块和已被弃用的（1.4 或更早版本的）TypeScript 外部模块语法相似。

上面的 CommonJS 模块不需要进行任何修改，就能被 Node.js 程序使用 import 和 require 关键字加载：

```
import UserModel = require('./UserModel');
var user = new UserModel();
```

然而，当尝试在浏览器中使用 require 关键字时，将抛出一个异常，因为 require 关键字未被定义。可以使用 Browserify 轻松解决这个问题。

我们只需执行下面三个简单的步骤：

1. 使用 npm 安装 Browserify：

```
npm install -g browserify
```

2. 使用 Browserify 将所有的 CommonJS 模块打包成一个 JavaScript 文件，这样就可以使用 HTML 的<script>标签引入它。可以通过执行下面的代码做到这一点：

```
browserify main.js -o bundle.js
```

在上面的命令中，main.js 文件包含了程序内部依赖树的根模块。bundle.js 是我们要在 HTML 的 <script> 标签中引入的文件。

3. 使用 HTML 的 <script> 标签引入 bundle.js 文件。

如果需要了解更多关于 Browserify 的信息，可以访问官方文档地址 https://github.com/substack/node-browserify#usage。

UMD 模块定义语法——仅在运行时使用

如果想发布一个 JavaScript 库或者框架，需要将 TypeScript 程序编译成 CommonJS 和 AMD 模块。我们的库同样也要让开发者能够直接在浏览器中用 HTML 的<script>标签加载。

Web 开发者社区开发出了下面的代码片段帮助我们实现通用模块定义（UMD）：

```
(function (root, factory) {
  if (typeof exports === 'object') {
    // CommonJS
    module.exports = factory(require('b'));
  } else if (typeof define === 'function' && define.amd) {
    // AMD
    define(['b'], function (b) {
      return (root.returnExportsGlobal = factory(b));
  });
  } else {
    // 全局变量
    root.returnExportsGlobal = factory(root.b);
  }
}(this, function (b) {
```

```
    // 真正的模块
    return {};
}));
```

这段代码非常棒，但是我们希望避免在每一个模块中加入这些代码。幸运的是，有一些选项可以轻松实现 UMD。

第一个选项是使用--compile umd 标识，为程序中的每一个模块生成一个 UMD 模块。第二个选项是使用模块加载器，比如 Browserify，只创建一个 UMD 模块，它包含程序中所有的模块。

访问 http://browserify.org/ 官方项目地址可了解更多关于 Browserify 的内容。根据 Browserify-standalone 选项可了解如何生成单一优化后的文件。

SystemJS 模块定义——仅在运行时使用

如同 UMD 为你提供了一种生成一个模块既兼容 AMD 又兼容 CommonJS 的方式，SystemJS 可以让你在不兼容 ES6 的浏览器上，更加贴近它们语义地使用 ES6 模块定义方式。

SystemJS 被 Angular 2.0 所使用。Angular 2.0 是一个当前很流行的 Web 应用开发框架的下一个版本。

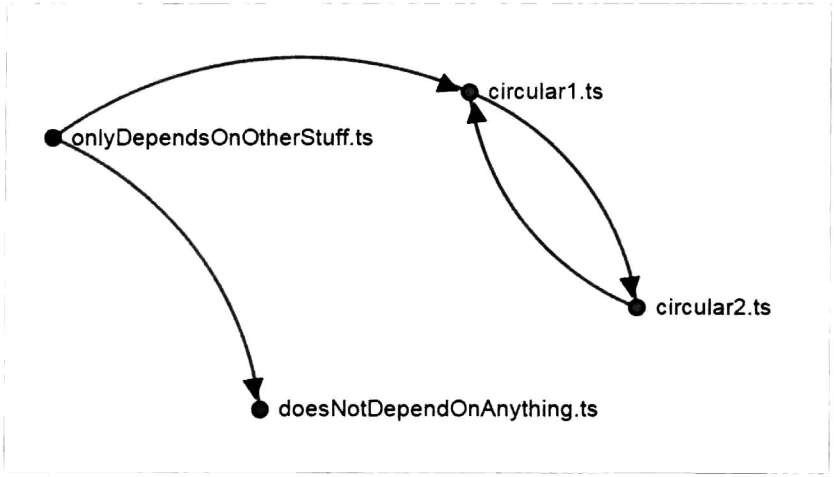
访问 SystemJS 的官方网站 https://github.com/systemjs/systemjs 可获取更多关于 SystemJS 的信息。

这里有一个免费的、关于使用模块遇到的常见错误的在线列表 http://www.typescriptlang.org/Handbook#modules-pitfalls-of-modules。

循环依赖

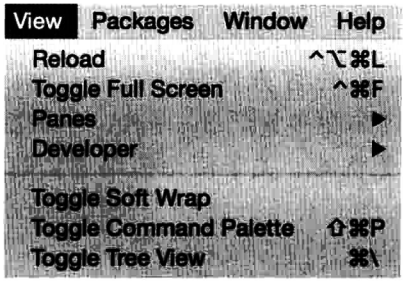
循环依赖是在处理多个模块和依赖时会遇到的问题。有时候，我们可能会遇到这样的情景，一个组件（A）依赖第二个组件（B），而第二个组件（B）又依赖第一个组件（A）。在下面的示意图中，每一个节点代表着一个组件，可以看到，节点 circular1.ts 和

`circular2.ts` 有循环依赖。名为 `doesNotDependOnAnything.ts` 的节点没有任何依赖，名为 `onlyDependsOnOtherStuff.ts` 的节点依赖 `circular1.ts`，但没有循环依赖。

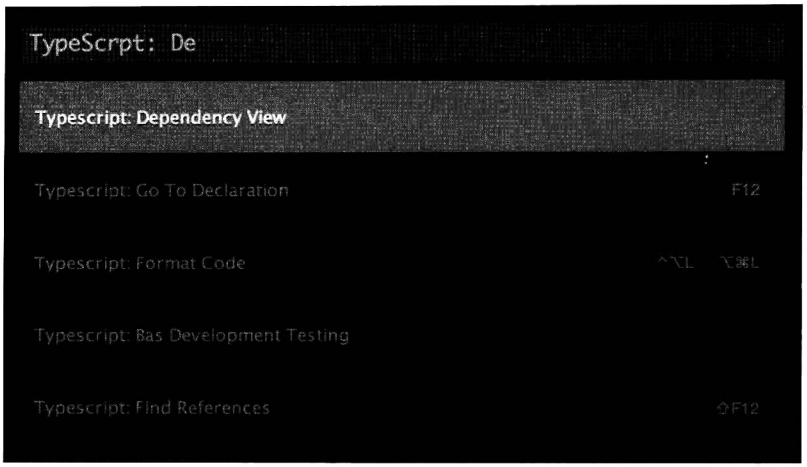


循环依赖并不一定只包含两个组件，我们可以设想一个组件依赖另一个依赖其他组件的组件，依赖树上的一些组件最终指向树上的它们的父组件。

辨认循环依赖是非常耗时的。幸运的是，Atom 内置一个命令行工具，可用来生成一个像上图那样的依赖树的图。要访问 Atom 的命令行工具，需要导航到 View（在导航条上部）菜单，然后选择 Toggle Command Palette 项。



打开 Toggle Command Palette 之后，需要输入 TypeScript: Dependency View 显示依赖树的图。



如果你想了解更多关于依赖图的信息，可以访问官方文档 `https://github.com/ TypeStrong/atom-typescript/blob/master/docs/dependency-view.md`。

小结

在这一章中，我们学习了如何深入使用类、接口和模块，学会了如何使用封装和继承等技术来简化应用的复杂度。

我们也可以使用类似 RequireJS 和 Browserify 等工具创建外部依赖，并管理依赖。

在下一章中，我们将讨论 TypeScript 运行时环境。

5

运行时

在读完本书前面部分的内容后，你应该已经跃跃欲试地想要用这些知识写一个新项目了。不过，当新项目的代码量越来越多、添加的功能越来越复杂时，你可能会遇到一些关于运行时（runtime）的问题。

对于程序设计阶段产生的问题，你应该已经能够轻松应对了。因为在本书之前的章节里，我们已经详细探讨了 TypeScript 的主要特性。

但是，我们还没有学习关于 TypeScript 运行时方面的内容。好消息是，TypeScript 的运行时就是 JavaScript 的运行时，所以你应该对它有一些了解。TypeScript 只存在于程序设计阶段。在运行前，TypeScript 代码会被编译为 JavaScript 代码，然后 JavaScript 代码被执行。再次强调，在执行时，我们执行的是 JavaScript 代码。正因如此，在讨论 TypeScript 的运行时的时候，讨论的其实是 JavaScript 的运行时。

当编译 TypeScript 代码时，将会生成 JavaScript 代码，然后在服务端（使用 Node.js）或客户端（浏览器）中执行。然后我们就可能会遇到一些扰人的运行时问题。在本章中，我们将会讨论以下话题：

- 运行时环境
- 事件循环
- `this` 操作符
- 原型
- 闭包

让我们从运行时环境开始。

环境

在开始开发一个 TypeScript 应用前，运行时环境是首先要考虑到的事情。一旦编译完 TypeScript 代码，它可能会被不同的 JavaScript 引擎执行。主要有浏览器，如 Chrome、Internet Explorer 或 Firefox，也可能是在 Node.js 或 RingoJS 环境中运行的服务端程序或桌面应用。

所以需要谨慎使用那些仅在特定运行时环境中才可用的变量。例如，我们可以创建一个库，然后访问 `document.layers` 变量。`document` 对象是 W3C 文档对象模型（DOM）标准中的一部分，而 `layers` 只是 Internet Explorer 中的特有变量，并不属于 W3C DOM 标准。

W3C 对文档对象模型有如下定义：

> 文档对象模型是用于动态访问和更新页面中视图结构的接口，它独立于平台和语言。模型的数据可以被进一步处理和修改，并且修改会反映在当前视图中。

和 DOM 类似，浏览器对象模型（BOM）也是一个仅在浏览器运行环境下特有的对象集合。BOM 包含了导航栏、历史记录、屏幕、地址栏和文档等窗口对象。

你必须明确 DOM 只是存在于浏览器中，它并不是 JavaScript 的一部分。如果在浏览器中运行应用，那么我们可以访问到 DOM 和 BOM。但是，在像 Node.js 或 RingoJS 这样的纯 JavaScript 环境中，它们是不存在的。在那些服务器端的运行环境中，也存在一些浏览器中不能访问的对象（如 Node.js 中的 `process.stdin`）。

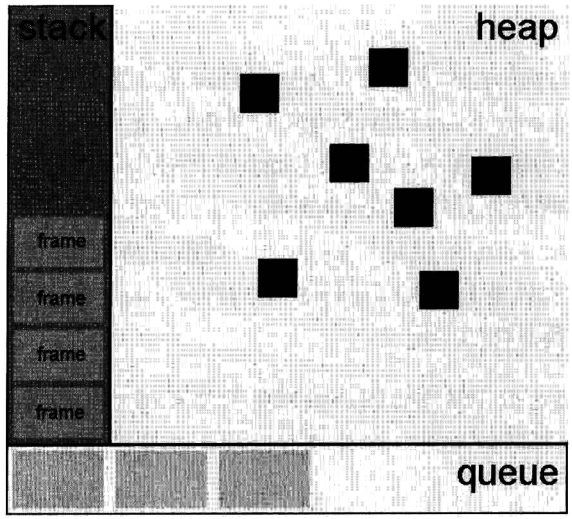
另外，还需要把同一 JavaScript 运行时的不同版本考虑在内。我们的应用经常需要兼容多个浏览器或多个版本的 Node.js。处理这个问题的一个比较好的实践是添加一个逻辑，在使用一个特性前，先检测一下这个特性是否存在。

有一个非常好用的库，可以帮助我们实现开发浏览器应用时的特性检测。它的名字叫 Modernizr，下载地址是 `http://modernizr.com/`。

运行时的一些概念

TypeScript 的运行时环境（JavaScript）有一个基于事件循环的并发模型。这个模型与其他如 C 或 Java 这样的编程语言中的模型非常不同。在讨论事件循环前，你必须了解一些其他的运行时概念。

下图为一些关键的运行时概念的模拟图：堆（heap）、栈（stack）、队列（queue）和帧（frame）：



我们将逐个探讨它们的职责。

帧

一个帧是一个连续的工作单元。在上面的图示中，帧表现为栈中的小块。

当一个 JavaScript 函数被调用时，运行时环境就会在栈中创建一个帧。帧里保存了特殊函数的参数和局部变量。当函数返回时，帧就被从栈中推出。让我们看一个例子：

```
function foo(b) {
  var a = 12;
  return a + b + 35;
}

function bar(x) {
  var m = 4;
  return foo(m * x);
}
```

在声明了 `foo` 和 `bar` 函数之后，我们执行 `bar` 函数：

```
bar(21);
```

当 `bar` 被执行时，运行时将会创建一个包含 `bar` 的参数和所有局部变量的帧。这个帧

（上图中的小正方形）被添加在了栈的顶部。

bar 函数在内部调用了 foo 函数。当 foo 函数被调用时，栈的顶部就又创建了一个新的帧。当 foo 函数执行完毕（foo 函数返回）后，栈顶部对应的帧就被移除。当 bar 函数执行完毕后，相应的帧也同样被移除。

如果在 foo 函数中又调用了 bar 函数，那么就创建了一个无限的函数调用循环。每调用一次函数，一个新的帧就被添加在栈的顶部，并且最终，栈会被完全填满，然后抛出一个错误。大多数程序员都对这个错误很熟悉，它的名字叫栈溢出错误。

栈

栈包含了一个信息在执行时的所有步骤（帧）。栈的数据结构为一个后进先出的对象集合。因此，当一个帧被加入到一个栈中时，它总是被添加在最上面。

由于栈是一个后进先出的集合，所以事件循环会从上至下地处理栈中的帧。单帧所依赖的其他帧，将会被添加在此帧的上面，以保证它从栈中可以获取到需要的依赖信息。

队列

队列中包含一个待执行信息的列表，每一个信息都与一个函数相互联系。当栈为空时，队列中的一条信息就会被取出并且被处理。处理的过程为调用该信息所关联的函数，然后将此帧添加到栈的顶部。当栈再次为空时，本次信息处理即视为结束。

在上图中，队列中的方块表示其中的信息。

堆

堆是一个内存存储空间，它不关注内部储存的内容的保存顺序。堆中保存了所有正在被使用的变量和对象。同时也保存了一些当前作用域已经不再会被用到，但还没有被垃圾回收的帧。

事件循环

并发是指同一时间有两个或更多的操作一起执行。不过由于运行时执行在一个单线程中，所以这意味着我们并不能达到真正意义上的并发。

事件循环内的信息是线性执行的。这意味着它接收到一个信息后，在处理完毕之前，不会再处理其他任何信息。

正如在第 3 章中讨论的，可以使用关键字 `yield` 和生成器函数来暂停一个函数的执行。

每当一个函数被调用，队列中就被加入一个新的信息。如果栈是空的，那么函数就会立刻执行（即表示该函数的帧被添加到了栈中）。

当所有的帧都被加入栈中后，栈便开始从上至下地一个个清除（执行）这些帧。最后，栈会被清空，然后下一个信息将会被处理。

HTML5 标准中的 web worker 可以在不同的线程内处理后台任务。它们各自拥有自己的队列、堆和栈。

使用事件循环的一个好处是，执行顺序是非常容易预测且容易追踪的。另一个好处就是，在事件循环内可以进行非阻塞 I/O 操作。这意味着当一个应用在等待 I/O 操作的执行结果时，它还可以处理其他事情，比如处理用户输入。

事件循环的一个缺点是，当一个信息需要大量时间来处理时，应用会变得无响应。一个好的做法是，保持每个信息处理尽量简短，可能的话，将一个信息函数分割为多个小函数。

this 操作符

JavaScript 中的 `this` 操作符和其他语言中的略有不同。它的值通常由它所属的函数被调用的方式来决定。它的值不能在执行时通过赋值操作来设置，并且同一个函数以不同的方式被调用，其 `this` 值也都可能会不同。

`this` 操作符在严格模式下的表现也会与在非严格模式下有所不同。更多关于严格模式的信息，请参阅 `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`。

全局上下文中的 this 操作符

在全局上下文中，this 操作符总是指向全局对象。在浏览器中，window 对象即是全局对象：

```
console.log(this === window); // true
this.a = 37;
console.log(window.a); // 37
console.log(this.document === document === window. document); // true
```

函数上下文中的 this 操作符

函数内的 this 操作符的值由函数被调用的方式所决定。在非严格模式下，如果我们简单地调用一个函数，那么内部的 this 值将指向全局对象：

```
function f1(){
  return this;
}
f1() === window; // true
```

但是，如果在严格模式中这样调用函数，那么内部的 this 值将会指向 undefined：

```
console.log(this); // 全局 (window)

function f2(){
  "use strict";
  return this; // 未定义
}
console.log(f2()); // 未定义
console.log(this); // window
```

但是，如果函数以实例的方法的形式被调用，this 操作符将会指向该实例。换句话说，调用一个类的成员方法，那么 this 操作符的值将指向该类：

```
var p = {
  age: 37,
  getAge: function() {
    return this.age; // this指向类实例(p)
  }
```

```
};
console.log(p.getAge()); // 37
```

在上面的例子里，我们使用对象字面量声明了一个名为 p 的对象，使用原型声明的方法也一样：

```
function Person() { }
Person.prototype.age = 37;
Person.prototype.getAge = function() {
  return this.age;
}
var p = new Person();
p.age;    // 37
p.getAge(); // 37
```

当一个函数被作为构造函数（使用 new 关键字）调用时，this 操作符将指向正在被构造的对象：

```
function Person() { // 将被作为构造函数调用的函数
  this.age = 37;
}
var p = new Person();
console.log(p.age); // 打印出 37
```

call、apply 和 bind 方法

所有的函数都从 Function.prototype 中继承了 call、apply 和 bind 方法。我们可以使用这些方法来设置函数内部 this 操作符的值。

call 和 apply 方法非常相似，它们都让我们先设置函数中的 this 操作符的值，并且执行这个函数。主要的区别是，apply 以数组的形式接受传递给函数的参数，而 call 则是以单个分开参数的形式。

一个简单的记忆方法是，*A（apply）*对应数组（array），而 *C（call）*对应逗号（comma）。

下面来看一个例子。我们将定义一个名为 Person 的类。这个类有两个属性（name 和 surname）和一个方法（greet）。greet 方法在内部使用了 this 操作符来访问实例的 name

和 surname 属性：

```
class Person {
  public name: string;
  public surname: string;

  constructor(name: string, surname: string) {
    this.name = name;
    this.surname = surname;
  }

  public greet(city: string, country: string) {
    // 使用this操作符来访问实例的name和surname属性
    var msg = `Hi, my name is ${this.name} ${this.surname}. `;
    msg += `I'm from ${city} (${country}).`;
    console.log(msg);
  }
}
```

在声明了 Person 类之后，创建了一个它的实例：

```
var person = new Person("remo", "jansen");
```

如果执行 greet 方法，那么它将得到预期的输出：

```
person.greet("Seville", "Spain");
// Hi, my name is remo jansen. I'm from Seville (Spain).
```

另外，也可以通过使用 call 和 apply 函数调用该方法。我们要将第一个参数指定为 person 对象，因为希望 greet 方法的内部的 this 操作符指向它：

```
person.greet.call(person, "seville", "spain");
person.greet.apply(person, ["seville", "spain"]);
```

如果将方法内部的 this 指向其他值，那么在 greet 方法内，我们将访问不到 name 和 surname 属性：

```
person.greet.call(null, "seville", "spain");
person.greet.apply(null, ["seville", "spain"]);
// Hi, my name is undefined.I'm from seville spain.
```

上面两个例子看上去没有意义，因为第一个得到了正确的结果，而第二个产生了非预期的结果。call 和 apply 方法只在我们的确希望函数内的 this 操作符指向其他值时才有意义：

```
var valueOfThis = { name : "anakin", surname : "skywalker" };
person.greet.call(valueOfThis, "mos espa", "tatooine");
person.greet.apply(valueOfThis, ["mos espa", "tatooine"]);
// Hi, my name is anakin skywalker. I'm from mos espa tatooine.
```

bind 方法也可以为我们设置（函数内的）this 操作符的值，但不执行它。

调用函数的 bind 方法时，它返回了一个和原函数具有相同函数体和作用域的新函数，但是（函数体）内部 this 操作符指向的值，已被永久地改变为传递给 bind 方法的第一个参数，不论之后这个函数如何被调用，都不会变。

让我们来看一个例子，从创建一个之前定义的 Person 类的实例开始：

```
var person = new Person("remo", "jansen");
```

然后，可以使用 bind 方法来使 greet 函数成为一个具有相同作用域和函数体的新函数：

```
var greet = person.greet.bind(person);
```

如果再次使用 bind 或 apply 方法来调用 greet 函数，会发现，无论怎么被调用，函数内部的 this 操作符的值都将不会再被改变：

```
greet.call(person, "seville", "spain");
greet.apply(person, ["seville", "spain"]);
// Hi, my name is remo jansen. I'm from seville spain.

greet.call(null, "seville", "spain");
greet.apply(null, ["seville", "spain"]);
// Hi, my name is remo jansen. I'm from seville spain.

var valueOfThis = { name: "anakin", surname: "skywalker" };
greet.call(valueOfThis, "mos espa", "tatooine");
greet.apply(valueOfThis, ["mos espa", "tatooine"]);
// Hi, my name is remo jansen. I'm from mos espa tatooine.
```

除非你明确地知道后果，否则使用 apply、call 和 bind 函数是不推荐的做法。它们可能会对其他开发者产生对运行时的困惑。

一旦使用 bind 方法为一个函数内的 this 操作符进行了绑定，就不能再次覆盖它：

```
var valueOfThis = { name: "anakin", surname: "skywalker" };
var greet = person.greet.bind(valueOfThis);
greet.call(valueOfThis, "mos espa", "tatooine");
greet.apply(valueOfThis, ["mos espa", "tatooine"]);
// Hi, my name is remo jansen. I'm from mos espa tatooine.
```

一般来说，不推荐使用 bind、apply 和 call 方法，因为它们会导致其他人的困惑。改变函数内的默认 this 操作符的值可能会导致产生未预期的结果。记住，仅在必要时才使用它们，并且为你的代码加上合适的注释，来减少潜在的可维护性问题。

原型

当我们编译 TypeScript 代码后，所有的类和对象都变为了 JavaScript 对象。有时，我们会在应用中遇到一些不符合预期的运行时问题。如果没有理解 JavaScript 的继承机制，那么就很难找到和理解问题的根源。这些知识将有助于我们更好地控制应用在运行时的行为。

运行时的继承系统使用的是原型继承模型。在一个原型继承模型中，并没有类，对象直接继承自对象。但是，可以使用原型来模拟对象。让我们看看它是如何运作的。

在运行时，几乎所有的 JavaScript 对象都有一个内部的名为 prototype 的属性。这个属性的值是一个对象，这个对象中包含了一些属性（数据）和方法（行为）。

在 TypeScript 中，我们使用基于类的继承系统：

```
class Person {
  public name: string;
  public surname: string;
  public age: number = 0;
  constructor(name: string, surname: string) {
    this.name = name;
```

```
    this.surname = surname;
  }
  greet() {
    var msg = `Hi! my name is ${this.name} ${this.surname}`;
    msg += `I'm ${this.age}`;
  }
}
```

我们声明了一个名为 Person 的类。在运行时，这个类的继承将会使用到原型：

```
var Person = (function() {
  function Person(name, surname) {
    this.age = 0;
    this.name = name;
    this.surname = surname;
  }
  Person.prototype.greet = function() {
    var msg = "Hi! my name is " + this.name +
              " " + this.surname;
    msg += "I'm " + this.age;
  };
  return Person;
})();
```

TypeScript 编译器使用一个*立即调用函数表达式*包装了一个对象声明(这里我们不将它称为类，因为它不是类)。在立即调用函数表达式的内部，我们发现了一个名为 Person 的函数。如果拿它和 TypeScript 中的类进行比较，会注意到它与 TypeScript 类的构造函数具有相同的参数。这个函数用来创建 Person 类的新实例。

在构造函数之后，我们看到了 greet 方法的定义。正如你所见，我们使用 prototype 属性来为 Person 类添加 greet 方法。

实例属性与类属性的对比

由于 JavaScript 是动态编程语言，可以在运行时为一个实例添加属性和方法，并且它们不必为对象（类）声明中的一部分。来看一个例子：

```
function Person(name, surname) {
  // 实例属性
```

```
  this.name = name;
  this.surname = surname;
}
var me = new Person("remo", "jansen");
me.email = "remo.jansen@wolksoftware.com";
```

这里，我们为一个名为 Person 的对象定义了一个构造函数，它接受两个参数（name 和 surname）。然后创建了一个 Person 类的实例，并为其添加了一个名为 email 的新属性。在运行时，可以使用 for...in 语句来检查实例的属性：

```
for (var property in me) {
  console.log("property: " + property + ", value: '" +
  me[property] + "'");
}
// property: name, value: 'remo'
// property: surname, value: 'jansen'
// property: email, value: 'remo.jansen@wolksoftware.com'
// property: greet, value: 'function (city, country) {
//    var msg = "Hi, my name is " + this.name + " " +
// this.surname;
//
//    msg += "\nI'm from " + city + " " + country;
//   console.log(msg);
// }'
```

所有这些属性都是*实例属性*，因为它们保存着每个实例各自的值。例如，如果我们创建了一个新的 Person 类的实例，那么它的属性也将保存实例自己的值：

```
var hero = new Person("John", "117");
hero.name; // "John"
me.name;   // "remo"
```

我们通过在构造函数中使用 this 操作符来定义这些属性，因为在构造函数中，this 操作符指向对象的原型。所以也可以直接在对象的 prototype 属性上定义实例属性：

```
Person.prototype.name = name;    // 实例属性
Person.prototype.name = surname; // 实例属性
```

还可以声明类的属性和方法。它们之间的主要区别在于，类的属性和方法的值是在它

的实例间共享的。类的属性和方法有时也被称为静态属性和方法。

类属性经常用来保存一些静态值：

```
function MathHelper() {
  /* ... */
}
// 类属性
MathHelper.PI = 3.14159265359;
```

类方法常常被作为工具函数使用，为其提供参数，经过计算，然后返回一个结果：

```
function MathHelper() { /* ... */ }

// 类方法
MathHelper.areaOfCircle = function(radius) {
 return radius * radius * this.PI;
}

// 类属性
MathHelper.PI = 3.14159265359;
```

在上述例子里，我们在一个类方法（areaOfCircle）内访问了类属性（PI）。可以在实例方法中访问类属性，但不可以在类属性或方法中访问实例属性或方法。可以通过将 PI 定义为实例属性来解释这种情况：

```
function MathHelper() {
  // 实例属性
  this.PI = 3.14159265359;
}
```

如果尝试在类方法中访问 PI，它将是 undefined：

```
// 类方法
MathHelper.areaOfCircle = function(radius) {
 return radius * radius * this.PI;  // this.PI为undefined
}

MathHelper.areaOfCircle(5); // NaN
```

当需要从实例方法中访问类属性或方法时，可以使用原型的 constructor 属性来取得

它们。例子如下所示：

```
function MathHelper () { /* ... */ }

// 类属性
MathHelper.PI = 3.14159265359;

// 实例方法
MathHelper.prototype.areaOfCircle = function(radius) {
  return radius * radius * this.constructor.PI;
}

var math = new MathHelper ();
console.log(MathHelper.areaOfCircle(5)); // 78.53981633975
```

由于原型的 constructor 属性返回了对象构造函数的引用，所以我们可以在 areaOfCircle（实例方法）中通过原型访问到 PI（类属性）。

在 areaOfCircle 方法中，this 操作符返回了对象原型的引用：

```
this === MathHelper.prototype //true
```

我们可以推断出 this.constructor 等于 MathHelper.prototype.constructor，因此，MathHelper.prototype.constructor 等于 MathHelper。

基于原型的继承

你可能想知道 extends 关键字是如何工作的。让我们创建一个新的 TypeScript 类，让其继承自 Person 类，来帮助你理解它：

```
class SuperHero extends Person {
  public superpower : string;
  constructor(name : string, surname : string, superpower :
  string) {
    super(name, surname);
    this.superpower = superpower;
  }
  userSuperPower() {
    return `I'm using my ${this.superpower}`
```

```
    }
}
```

上述 SuperHero 类继承自 Person 类，它有一个额外的属性（superpower）和方法（useSuperPower）。如果编译代码，在编译的结果中，以下这段代码将会引起我们的注意：

```
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
```

这段代码是由 TypeScript 生成的。尽管这只是一小段代码，但它几乎包含了本章中的所有概念，完整地理解这段代码是颇具挑战性的。为了理解它，需要将其拆分开，分别进行理解，但这样是值得的。让我们仔细看一看这个函数。

在函数第一次执行之前，this 操作符指向全局对象，在那时，它并没有一个名为__extends 的方法。这意味着那时__extends 变量的值为 undefined：

```
console.log(this.__extends); // undefined
```

当函数第一次被执行完毕后，函数表达式（即这个匿名函数）的值被保存在了全局对象的__extends 属性中：

```
console.log(this.__extends); // extends(n, e, t);
```

TypeScript 在每次检测到 extends 关键字时就会生成这个函数表达时。但是，它仅仅只会被执行一次（当__extends 变量是 undefined 时）。这个行为由第一行代码所实现：

```
var __extends = this.__extends || function (d, b) { // ...
```

当这行代码被执行后，全局变量中__extends 变量的值就被赋值为了这个匿名函数。由于我们在全局作用域中，var __extends 和 this._extends 在此刻指向的是同一个对象。

当一个新文件被执行时，它将会发现全局作用域中的__extends 变量已经可用了。这意味着函数表达式的值只会被赋值到__extends 变量一次。

正如你所知的，函数表达式的值是一个匿名函数。让我们看看它的内部：

```
function (d, b) {
```

```
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
}
```

这个函数接受两个名为 d 和 b 的参数。当调用它时，我们必须提供一个子类对象构造函数（d）和父类对象构造函数（b）。

匿名函数中的第一行遍历了父类中的所有类属性和方法，并将它复制给了子类：

```
for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
```

当使用 for...in 语句来遍历一个对象的实例时，它将迭代对象的实例属性。但是，如果使用 for...in 语句来迭代一个对象的构造函数时，它将会迭代类属性。在上面的例子里，for...in 语句被用来继承对象的类属性和方法。为了继承实例属性，我们将会复制对象的原型。

第二行声明了一个名为__的新构造函数。在它的内部，this 操作符被用来访问它的原型。

```
function __() { this.constructor = d; }
```

在原型中包含了一个名为 constructor 的特殊属性，它返回对象构造函数的引用。这个函数名为__并且此刻 this.constructor 指向的是同一个变量。紧接着，__对象的构造函数被赋值为了子类对象（d）的构造函数。

在第三行，父类对象构造函数的原型被赋值给了__对象构造函数的原型：

```
__.prototype = b.prototype;
```

在最后一行，调用了 new __()，其结果被赋值给了子类（d）的原型。经过以上步骤，我们通过执行以下代码就可完成继承：

```
var instance = new d();
```

经过上述步骤后，我们将会得到一个包含了子类（d）和父类（b）的所有属性的对象。这个新实例会如我们预期进行工作：

```
var superHero = new SuperHero();
console.log(superHero instanceof Person);    // true
console.log(superHero instanceof SuperHero); // true
```

我们可以看到编译生成的 SuperHero 类代码的做法：

```
var SuperHero = (function(_super) {
  __extends(SuperHero, _super);
  function SuperHero(name, surname, superpower) {
    _super.call(this, name, surname);
    this.superpower = superpower;
  }
  SuperHero.prototype.userSuperPower = function() {
    return "I'm using my " + superpower;
  };
  return SuperHero;
})(Person);
```

可再次看到这个立即调用函数表达式。这次，这个立即调用函数表达式接受一个 Person 对象的构造函数作为参数。在函数内部，使用_super 参数来表示它。之后，SuperHero（子类）和_super（父类）作为参数传入了__extends 函数。

在后面，我们看到了 SuperHero 对象的构造函数和 useSuperPower 函数。之所以可以在 SuperHero 函数定义前将其作为参数传入__extend 函数，是因为函数声明会被提升至作用域的顶部。

函数表达式不会被提升。当将函数赋值给一个变量时，变量名会被提升，但它的值（函数本身）不会被提升。

在 SuperHero 构造函数的内部，父类（Person）的构造函数通过 call 方法被调用：

```
_super.call(this, name, surname);
```

正如在本章前面所说的，可以通过 call 方法设置函数内 this 操作符的值。在本例中，我们将父类构造函数中的 this 操作符指向了将被创建的 SuperHero 实例，然后调用它：

```
function Person(name, surname) {
  // this指向将被创建的SuperHero实例
  this.name = name;
```

```
    this.surname = surname;
}
```

原型链

当我们尝试访问一个对象的属性或方法时，运行时将会搜索对象自身原型上的属性和方法，如果没有找到，那么运行时将会继续沿着对象的继承树，在被继承的对象的原型中继续搜索。由于父类对象通过原型链接了子类，我们称这种继承树为原型链。

来看一个小例子，将声明两个简单的名为 `Base` 和 `Derived` 的 TypeScript 类：

```
class Base {
  public method1(){ return 1; };
  public method2(){ return 2; };
}

class Derived extends Base {
  public method2(){ return 3; };
  public method3(){ return 4; };
}
```

现在，来看看生成的 JavaScript 代码：

```
var Base = (function() {
  function Base() {
  }
  Base.prototype.method1 = function() { return 1; };
  Base.prototype.method2 = function() { return 2; };
  return Base;
})();

var Derived = (function(_super) {
  __extends(Derived, _super);
  function Derived() {
    _super.apply(this, arguments);
  }
  Derived.prototype.method2 = function() { return 3; };
  Derived.prototype.method3 = function() { return 4; };
  return Derived;
```

```
})(Base);
```

然后可以创建 Derived 类的实例：

```
var derived = new Derived();
```

如果尝试访问 method1 方法，那么运行时将会在实例自己的原型上找到它：

```
console.log(derived.method1()); // 1
```

这个实例的原型上还有一个名为 method2 的方法（返回 3），但是它还继承了一个同名方法（返回 2）。当访问这个方法时，访问到的将是对象自己原型上的那个方法（返回 3）。这被称为原型遮蔽（property shadowing）：

```
console.log(derived.method2()); // 3
```

实例自己并没有一个名为 method3 的方法，但它的原型链上有一个名为 method3 的方法：

```
console.log(derived.method3()); // 4
```

在实例自身和其原型链上，都不存在一个名为 method4 的方法：

```
console.log(derived.method4()); // 报错
```

访问对象的原型

原型可以通过以下三种方式进行访问。

- Person.prototype：可以直接使用 prototype 属性来访问原型。
- Person.getPrototypeOf(person)：可以使用 getPrototypeOf 函数来访问一个实例对象的原型。
- person.__proto__：这个属性暴露对象内部可访问的原型属性。

对于__proto__属性的使用是有争议的，并且它已经不再被推荐使用。它从来没有在 ECMAScript 语言标准中出现过，但是现代浏览器中都实现了它。如今，__proto__属性已经被收入 ECMAScript 6 语言标准中，在未来会被正式支持。但访问它的效率十分低，如果考虑到应用的性能的话，也要避免使用它。

new 操作符

我们可以使用 new 操作符来生成一个 Person 类的实例：

```
var person = new Person("remo", "jansen");
```

运行时不会遵循基于类的继承模型。当使用 new 操作符时，运行时会创建一个新对象，并让其继承自 Person 类的原型。

我们可以认为运行时（JavaScript）的 new 操作符的行为和程序设计阶段（TypeScript）的 extends 关键字的行为没有区别。

闭包

闭包是最强大的运行时特性之一，但同时它也是被误解得最多的。Mozilla 开发者文档对闭包有如下定义：

> 闭包是指向独立变量的函数。换句话说，在闭包中定义的函数会“记住”它创建时的环境。

可以将独立变量理解为在我们创建的字面作用域上持续存在的变量。让我们来看一个例子：

```
function makeArmy() {
  var shooters = []
  for (var i = 0; i < 10; i++) {
    var shooter = function() { // 一个shooter即一个函数
      alert(i) // 将会弹出一个数字
    }

    shooters.push(shooter)
  }
  return shooters;
}
```

我们声明了一个名为 makeArmy 的函数。在函数内部，创建一个名为 shooters 的数组，其元素都是函数。shooters 数组中的每一个函数都将弹出一个数字，数字值为储存在函数内部的 i 值。现在调用一下这个 makeArmy 函数：

```
var army = makeArmy();
```

现在，army 变量包含了一个函数 shooters 的数组，但是当执行其中的函数时，会发现问题：

```
army[0](); // 10 （期望值为0）
army[5](); // 10 （期望值为5）
```

之所以上述代码没有按预期工作，是因为我们犯了一个普遍的关于闭包的错误。当在 makeArmy 函数中声明一个 shooter 函数时，在无意识中，我们创建了一个闭包。

造成上述情况的原因是赋值给 shooter 的函数都是闭包。它们包含自身的函数定义并保存了 makeArmy 的作用域环境。创建出了 10 个闭包，但它们共享了同一个环境。当 shooter 函数被执行时，函数中的循环已经执行完毕，i 变量的值也变成 10。

一个解决的办法是使用更多的闭包：

```
function makeArmy() {
  var shooters = []
  for (var i = 0; i < 10; i++) {
    (function(i) {
      var shooter = function() {
        alert(i);
      }
      shooters.push(shooter)
    })(i);
  }
  return shooters;
}

var army = makeArmy();
army[0](); // 0
army[5](); // 5
```

上述代码的运行符合预期。我们不再简单地共享同一个环境，而是使用一个自执行函数创建一个新的作用域环境，在其中，i 指向的是对应的期望值。

闭包和静态变量

在前文中我们看到，当一个变量在一个闭包环境中被声明后，它可以在一个类的不同实例间共享，换句话说，这个变量表现得像一个静态变量。

我们将会看到如何创建表现得像静态变量的属性和方法，先来创建一个名为 Counter 的 TypeScript 类：

```
class Counter {
  private static _COUNTER = 0;
  constructor() { }
  private _changeBy(val) {
    Counter._COUNTER += val;
  }
  public increment() {
    this._changeBy(1);
  }
  public decrement() {
    this._changeBy(-1);
  }
  public value() {
    return Counter._COUNTER;
  }
}
```

上述类中包含了一个名为_COUNTER 的静态成员。TypeScript 编译器将会生成如下代码：

```
var Counter = (function() {
  function Counter() {
  }
  Counter.prototype._changeBy = function(val) {
    Counter._COUNTER += val;
  };
  Counter.prototype.increment = function() {
    this._changeBy(1);
  };
  Counter.prototype.decrement = function() {
    this._changeBy(-1);
  };
```

```
    Counter.prototype.value = function() {
      return Counter._COUNTER;
    };
    Counter._COUNTER = 0;
    return Counter;
  })();
```

正如你所见，静态变量被 TypeScript 编译器声明为了类属性（而不是实例属性）。编译器之所以使用类属性是因为它被所有实例共享。

或者，我们也可以使用 JavaScript 代码（所有合法的 JavaScript 代码都是合法的 TypeScript 代码）用闭包来模拟静态属性：

```
var Counter = (function() {
  // 闭包上下文
  var _COUNTER = 0;

  function changeBy(val) {
    _COUNTER += val;
  }

  function Counter() { };

  Counter.prototype.increment = function() {
    changeBy(1);
  };
  Counter.prototype.decrement = function() {
    changeBy(-1);
  };
  Counter.prototype.value = function() {
    return _COUNTER;
  };
  return Counter;
})();
```

上面的代码声明了一个名为 Counter 的类。这个类包含一些方法来增大、减小或读取 _COUNTER 变量的值。_COUNTER 变量本身并不是对象原型的一部分。

Counter 构造函数自身就是一个闭包的一部分。所以，所有 Counter 类的实例都共享

同一个闭包上下文，这意味着这个上下文中的 counter 变量和 changeBy 函数将会表现得如单例一样。

单例模式使一个对象可以像静态变量一样被声明，避免了每次使用都创建它的实例。因此这个对象的实例被应用中的所有组件所共享。单例模式常在一些如每一个类的实例都不必是相互唯一、设置应用的全局状态等场景下使用。

所以，你已经知道了可以使用闭包来模拟静态变量：

```
var counter1 = new Counter();
var counter2 = new Counter();
console.log(counter1.value()); // 0
console.log(counter2.value()); // 0
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2
console.log(counter2.value()); // 2（期望值为0）
counter1.decrement();
console.log(counter1.value()); // 1
console.log(counter2.value()); // 1（期望值为0）
```

闭包和私有成员

我们已经看到闭包函数可以访问到在创建的字面作用域上持续存在的变量。这些变量并不是函数原型或函数体中的一部分，但它们是闭包上下文中的一部分。

由于我们不可以直接访问闭包上下文，那么该上下文中的变量和成员就可以用来模拟私有成员。使用闭包来模拟私有成员（而不是使用 TypeScript 的 private 描述符）的主要好处是闭包会在运行时阻止对它们的访问。

TypeScript 在运行时并不会模拟私有变量。如果我们试图访问一个私有成员，那么 TypeScript 编译器会在编译时抛出一个错误。

TypeScript 由于性能原因，并没有在运行时使用闭包模拟私有成员。不管我们添加和删除 private 描述符，生成的 JavaScript 代码都不会有变化。这意味着在运行时，私有成员变成了公开成员。

但是，使用闭包在运行时模拟私有变量是可行的。就像我们使用闭包来模拟静态变量一样，也可以使用纯 JavaScript 来完成成员的可见性控制。来看一个例子：

```
function makeCounter() {
  // 闭包上下文
  var _COUNTER = 0;
  function changeBy(val) {
    _COUNTER += val;
  }

  function Counter() { };

  Counter.prototype.increment = function() {
    changeBy(1);
  };
  Counter.prototype.decrement = function() {
    changeBy(-1);
  };
  Counter.prototype.value = function() {
    return _COUNTER;
  };
  return new Counter();
};
```

上述代码和我们之前使用闭包模拟静态变量时的代码十分类似。

此时，当调用 makeCounter 函数时，一个新的闭包上下文即被创建，所以每个新的实例都有独立的上下文（counter 变量和 changeBy 方法）：

```
var counter1 = makeCounter();
var counter2 = makeCounter();
console.log(counter1.value()); // 0
console.log(counter2.value()); // 0
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2
console.log(counter2.value()); // 0 (预期值为0)
counter1.decrement();
```

```
console.log(counter1.value()); // 1
console.log(counter2.value()); // 0 (预期值为0)
```

由于闭包上下文不能直接被访问到，我们可以说变量 counter 和函数 changeBy 是私有成员：

```
console.log(counter1.counter); // undefined
counter1.changeBy(2); // changeBy并不是一个函数
console.log(counter1.value()); // 1
```

小结

在本章中，我们讨论了如何理解运行时，这不但可以解决运行时问题，也可以让我们写出更好的 TypeScript 代码。对闭包的深入理解可以让你写出一些不了解这些知识就无法实现的复杂特性。

在下一章，我们将会讨论性能、内存管理和异常处理。

6

应用性能

在本章中，我们将学习如何以高效的方式来管理可用资源，从而实现高性能的表现。我们还会了解不同类型的资源、影响性能的因素及自动化性能分析技术。

本章首先会介绍一些诸如延迟或带宽等核心的性能概念，之后会展示在自动构建过程中如何测量并监视性能。

正如在前面的章节中讨论过的，我们可以使用 TypeScript 来生成能在诸多不同环境中执行的 JavaScript 代码（如 Web 浏览器、Node.js、移动设备等）。在本章中，我们将探索主要适用于 Web 应用程序开发的性能优化。本章包含以下内容：

- 性能与资源
- 性能的各个方面
- 内存性能分析
- 网络性能分析
- CPU 与 GPU 的性能分析
- 性能测试
- 性能优化建议
- 性能自动化测试

准备工作

在开始之前，需要先安装 Google Chrome 浏览器。之后，我们将使用它的开发者工具进行网络性能分析。

性能和资源

在着手进行性能的分析、监控和自动化测试之前，我们必须先花点时间了解一些有关性能的核心要点及概念。

一个好的应用程序应该集功能性、可靠性、可用性、可复用性、效率、可维护性和可移植性等理想特性于一身。到目前为止，我们主要着重于与可靠性和可维护性紧密相关的性能问题。

性能是指相对于时间及资源消耗所能完成的任务量。资源是指有限的物理（CPU、内存、GPU、硬盘等）或虚拟（CPU 时间、内存区、文件等）组件。由于资源是有限的，每个资源会在进程间共享。当一个资源被一个进程使用完毕，只有当资源被该进程释放后，其他进程才可使用它。高效管理可用资源，将有助于减少其他进程等待资源可用时所消耗的时间。

当开发 Web 应用时，我们需要了解以下资源是有限的。

- **中央处理单元（CPU）：** 通过执行由指令所指定的基本的算术、逻辑、控制和输入/输出（I/O）操作来执行计算机的指令。
- **图形处理单元（GPU）：** 这是一个为了加速图像在显示用帧缓冲器中的创建，而专门用来操作和变更存储器数据的处理器。当创建一个使用 WebGL API 或 CSS3 动画的应用时，我们就是在使用 GPU。
- **内存（RAM）：** 它使得数据项能够无视访问顺序地进行时间大致相同的读出和写入。当我们声明一个变量时，它将被存储在内存中。当变量离开作用域后，它将被垃圾收集器从内存中移除。
- **硬盘（HDD）和固态硬盘（SSD）：** 这两者都是用来存储和检索信息的存储设备。当开发 Web 应用程序的客户端时，我们不必太担心这类资源，因为应用程序通常不会广泛而持久地使用数据存储空间。但是应记住，当使用持久的存储方式时（cookie、本地存储、索引数据库等），应用程序的性能会受到 HDD 或 SSD 可用性的影响。
- **网络吞吐量：** 这决定了在单位时间内通过网络可发送的数据量。网络吞吐量是由诸如网络延迟或带宽等因素决定的。我们将在本章的稍后部分更多地讨论这些因素。

所有在之前列表中出现的资源在使用于 Node.js 应用或混合应用时也都是有限的。虽然 Node.js 应用并不常广泛使用 GPU，但也存在例外情况。

性能指标

由于性能受到多种物理和虚拟设备可用性的影响，我们能找到许多种不同的性能指标（用来衡量性能的因素）。流行的性能指标包括可用性、响应时间、处理速度、延迟、带宽和可伸缩性。这些测量机制通常与一类或多类前一节提到的资源直接相关（CPU 和网络吞吐量等）。现在，我们逐个细看这些性能指标。

可用性

一个系统的可用性与它的性能相关，如果系统的某些部分不可用，我们会将其评定为性能低下。可用性可以通过提高系统的可靠性、可维护性和可测试性来改进。如果系统易于测试和维护，那么增强它的可用性也将变得简单。

响应时间

响应时间是对一个服务请求做出响应所需要的时间。这里的服务不是指一个 Web 服务；此处的服务可指代任意的任务单元。响应时间可分为 3 个部分。

- **等待时间**：这一时间是指该请求等待其他较早产生的请求完成所花的时间。
- **服务时间**：这一时间是指完成服务（工作单位）所消耗的时间。
- **传输时间**：一旦该任务单元已经完成，该响应将会被发送回请求者。传输响应所消耗的时间被称为传输时间。

处理速度

处理速度（也被称为时钟速率）是指处理单元（CPU 或 GPU）的运行频率。一个应用程序包含很多任务单元。每个任务单元由许多处理器指令组成；通常处理器每个时钟周期可执行一条指令。由于一个操作需要多个时钟周期才能完成，时钟频率（处理速度）越快，在相同时间内完成的指令数目越多。

延迟

延迟这一概念可应用于系统的很多范畴中；但在 Web 应用范畴中使用时，我们用这个词来专指网络延迟。网络延迟可指代任意在网络数据通信中发生的延迟。

高延迟会在通信中产生瓶颈，从而限制带宽。基于延迟产生的原因，延迟对于网络带

宽的影响可以是临时抑或持久的。高延迟可由传输介质（电缆或无线信号）、路由及网关，或反病毒程序等问题导致。

带宽

与延迟的情况相似，本章中提到的带宽专指网络带宽。带宽，或数据传输速率，是指一定时间内从一点到另一点可以传输的数据量。网络带宽通常以比特每秒表示。

网络性能受到诸多因素的影响，而其中的一些因素会降低网络吞吐量。例如，高丢包率、高延迟以及网络的不稳定会降低网络吞吐量，而高带宽则可提高网络吞吐量。

可伸缩性

可伸缩性是系统处理更大规模任务的能力。可伸缩性良好的系统应该能够通过一些诸如峰值测试或压力测试等的性能测试。

在本章后面的部分，我们将更深入地了解性能测试（如峰值测试和压力测试）。

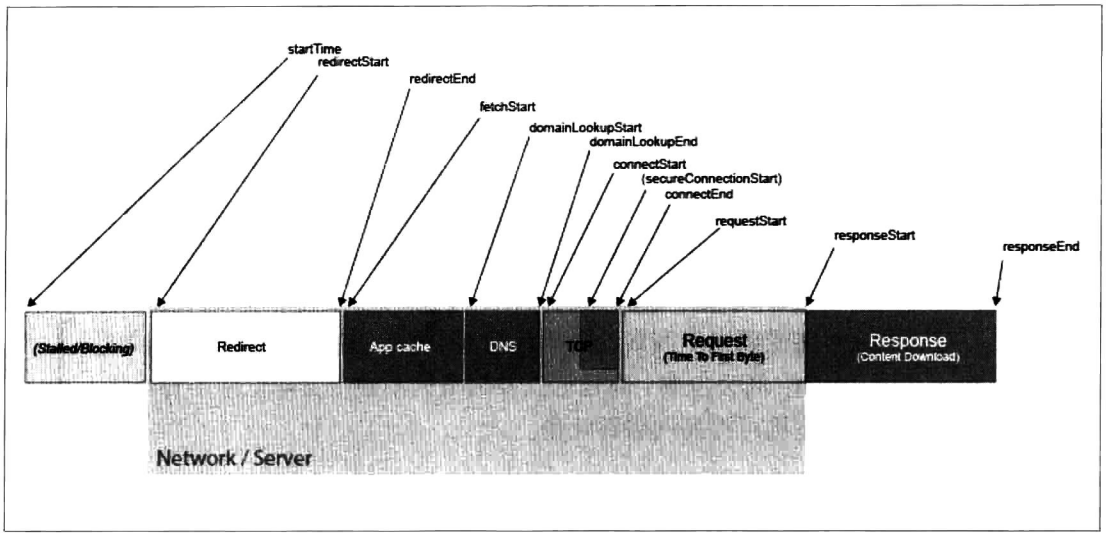
性能分析

性能分析（也被称为性能剖析）是指对于应用的资源使用状况的观察和研究。我们通过分析来确定应用程序中性能方面的问题。通过使用特定的工具，不同种类的性能分析被应用在每种资源上。现在，我们来看看如何使用 Google 的 Chrome 浏览器的开发者工具来进行网络性能分析。

网络性能分析

首先，我们来分析一下网络性能。以前，为了能够分析一个应用的网络性能，我们不得不自己写一个小型的网络日志应用。今天，有了性能计时 API（`http://www.w3.org/TR/resource-timing/`），我们的工作得以轻松许多。性能计时 API 能够访问到每个加载资源的详细网络计时数据。

下图展示了该 API 可提供的网络计时数据点：



可以通过下面的全局对象来访问性能计时 API：

```
window.performance
```

该全局对象的 performance 属性对象具有一些属性(memory、navigation 和 timing) 和方法 (clearMarks、clearMeasures 和 getEntries)。我们可以使用 getEntries 方法来得到一个包含每一个请求的计时数据点的数组：

```
window.performance.getEntries()
```

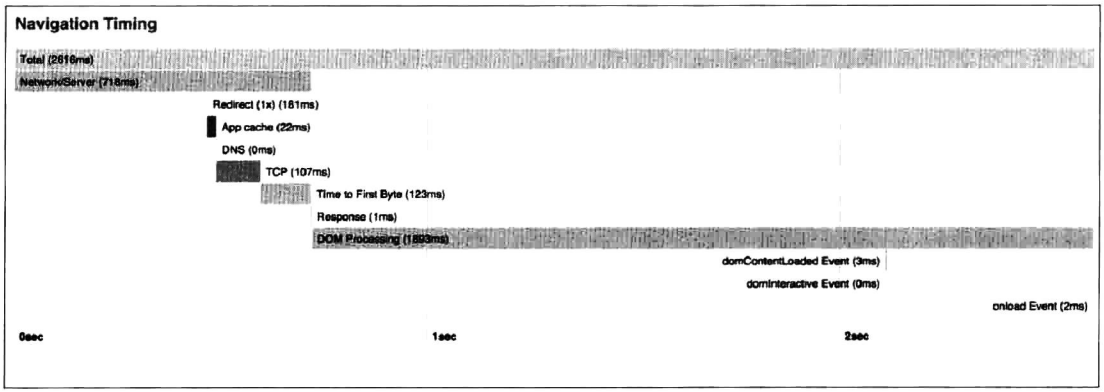
数组内的每一个元素都是 PerformanceResourceTiming 的实例，它包含了下面的信息：

```
{
  connectEnd: 1354.525000002468
  connectStart: 1354.525000002468
  domainLookupEnd: 1354.525000002468
  domainLookupStart: 1354.525000002468
  duration: 179.89400000078604
  entryType: "resource"
  fetchStart: 1354.525000002468
  initiatorType: "link"
  name: "https://developer.chrome.com/static/css/out/site.css"
  redirectEnd: 0
  redirectStart: 0
```

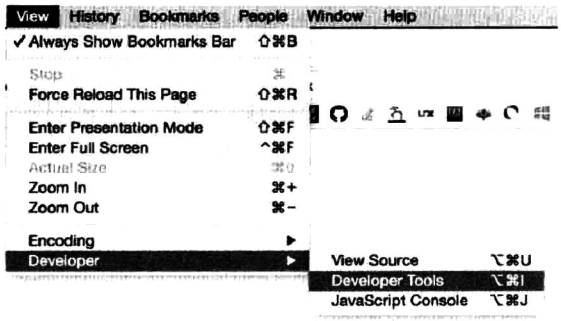
```
  requestStart: 1380.8379999827594
  responseEnd: 1534.419000003254
  responseStart: 1533.6550000065472
  secureConnectionStart: 0
  startTime: 1354.525000002468
}
```

不幸的是，上面这种格式的时间信息可能不能真正派上用场，但有一些工具可以帮助我们轻松分析它们。第一个工具是一个叫作 performance-bookmarklet 的浏览器插件。这个插件是开源的，可以用在 Chrome 和 Firefox 浏览器中。这个插件的下载链接可以在 `https://github.com/micmro/performance-bookmarklet` 找到。

在下面的截图中，你可以看到这个插件生成的图表。这个图表以更好的方式显示了性能计时 API 获取的数据，这就让我们能够轻松地找出性能问题：



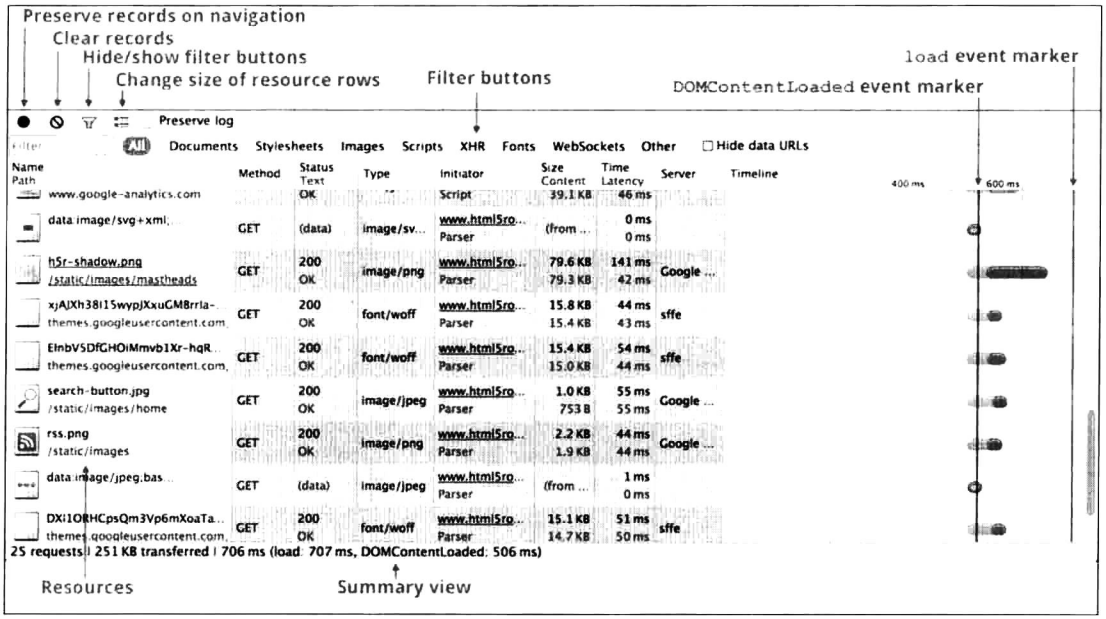
或者，你也可以使用 Chrome 开发者工具来进行网络性能测试。要访问网络面板，打开 View 菜单，选择 Developer 项中的 Developer Tools 项：



Windows 开发者可以使用 F12 键进入开发者工具。OS X 用户可以使用 Alt+Cmd+I 快捷键进入。

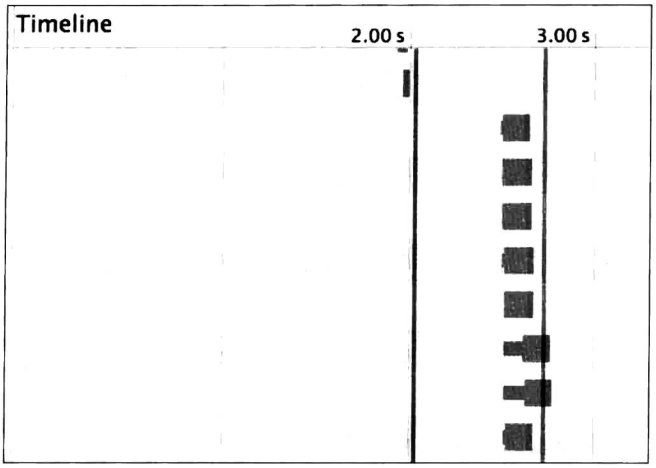
一旦进入开发者工具，可以单击 Network 按钮来访问它：

单击了 Network 按钮之后会带你进入类似下面的界面：



可以观察到，信息被展现在一个表格中，每一行是一个文件的加载信息。在右边，可以看到其中一列是时间轴。时间轴将性能计时 API 获取的数据像 performance-bookmarklet 那样进行展示。

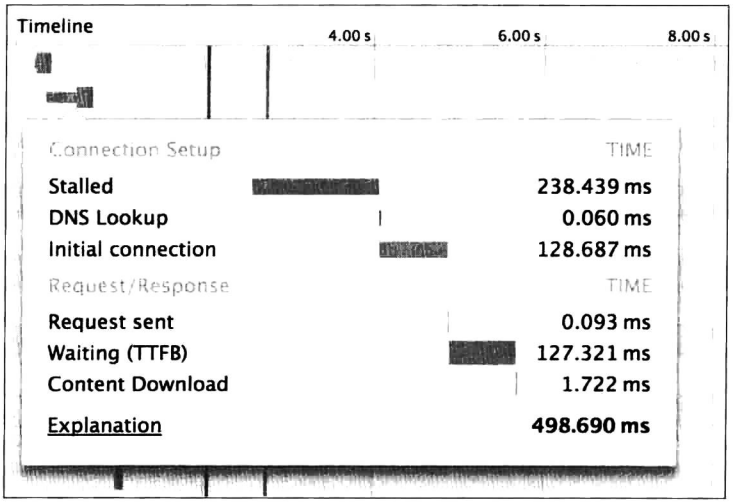
时间轴中两个重要的元素是红色和蓝色的线。这些线让我们知道 `DOMContentLoaded` 事件什么时候触发（蓝线），接下来是 `load` 事件什么时候被触发（红线）：



这两个事件非常重要，因为可以测试哪些请求在事件发生时完成，然后得知事件发生时用户可用的内容：

- `DOMContentLoaded` 事件在引擎解析完文档之后发生。
- `load` 事件在页面的资源加载完成后触发。

如果你将鼠标光标悬浮在 timing 列，可以看到每一个性能计时 API 的数据点：



事实上，开发者工具是通过性能计时 API 来读取这些信息的，我们来看看每个数据点的意义吧！

性能计时 API 数据点	描述
Stalled/Blocking	请求发送前的等待时间，开启 TCP 连接存在最长时限。达到时限后，一些请求会显示 blocking 时间而不是 stalled 时间
Proxy Negotiation	与代理服务器协商连接的时间
DNS Lookup	解析 DNS 地址的时间，每次域名解析都需要与 DNS 服务器建立双向通信
Initial Connection / Connecting	建立连接的时间
SSL	建立 SSL 连接的时间
Request Sent / Sending	网络请求所需时间，基本小于 1 毫秒
Waiting (TTFB)	请求发起到从服务器接收第一个字节的时间，又称首字节时间，从中可看出与服务器双向通信时除了等待服务器响应外的延时
Content Download / Downloading	接收服务器响应的数据的时间

网络性能与用户体验

到这里，我们已经知道如何分析网络性能了，是时候制定性能目标了。多项研究证明减少加载时间是多么重要，Akamai 于 2009 年 9 月发布的研究中，调查了 1048 名网购者，发现：

- 47%的人希望网页加载时间在 2 秒以内。
- 一旦加载时间超过 3 秒，40%的人会关闭网页。
- 52%的人声称他们对加载快的网站黏性更高。
- 如果一个购物网站加载得很慢，14%的人表示会选择去其他网站购物，23%的人表示不想再网购甚至关掉电脑。
- 64%的人表示不满意的网站下次再也不想再逛。

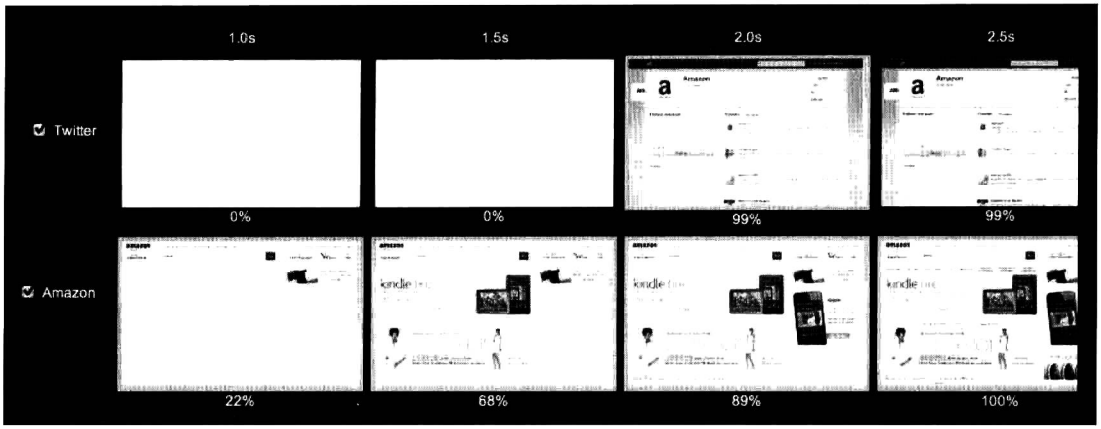
可以通过这个链接访问 Akamai 的研究：`http://www.akamai.com/html/about/press/releases/2009/press_091409.html`。

从以上结论中可以看出，我们应该意识到网络性能多么重要，因此首要目标是提高加载速度。

一旦努力将网站的加载时间控制在 2 秒内，我们就可能会犯一个常见的错误：在 2 秒内触发 `onLoad` 事件。

虽然尽快触发 `onLoad` 事件可以提高网络性能，但这并不意味着用户体验的提高。

onLoad 事件不足以决定网络性能，我们可以比较一下 Twitter 和 Amazon 的加载情况，下方截屏中显示，Amazon 比 Twitter 更快接触用户，虽然它们的 onLoad 事件是相同的，但是用户体验差别很大。



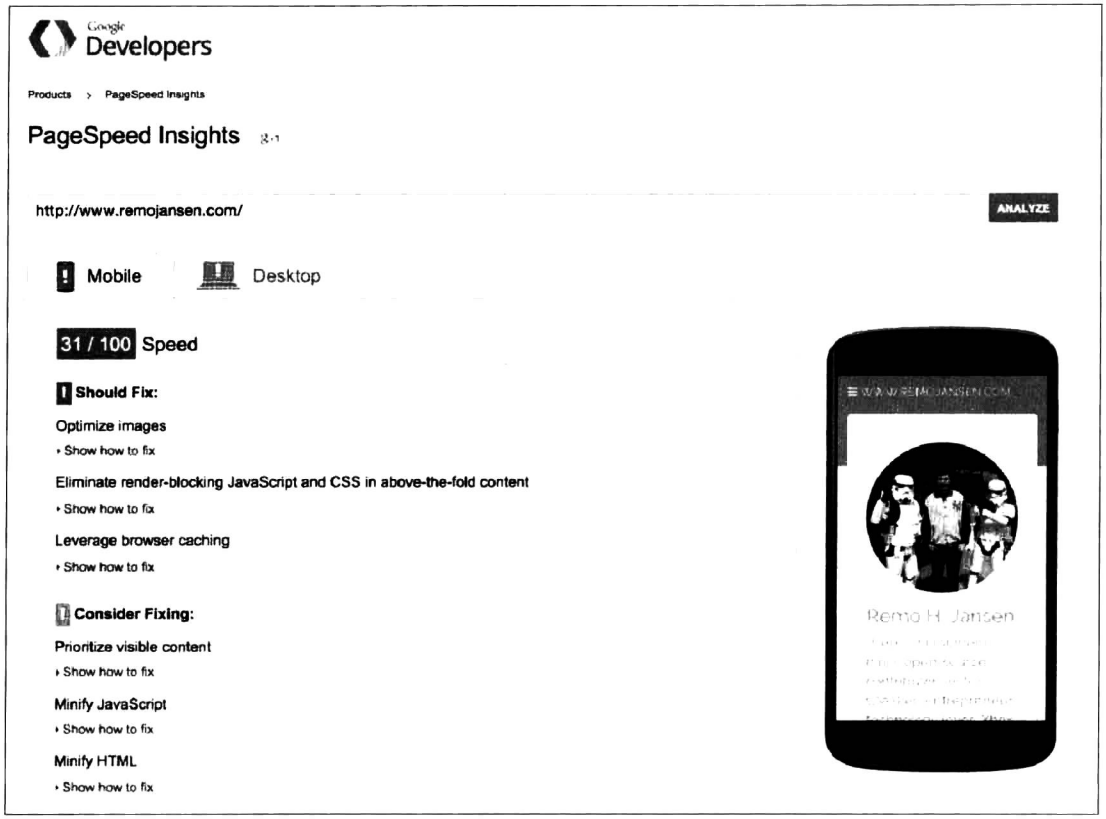
从这个示例可以看出，为提高用户体验，不仅要提高加载速度，还要让用户能尽快看到网页内容。为了实现这一点，应异步加载次要的内容。

可前往第 3 章学习更多关于 TypeScript 异步函数的知识。

网络性能最佳实践和规则

另外一种分析 Web 应用性能的方式是，通过使用最佳实践的网络性能工具，比如 Google PageSpeed Insights 和 Yahoo YSlow。

Google PageSpeed Insights 有在线版和 Chrome 浏览器插件，通过 https://developers.google.com/speed/pagespeed/insights/ 可访问在线版，填入想分析的 Web 应用的 URL，几秒后，就可以得到类似下图所示的报告：

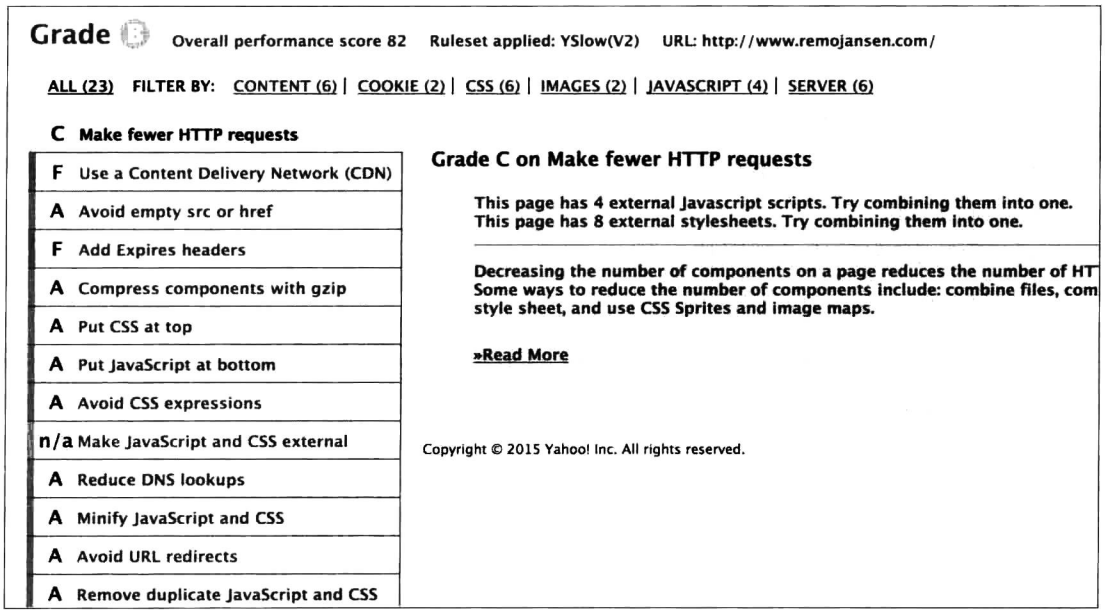


这个报告中包含一些实用建议，可帮助我们提高应用的网络性能和整体用户体验，Google PageSpeed Insights 运用以下规则来提高 Web 应用的速度：

- 避免页面重定向
- 启用压缩
- 优化服务器响应时间
- 善用浏览器缓存
- 压缩静态资源
- 优化图片尺寸
- 优化 CSS 加载
- 优先考虑首屏内容
- 移除阻塞渲染的 JavaScript
- 使用异步脚本

使用这个工具时，单击每条规则的分数，就能看到相关建议与细节，包括每条规则中做得不对和需要改进的地方。

至于 Yahoo YSlow，它有浏览器插件、Node.js 模块和 PhantomJS 插件等版本，通过 `http://yslow.org/`可以找到你想要的版本。使用 YSlow 的话，也会产生一份报告，包括总得分和各项分数，如下图所示。



YSlow 运用以下规则来提高 Web 应用的速度：

- 压缩 HTTP 请求
- 使用 CDN
- 避免空的 `src` 和 `href` 属性
- 增加 expires 或缓存头
- 使用 Gzip 压缩组件
- 把样式表放在顶部
- 把脚本放在底部
- 避免 CSS 表达式（又称动态属性）
- 把 JavaScript 和 CSS 放到外部链接
- 减少 DNS 查询

- 压缩 JavaScript 和 CSS
- 避免重定向
- 移除重复的脚本
- 配置 ETags
- 缓存 AJAX 请求
- AJAX 多用 GET 请求
- 减少 DOM 节点的数目
- 避免 404 错误
- 减小 cookie 的大小
- 为组件使用无 cookie 站点
- 避免过滤
- 不在 HTML 中扩展图片
- 将 `favicon.ico` 压缩并缓存

和之前一样，单击每项规则可以看到相关的建议和细节，包括每条规则中做得不对和需要改进的地方。

如果你希望了解更多关于网络性能优化的知识，可以看一下 Ilya Grigorik 写的 *High Performance Browser Networking* 一书。

GPU 性能分析

Web 应用中部分元素的渲染可被 GPU 加速，GPU 是专门处理图形相关指令的，因此在处理图形时会优于 CPU。比如，CSS3 动画可被 GPU 加速，而 CPU 则加速 JavaScript 动画。过去，实现动画的唯一方法就是通过 JavaScript，但是今天我们应更多使用 CSS3 来代替 JavaScript 动画，因为 CSS3 动画的性能真是好太多了。

最近几年，通过浏览器的 WebGL API 可以直接调用 GPU，通过这个 API 可以使用 GPU 来帮助我们创建 3D 游戏和其他高质量图形应用。

每秒传输帧数

不会过多讨论 3D 应用的性能，因为这个领域广到足以让我们再写一本书。尽管如此，我们仍会提到一个重要的概念：每秒传输帧数 FPS 或帧率，当一个应用在屏幕上显示时，

每秒经历了许多帧，低帧率并不利于整体用户体验。许多相关主题的研究表示，FPS 为 60（每秒 60 帧）可获得最佳用户体验。

当开发一个 Web 应用时，我们应该关注一下帧率并努力保持 FPS 大于 40，特别是对于动画和用户行为。

有一个开源库 stats.js，它可以让我们看到应用的帧率，其 GitHub 地址为 https://github.com/mrdoob/stats.js/。将其下载下来并加载到应用中去，通过新建一个文件或者直接在开发者工具的控制台中运行下面的代码：

```
var stats = new Stats();
stats.setMode(1); // 0: fps, 1: ms

// 帧率计数器的位置（左上角）
stats.domElement.style.position = 'absolute';
stats.domElement.style.left = '0px';
stats.domElement.style.top = '0px';

document.body.appendChild( stats.domElement );

var update = function () {
  stats.begin();
  // 计数器代码如下
  stats.end();
  requestAnimationFrame( update );
};
requestAnimationFrame( update );
```

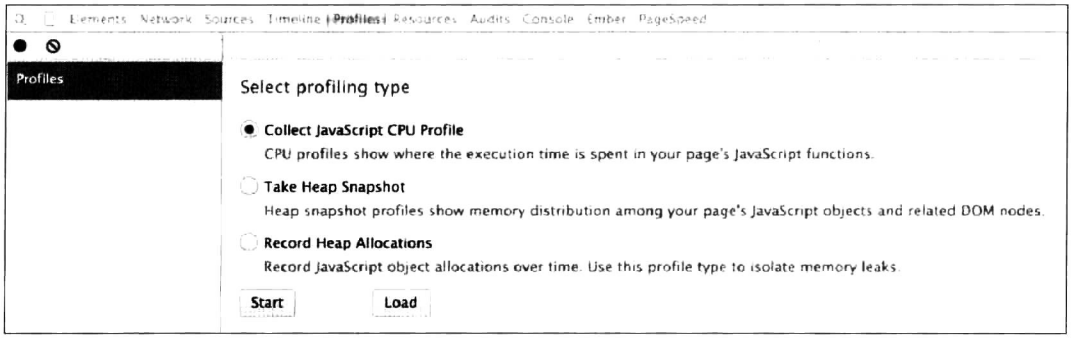
一切正常的话，可以在屏幕左上角看到一个帧率计数器，单击后可从 FPS 模式切换到毫秒模式。

- FPS 模式显示上一秒中渲染过的帧，数字越大越好。
- 毫秒模式显示渲染一帧需要的时间，数字越小越好。

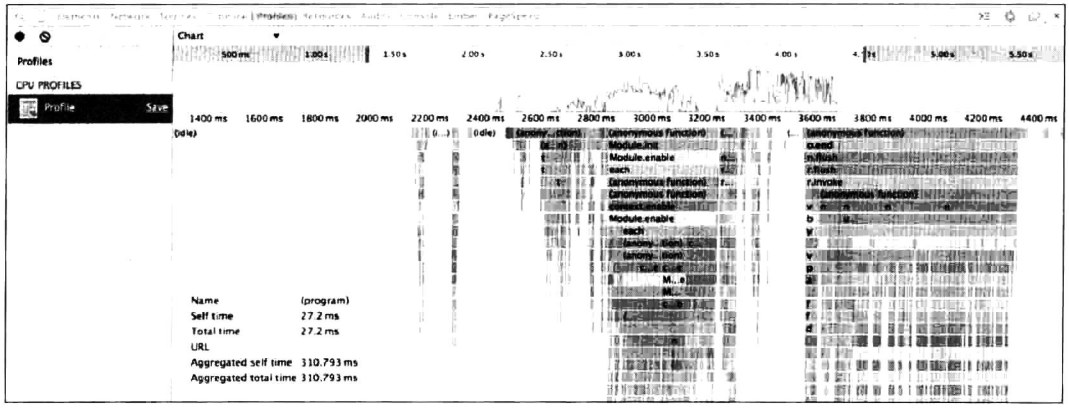
部分高级 WebGL 应用可能需要深度性能分析，这时候可以选择 Chrome 提供的 Trace Event Profiling 工具。如果想了解这个工具，可访问其官方网站 `https://www.chromium.org/developers/how-tos/trace-event-profiling-tool`。

CPU 性能分析

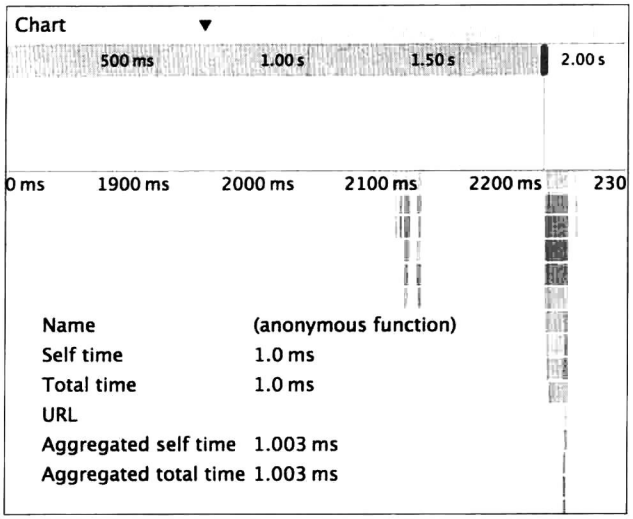
为分析进程的时间损耗情况，首先要看一下应用的运行环境。打开 Chrome 开发者工具的 Profiles 选项卡，可以测试每个函数的运行时间。



在该选项卡中，选中 Collect JavaScript CPU Profile 后单击 Start 按钮，即可开始记录 CPU 的使用情况。当想选择特定的函数进行分析时，只要简单地单击开始/停止按钮即可开始记录。比如我们想分析 `foo` 函数，先单击 Start 按钮开始记录，然后调用 `foo` 函数，再单击一下停止，是不是很简单？然后就能看到类似下方截屏的时间轴：



时间轴的水平方向显示了函数调用的时间顺序，若在某个函数中调用了其他函数，这个函数的调用栈会在垂直方向显示。当把鼠标光标悬浮在其中一个函数上时，可以在时间轴左下角看到其中的细节。



这些细节内容包括如下几项。

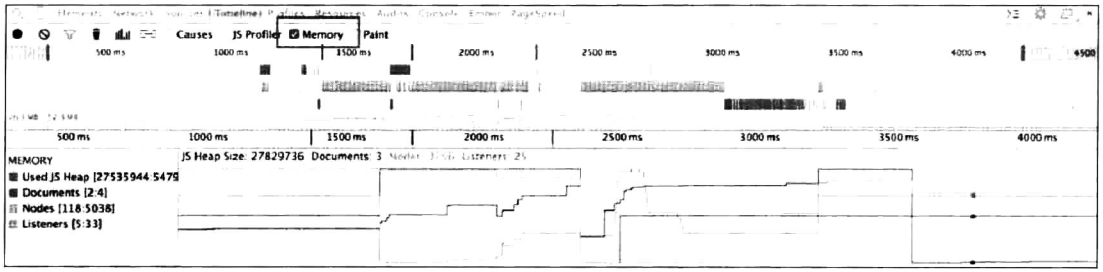
- Name：函数名。
- Self time：当前函数调用完成所需的时间。将该函数中的所有语句执行所需的时间考虑在内，但不包括调用其他函数所需的时间。
- Total time：函数调用完成的总时间。将该函数中的所有语句执行所需的时间考虑在内，包括调用其他函数所需的时间。
- Aggregated self time：单次记录内函数所有调用所需时间，但不包括调用其他函数所需的时间。
- Aggregated total time：单次记录内函数所有调用所需时间，包括调用其他函数所需的时间。

如前面章节中所说的，所有的 JavaScript 代码都在一个单线程中运行，因此，当一个函数执行时，其他函数不会执行。有时，一个函数执行所花费的时间很多，应用就会呈现为不响应状态。可以通过 CPU 性能报告确定具体是哪个函数运行占据过多时间，确定以后就可以去优化、重构以提高应用的表现，通常是尽可能多地使用异步函数和减小函数的体积。

内存性能分析

声明一个变量后，它就会被分配到内存中，当变量离开作用域后，就被垃圾回收器回收内存。在某些特定场景下，变量不会离开作用域，也就不会被销毁、内存不会被回收，这样最终会导致内存泄露（持续性内存不足）。

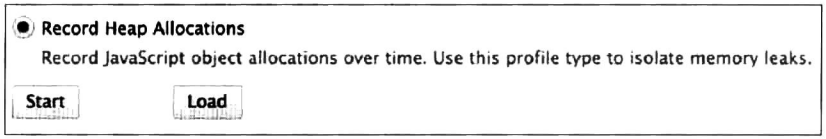
我们可以通过 Chrome 开发者工具来确定内存泄露的原因。首先，需要确定是否存在内存泄露的情况，选择 Timeline 选项卡，单击左上角的图标即可开始记录资源使用情况，单击停止按钮后，会生成时间轴图表，如下所示。



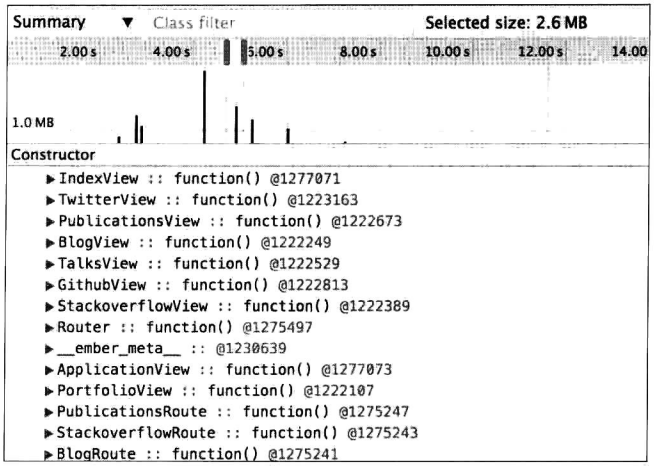
在时间轴上，可以勾选 Memory 选项以查看内存使用情况（Used JS Heap）。在之前的例子中，我们可以看到最后一段有一个明显的下降，这是一个好消息，说明页面加载完成后，被占用的内存大多都被释放了。

内存泄露也可能发生在页面加载完成后，在这种情况下，我们可以先使用一下应用，再去观察内存占用情况。

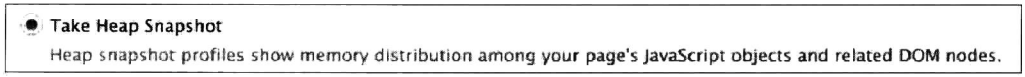
另外一种检查内存泄露的方法是分析内存分配情况，我们可以通过开发者工具中的 Profiles 选项卡记录堆分配情况。



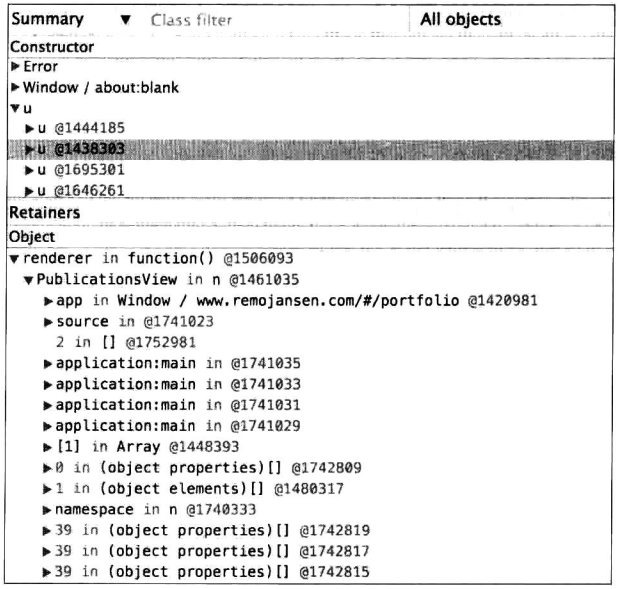
通过记录资源使用情况后可生成报告，只需轻轻单击开始和结束按钮，内存分配报告就会以时间轴的形式进行展示，每条线表示一次内存分配，线的高度表示内存使用多少，可以看出在 8 秒左右时内存已基本全部释放出来：



单击其中一条线时，就能看到存在内存中的所有变量及其值。使用 Profiles 选项卡，还能看到时间轴上任意点的内存快照。



在调试时，想看特定断点下的内存使用时，这个特性十分有用，内存快照工作方式和之前介绍的内存分配比较接近：



上图中，内存快照允许查看快照发生时内存中的所有变量及其值。

垃圾回收器

低抽象级别的编程语言具有低级内存管理机制，另一方面，高抽象级别的编程语言，比如 C# 或 JavaScript，内存会自动分配，靠垃圾回收器回收。

JavaScript 的垃圾回收器十分高效，但这并不意味着我们不需要关心内存。

不管你在使用哪种编程语言，内存的生命周期大多遵循相应模式：

- 按需分配内存
- 使用内存进行读写
- 及时释放内存

垃圾回收器遵循标记-清除原则，周期性扫描标记过的变量，及时回收不再使用的变量所占用的内存。该扫描包括两个阶段：第一阶段为标记阶段，即垃圾回收器标记可回收的变量；第二阶段为清除阶段，即释放上阶段中标记过的变量所占用的内存。

垃圾回收器能够确定哪些东西可以清除，但是我们应保证，当不再需要一个变量时，变量会离开作用域，若一个变量无法离开作用域，一直存在内存里，可能会导致内存泄露。

变量被引用的次数会阻止其内存的释放，因此大多数内存泄露都可以通过解除变量的引用修复，这里列出一些规则可以帮助我们避免潜在的内存泄露：

- 及时清除计时器。
- 及时清除事件监听器。
- 闭包会记住其上下文环境，这意味着需要占据部分内存。
- 使用对象组合时，若存在循环引用，也会有部分变量占据内存导致无法释放的情况。

性能自动化

本部分我们将学习性能优化的自动化，从内容的拼接和压缩，到自动化性能监测和性能测试。

性能优化自动化

在分析应用性能之后，我们开始做一些性能优化工作。主要集中于将应用组件文件进

行拼接和压缩，同时这样做会导致难以调试与维护，而且每次修改组件后都得创建一个新版本。这种重复且无聊的工作，可以使用 Gulp 来帮助进行。Gulp 的诸多插件可以用来专门完成合并压缩、优化图片、生成缓存清单（cache manifest）以及其他自动化性能优化的任务。

如果想了解更多关于 Gulp 的知识，请参考第 2 章。

性能监测自动化

我们已经知道许多性能自动化任务都可以通过 Gulp 来做，这同样能用在性能监测任务上。

为了监测应用的性能，我们可通过收集数据进行性能比较。收集数据的方式主要分为以下三种。

- **真实用户监测（RUM）**：这类方法主要收集真实用户访问的性能数据，该方法通过在浏览器中加载小段 JavaScript 代码实现，主要作用是收集数据、发现性能趋势和模式。
- **模拟浏览器**：这类方法主要收集模拟浏览器的性能数据，该方法最经济省事，但也有限制，因为模拟浏览器不能准确模拟真实用户操作。
- **真实浏览器监测**：这类方法主要收集真实浏览器的性能数据，该方法能更准确呈现真实用户操作，因为收集的数据为真实用户访问网页的真实情况（浏览器、地理位置、网络情况等）。

在第 2 章中，我们学习了如何在 Gulp 中配置 Karma 以便在 PhantomJS 中进行测试。

PhantomJS 是一种可以配置后生成 HTTP *存档*（HAR）文件的模拟浏览器。HAR 文件中记录着 HTTP 跟踪信息，对于我们来说，只需要其中有关浏览器加载内容的记录。

现在网上有不少脚本可以调用 PhantomJS API 来收集数据并格式化，比如，`netsniff.js` 可以导出网络数据的 HAR 文件，该项目的 GitHub 地址为 `https://github.com/ariya/phantomjs/blob/master/examples/netsniff.js`。一旦有了 HAR 文件，就可以使用 HAR 视图以时间轴形式查阅性能信息了，该项目的 GitHub 地址为 `https://github.com/janodvarko/harviewer`。此外，也可以通过自己写脚本或者 Gulp 来读取 HAR 文件。

还可以通过配置 PhantomJS 来运行 YSlow 性能分析报告并将其接入自动化流程，想要了解更多关于 PhantomJS 及性能检测的相关知识，请查阅官方文档 http://phantomjs.org/network-monitoring.html。

如果考虑使用 RUM，可以看一下 New Relic 的解决方法 http://newrelic.com/，或者是 Google Analytics http://www.google.com/analytics/。

性能测试自动化

另外一种提升应用性能的方式是，编写自动化性能测试。这些测试可以用来保证系统满足一组性能目标。下面列出了几种常见的性能测试。

- **加载测试：**这是一种很基础的性能测试，可以通过加载测试来了解预设的加载情况下的系统行为（并发用户数、事务数和时长）。
  - **压力测试：**通常用来测试应用的最大负荷能力，这种测试能看出应用在长时间极端负载下能否正常工作。压力测试在客户端应用中通常作用不大，然而对于开发 Node.js 应用就特别有用，因为 Node.js 应用通常有很多并发用户。
  - **渗透测试：**也叫耐力测试，跟压力测试类似，但仅通过长时间预设的负载量进行测试。用于收集内存使用情况以发现潜在的内存泄露问题，这种测试可帮助我们监测性能是否会下降。
  - **峰值测试：**跟压力测试类似，但仅通过剧增的间隔和预期负载进行测试。这种测试可帮助监测性能能否应对剧烈变化的访问量。
  - **配置测试：**这种测试用于监测配置变化对应用性能和行为的影响，通常用来测试不同的负载平衡问题。

这种测试也能通过类似 JMeter（http://jmeter.apache.org/）和 Locust（http://locust.io/）的工具实现自动化。

错误处理

有效利用可用资源可以帮助我们创建更好的应用，类似的，了解如何处理运行时错误

可以帮助提高应用的整体质量。TypeScript 中的错误处理主要有三种。

Error 类

运行时错误发生后会抛出一个 Error 类：

```
throw new Error();
```

我们可以创建自定义错误，比如传一个参数给 Error 类的构造器：

```
throw new Error('My basic custom error');
```

若需要更多的自定义及控制能力，可以通过继承实现：

```
module CustomException {
  export declare class Error {
    public name: string;
    public message: string;
    public stack: string;
    constructor(message?: string);
  }

  export class Exception extends Error {

    constructor(public message: string) {
      super(message);
      this.name = 'Exception';
      this.message = message;
      this.stack = (<any>new Error()).stack;
    }

    toString() {
      return this.name + ': ' + this.message;
    }
  }
}
```

在上面的代码中，我们声明了一个 Error 类，该类存在于运行时，但并非 TypeScript 自带，所以需要自己声明。然后创建了一个继承自 Error 类的 Exception 类，最后创建了

customError 继承自 Exception 类：

```
class CustomError extends CustomException.Exception {
  // ...
}
```

try…catch 语句和 throw 语句

catch 语句会捕获 try 中抛出的异常。在 try 中写的代码，如果运行失败了，程序就会运行 catch 内的代码。

另外还有一个可选的 finally，会在 try 和 catch 运行后执行：

```
try {
  // 希望运行的代码
  throw new Error("Oops!");
}
catch (e){
  // try中抛出异常时运行的代码
  console.log(e);
}
finally {
  // 无论是否抛出异常都会执行的代码
  console.log("finally!");
}
```

在大多数编程语言中，包括 TypeScript，抛出和接收异常的操作很耗资源，在不必要的情况下应尽量避免使用它们，因为这会影响性能。因此，在会影响性能的函数和循环中尽量不要使用 try…catch 和 throw 语句。

小结

本章我们了解了何为性能、有限资源如何影响性能，还学习了使用一些工具去分析 TypeScript 应用的性能，这些工具允许我们找出可能发生的错误，比如低帧率、内存泄露、加载时间过长等问题。我们还知道了可以通过自动化任务去处理类似性能检测和性能测试等工作。

在下一章，我们将学习如何开展自动化测试，以提高 TypeScript 应用的可维护性和可靠性。

7
应用测试

本章，我们会了解如何为 TypeScript 应用编写单元测试，也会看到如何使用工具和框架使我们的测试更加便利。

本章内容主要包含以下几个要点：

- 建立基础测试结构
- 测试规划和方法
- 如何使用 Mocha、Chai 和 Sinon.JS
- 如何使用断言、用例和套件
- 测试监视
- 测试桩
- 多环境测试
- 如何使用 Karma 和 PhantomJS
- 端对端测试
- 生成测试覆盖率报告

我们将会从安装一些必要的第三方依赖开始。

软件测试术语表

在本章中，我们会使用一些没有软件测试经验的人可能不熟悉的概念。让我们在开始之前快速看一下这些流行的关于测试的概念。

断言

断言是一个条件，它是必须被测试确认的一段代码的行为是否与期望相符，或换句话说是否与要求一致。

想象一下，我们在 Google Chrome 团队中工作，要实现 JavaScript Math 对象，需求差不多应该是这样的:

Math.pow(base, exponent)函数应该返回底数（base）的指数（exponent）次方（指数用来提升底数的幂——base ^ exponent）。

有了这些信息，我们可以创建以下的实现：

```
class Math1 {
  public static pow(base: number, exponent: number) {
    var result = base;
    for(var i = 1; i < exponent; i++){
      result = result * base;
    }
    return result;
  }
}
```

为了确保这个方法被正确实现，我们必须测试它是否遵从需求。如果我们进一步分析需求，至少可以得到两个必要的断言。

这个函数必须返回底数的指数幂：

```
var actual = Math1.pow(3,5);
var expected = 243;
var asertion1 = (Math1.pow(base1, exponent1) === expected1);
```

底数没有被用作指数（或指数没有被用作底数）:

```
var actual = Math1.pow(5,3);
var expected = 125;
var asertion2 = (Math1.pow(base2, exponent2) === expected2);
```

如果两个断言都合法，那代码就是满足需求的，并且我们知道它会如预期那样工作：

```
var isValidCode = (asertion1 && asertion2);
console.log(isValidCode);
```

测试规范

测试规范（specs）是软件开发人员用来指代测试规范的一个术语。一个测试规范（不要和测试计划混淆）是一个详细的清单，它包含了需要测试的场景，这些场景将如何被测试，它们应该怎样被测试等。稍后将在本章看到如何使用测试框架定义一个测试规范。

测试用例

*测试用例*是决定一个程序中的功能是否按照原始期望工作的一些条件。我们可能想知道断言和测试用例有什么区别。断言是一个条件，而测试用例是一组条件。接下来会看到如何使用测试框架定义测试用例。

测试套件

*一个测试套件*是许多测试用例的集合。测试用例只是针对某一个具体的测试场景，一个测试套件可以包含多个测试用例来覆盖很多的测试场景。

测试监视

测试监视（Spy）是某些测试框架提供的功能。它允许我们包裹一个函数，并且记录它的使用情况（输入、输出和被调用的次数）。当用 spy 包裹一个函数时，这个函数的功能并没有被改变。

替身

*替身*对象指测试执行时被传入但并没有实际用到的对象。

测试桩

桩（stub）是测试框架提供的一种功能。测试桩也允许包裹一个方法然后观察它的使用情况。和测试监视不一样的是，当使用测试桩包裹一个函数时，这个函数的功能会被新的行为替换。

模拟

模拟（mock）很容易和测试桩（stub）混淆。Martin Fowler 有一次写了一篇题为《模

拟不是测试桩》（Mocks Aren't Stubs）的文章：

> 特别是我发现mock经常会和 stub 这个概念混淆——一个通常被用于测试环境的帮助类。我现在才理解为什么会被混淆，因为我也曾经有一段时间把它们当成类似的东西。在与 mock 开发者的交谈后，我茅塞顿开。
>
> 它们之间有两个不同点，其中一个是测试结果的验证方式不同：一个是状态验证，另一个是行为验证。
>
> 另一方面是测试与设计哲学完全不同：我将它们分别称为，一个是传统型的，另一个是测试驱动开发的 mockist（mock 风格）的。

测试模拟和测试桩都能为测试用例提供一些输出。但是尽管它们在这方面非常相似，但信息流差异非常大：

- 测试桩为被测试的程序提供输入值，可让被测试程序能扮演另外的程序。
- 模拟为测试提供输入来决定测试是否能通过。

当阅读到本书的末尾时，你会对模拟和测试桩的区别有更深的认识。

测试覆盖率

测试覆盖率是指程序中有多大比例的代码通过自动化测试被测试到。在接近本章结尾的部分，我们会看到如何使用 Istanbul（`http://gotwarlost.github.io/istanbul/`）来创建这样的报告。

必要的准备

贯穿本章，我们会用到一些第三方工具，包括一些框架和自动化工具。我们会从查看这些工具的细节开始。在开始之前，我们需要用 npm 在实现本章示例的文件夹中创建 `package.json` 文件。

新建一个名为app的文件夹并在里面运行`npm init`命令，生成一个新的`package.json`文件：

```
npm init
```

查阅第 2 章可获取更多关于 npm 的帮助。

Gulp

我们将使用 Gulp 任务运行器运行一些必要的任务来执行测试。可以通过 npm 安装 Gulp：

```
npm install gulp -g
```

如果你对任务运行器和持续集成构建服务不熟悉，可以先看一下第 2 章。

Karma

Karma 是一个测试运行器，我们将使用 Karma 自动执行测试。这将非常有用，因为有时候团队成员可能没有执行测试。相应的，它会被持续集成构建服务触发（通常通过一个任务运行器）。

Karma 可以通过安装插件与多种测试框架一起使用。让我们使用下面的命令进行安装：

```
npm install --save-dev karma
```

我们也将安装一个 Karma 插件来让创建测试覆盖率报告更加容易：

```
npm install --save-dev karma-coverage
```

Istanbul

Istanbul 是指出哪一行代码在自动化测试中被测试到的工具，它可以生成测试覆盖率报告。这些报告可以帮助我们了解项目被测试的程度，因为它会显示没有被测试到的代码的行数和测试到的代码在程序中所占的比例。一般建议，整个程序至少达到 75%的测试覆盖率，而很多开源项目的目标都是达到百分之百的测试覆盖率。

Mocha

Mocha 是一个流行的 JavaScript 测试框架库，使用它能方便地创建测试套件、测试用例和测试规范。Mocha 可以用来在前端或者后端测试 TypeScript，指出性能问题，生成不

同类型的测试报告，还有很多其他功能。

可通过以下命令安装 Mocha 和 Karma-Mocha 插件：

```
npm install --save-dev mocha karma-mocha
```

Chai

Chai 是一个支持测试驱动开发（TDD）和行为驱动开发（BDD）测试风格的断言库。

我们将在本章的后面了解更多关于 BDD 和 TDD 的内容。

Chai 的主要目标是减少创建一个测试断言需要的工作量，并且使测试更可读。

可以通过以下命令安装 Chai 和 Karma-Chai 插件：

```
npm install --save-dev chai karma-chai
```

Sinon.JS

Sinon.JS 是一个独立的框架，它提供了一组 API（测试监视、测试桩和测试模拟），可帮助我们独立地测试一个组件。独立地测试软件模块非常困难，因为通常模块之间有着很高程度的耦合，但通过 Sinon.JS 这种模拟框架就可以做到。

可以通过以下命令安装 Sinon.JS 和 Karma-Sinon 插件：

```
npm install --save-dev sinon karma-sinon
```

类型定义

为了能与第三方 JavaScript 框架共同使用并且得到良好的支持，需要引入每一个库的类型定义。可使用 `tsd` 包管理器来安装必要的类型定义：

```
tsd install mocha --save
tsd install chai --save
tsd install sinon --save
tsd install jquery - -save
```


在第 2 章可以找到更多关于 tsd 的帮助。

PhantomJS

PhantomJS 是一个无显示界面的浏览器。可以通过 PhantomJS 运行浏览器测试而不需要真正地开启浏览器。因为几个原因使这一功能非常有用，最主要的原因是，PhantomJS 可以通过命令行执行，这让它非常容易与任务运行器和持续集成服务集成。第二个原因是不必打开浏览器潜在地减少了完成测试套件所需要的时间。

需要安装将运行在 PhantomJS 中的 Karma 插件：

```
npm install --save-dev phantomjs
npm install --save-dev karma-phantomjs-launcher
```

Selenium 和 Nightwatch.js

Selenium 是一个测试运行器，但是它被特别设计只运行叫作端对端（E2E）测试的特定的测试。


稍后将在本章了解更多关于 E2E 测试的内容，目前我们不需要太过关注这个主题。

尽管我们只会在本章的结尾处看到如何使用 Selenium，现在也可以安装它。我们不会直接使用 Selenium，因为将会使用另一个工具（Nightwatch.js）来做 E2E 测试，它将自动运行 Selenium。

Nightwatch.js 是一个自动化测试框架，使用 Node.js 开发 Web 程序和网站，它使用了 Selenium 网络驱动 API。它是一个完整的浏览器（E2E）测试解决方案。

可以通过执行下面的命令安装 Selenium 和 Nightwatch.js：

```
npm install --save-dev gulp-nightwatch
npm install selenium-standalone -g
selenium-standalone install
```

Selenium standalone 需要 Java 的二进制文件被安装到开发环境中，并且可以通过$PATH 环境变量访问到。访问 https://www.java.com/en/download/help/index_installing.xml 可获取更多关于 Java 安装的信息。

测试计划和方法

在软件开发的过程中，我们通常有许多选择的机会。每次在设计一个全新的应用时，都可以选择要使用的数据库的类型、架构和框架。不是所有的选择都与技术相关。比如，可以选择一个软件开发方法，类似极限编程或者敏捷开发。在测试方面，有两种主要的风格或者方法可选：测试驱动开发（TDD）和行为驱动开发（BDD）。

测试驱动开发

测试驱动开发（TDD）是一种测试技术，它鼓励开发者在写程序代码之前写测试。通常情况下，使用 TDD 编码包含以下基本步骤：

1. 编写一个不通过的测试。
2. 运行这个测试，并保证它不能通过（这时候没有任何程序代码，所以测试应该无法通过）。
3. 编写应用代码，让测试通过。
4. 运行这个测试，保证它能通过。
5. 运行所有其他测试，保证程序的其他部分没有被破坏。
6. 重复以上步骤。

是否使用 TDD 是思维模式的差异。很多开发者不喜欢写测试，这带来的挑战是，如果把测试留在开发的最后一个环节，那么程序的测试最终就不会被实现，或者只有程序的一部分被测试。

TDD 是被推荐的，因为它会极大程度地帮助你和你的团队增加程序的测试覆盖率，并且因此显著减少潜在的错误数量。

行为驱动测试

行为驱动开发（BDD）在测试驱动开发后出现，它的使命是提取 TDD 的精华。以 BDD 方式测试的重点是描述并且阐明测试应该关注程序的需求而非测试的需求。理想状态下，它会鼓励开发者少思考测试这件事情本身而更多地思考整个程序。

BDD 原则第一次被提出的文章由 Dan North 所著，在线地址是 http://dannorth.net/introducing-bdd/。

如同我们已经看到的，Mocha 和 Chai 提供了一些 API 给 BDD 和 TDD 方法。在本章后面的部分，我们将进一步研究这两种方法。

推荐使用哪一种技术并不重要，因为 TDD 和 BDD 都是非常好的测试技术。BDD 是在 TDD 之后出现的，且其主要目标是改善 TDD 技术，所以可以认为 BDD 比 TDD 具有一些额外的优势。在 BDD 里，测试的描述关注于程序在做什么，而不是测试应该做什么，这能让开发者确认测试中反映的客户行为。BDD 还被用于记录能被客户和开发者理解与验证的需求。而另一方面，TDD 测试不能很轻松地被客户理解。

测试计划和测试类型

测试计划有时候会被错误地用来指代测试规范。而测试规范是用来定义有哪些场景会被测试，怎样被测试，测试计划是特定被测试区域内的测试规范的集合。

一般推荐为具体的测试计划撰写文档，因为一个测试计划包括非常多的步骤、文档和实践。测试计划的一个重要目标是，定义、指出什么类型的测试对于程序中的特定组件是合格的。

下面是最常用的几种测试类型。

- **单元测试：**这被用来测试独立的组件。如果组件不是独立的，或换句话说它包含一些依赖，我们就需要通过设置测试模拟和依赖注入来尽可能地让它在测试中独立。如果没办法操作组件的依赖，则使用测试监视以让测试更加方便。

  我们的主要目标是在测试的时候达到组件完全独立的程度。一个单元测试也应该需要快速运行，应该避免输入输出、网络请求，或任何可能影响到测试运行速度的操作。
- **部分集成测试和整体集成测试：**这被用来测试一组组件（部分集成测试），或者整个程序（整体集成测试）。在集成测试中，我们会正常地使用已知的数据与后端通

信，获取到将会显示在前端的数据。随后我们将断言显示的信息是否是正确的。

- **回归测试**：用来确认程序错误是否被修复。如果使用 TDD 或 BDD，当遇到一个程序错误时，应该新建一个单元测试来重现这个问题，然后改写代码。做完这些就可以运行单元测试尝试重现错误并通过测试，最后确认所有测试都能正常通过。
- **性能/加载测试**：用来确认程序是否达到性能预期。可以使用性能测试确认程序是否能处理大量的用户并发或活跃度剧增。想要了解更多此类测试，可以查阅第 6 章。
- **端对端（E2E）测试**：这种测试和整体集成测试其实并没有什么不同。最主要的区别是在一个 E2E 测试活动期间，我们会尝试完全模拟与正式用户一样的环境。为了达到这个目的，将使用 Nightwatch.js 和 Selenium。
- **验收测试（UAT）**：这被用来验证系统是否符合用户的所有需求。

建立测试基础结构

我们在本章前面部分看到，对于单元测试，通常需要让程序中的各个组件保持独立。

为了让程序中的组件独立，需要遵循一些原则（例如依赖反转原则）来帮助我们增加组件之间的分离程度。

现在将要使用 Gulp 和 Karma 设置一个测试环境并使用 Mocha 和 Chai 编写一些自动化测试。到本章结尾的时候，我们将学会如何编写单元测试以帮助提高应用中组件的独立性和互相之间的分离程度，以及单元测试如何引导我们开发出可维护、可扩展的程序。

让我们从创建一个新的程序文件夹开始，在本章开始时介绍过的 app 文件夹中新建两个文件夹。

将第一个文件夹命名为 `source`，第二个文件夹命名为 `test`。这里可以看看在本章结束前我们的文件夹是什么结构：

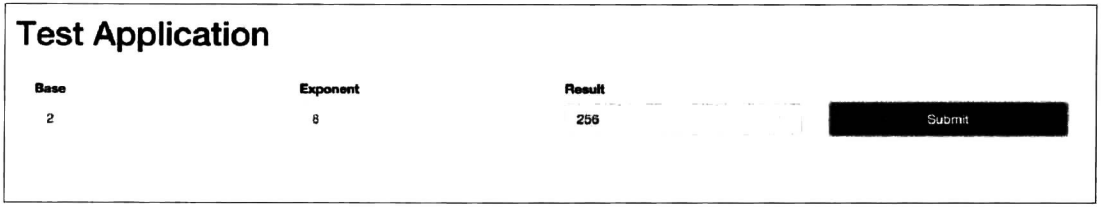
```
├──app
    ├── gulpfile.js
    ├── index.html
    ├── karma.conf.js
    ├── nightwatch.json
    ├── package.json
    ├── source
    |    ├── calculator_widget.ts
```

```
|   ├── demos.ts
|   ├── interfaces.d.ts
|   ├── math_demo.ts
├── style
|   └── demo.css
├── test
|   ├── bdd.test.ts
|   ├── e2e.test.ts
|   ├── tdd.test.ts
├── tsd.json
└── typings
```

我们将开发一个可以进行单元测试的小程序，将会写一个单元测试和一个端对端测试。

这个示例包含在随书的代码中。

一旦完成了我们的程序，就可以打开浏览器看到如下截图的表单。这个表单可以让我们得到一个数字（底数）的另一个数字（指数）次幂。



使用 Gulp 构建这个程序

我们从新建一个 gulpfile.js 文件开始，和我们在第 2 章中一样。首先将需要用的 node 模块引入进来：

```
var gulp        = require("gulp"),
    browserify  = require("browserify"),
    source      = require("vinyl-source-stream"),
    buffer      = require("vinyl-buffer"),
    run         = require("gulp-run"),
```

```
    nightwatch  = require('gulp-nightwatch'),
    tslint      = require("gulp-tslint"),
    tsc         = require("gulp-typescript"),
    browserSync = require('browser-sync'),
    karma       = require("karma").server,
    uglify      = require("gulp-uglify"),
    docco       = require("gulp-docco"),
    runSequence = require("run-sequence"),
    header      = require("gulp-header"),
    pkg         = require(__dirname + "/package.json");
```

记住，我们需要通过 npm 包管理器安装所有必需的依赖包。可以在 package.json 文件中查看所有的依赖和它们各自的版本。

第二件事是创建一些任务来编译 TypeScript 代码。这里需要注意，我们将要编译的程序代码放到/build/source 文件夹中，编译测试代码放到/build/test 文件夹中。

```
var tsProject = tsc.createProject({
  removeComments : false,
  noImplicitAny : false,
  target : "ES5",
  module : "commonjs",
  declarationFiles : false
});
gulp.task("build-source", function() {
  return gulp.src(__dirname + "/source/*.ts")
    .pipe(tsc(tsProject))
    .pipe(gulp.dest(__dirname + "/build/source/"));
});
```

上面的这个 Gulp 任务在 source 文件夹下将 TypeScript 文件编译为 JavaScript 文件，并将它们保存在 build/source 文件夹中。可以使用下面的命令运行这个任务：

```
gulp build-source
```

上面这行命令在没有源文件的情况下可能会失败。可以从随书源码中复制出项目源文件，或者继续本章的阅读跟着我们创建这些文件。

我们也会声明第二个任务来编译单元测试，它的输出结果将存储在 `build/test` 文件夹中：

```
var tsTestProject = tsc.createProject({
  removeComments : false,
  noImplicitAny : false,
  target : "ES5",
  module : "commonjs",
  declarationFiles : false
});
gulp.task("build-test", function() {
  return gulp.src(__dirname + "/test/*.test.ts")
    .pipe(tsc(tsTestProject))
    .pipe(gulp.dest(__dirname + "/build/test/"));
  });
```

我们可以通过以下命令运行这个新的任务：

```
gulp build-test
```

一旦 JavaScript 到了 `build` 目录下，我们就需要打包出外部依赖（在前面的编译设置中选择了`{module: "commonjs"}`）到可以在浏览器中运行的已打包库。

对每一个库，Browserify 需要一个唯一的入口，因此，我们将创建三个任务——与每一个打包的库一一对应。

创建一个任务打包程序自身：

```
gulp.task("bundle-source", function () {
  var b = browserify({
    standalone : 'demos',
    entries: __dirname + "/build/source/demos.js",
    debug: true
  });
```

```
  return b.bundle()
    .pipe(source("demos.js"))
    .pipe(buffer())
    .pipe(gulp.dest(__dirname + "/bundled/source/"));
});
```

如同我们在上一个 Gulp 任务中一样，可以通过以下命令行执行这个新任务：

```
gulp bundle-source
```

还需要新建一个另外的任务，将程序中所有的单元测试打包到一个单一的测试套件文件：

```
gulp.task("bundle-test", function () {

  var b = browserify({
    standalone : 'test',
    entries: __dirname + "/build/test/bdd.test.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("bdd.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest(__dirname + "/bundled/test/"));
});
```

打包的代码里包含了在 tdd.test.ts 和 bdd.test.ts 中 TDD 和 BDD 风格的代码。然而，在本章的例子中，我们将只关注 BDD 风格的代码。

可以通过以下命令执行新的任务：

```
gulp bundle-test
```

最后，我们将新建另一个任务，将程序中所有的 E2E 测试代码打包成一个 E2E 测试套件文件：

```
gulp.task("bundle-e2e-test", function () {
```

```
  var b = browserify({
    standalone : 'test',
    entries: __dirname + "/build/test/e2e.test.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("e2e.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest(__dirname + "/bundled/e2e-test/"));
});
```

可以通过以下命令执行这个新任务：

```
gulp bundle-e2e-test
```

使用 Karma 运行单元测试

我们已经在第 2 章中学习了 Karma，下面新建一个任务执行 Karma：

```
gulp.task("run-unit-test", function(cb) {
  karma.start({
    configFile : __dirname + "/karma.conf.js",
    singleRun: true
  }, cb);
});
```

这里的 Karma 任务配置非常简单，因为主要的配置包括合并代码都位于 karma.conf.js 文件中。来看一下配置文件：

```
module.exports = function(config) {
  'use strict';

  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
```

```
    coverageReporter: {
      type: 'lcov',
      dir: __dirname + '/coverage/'
    },
    plugins: [
       'karma-coverage',
       'karma-mocha',
       'karma-chai',
       'karma-sinon',
       'karma-phantomjs-launcher'
    ],
    preprocessors: {
      '**/bundled/test/bdd.test.js': 'coverage'
    },
    files: [{
      pattern: "/bundled/test/bdd.test.js",
      included: true
    }, {
      pattern: "/node_modules/jquery/dist/jquery.min.js",
      included: true
    }, {
      pattern: "/node_modules/bootstrap/dist/js/bootstrap.min.js",
      included: true
    }],

    client: {
      mocha: {
        ui: "bdd"
      }
    },
    port: 9876,
    colors: true,
    autoWatch: false,
    logLevel: config.DEBUG
  });
};
```

看一下这个配置文件，我们已经配置了测试文件所在的路径和将使用哪一个浏览器运行测试（PhantomJS）。仅声明要使用的是哪一个浏览器是不够的，还要安装一个插件让Karma 启动这个浏览器。

在使用 Mocha、Chai 和 Sinon.JS 编写测试前，我们已经加载了这些框架对应的插件并集成到 Karma 中。还有很多其他流行的测试框架，它们中的大部分都能通过使用插件与 Karma 适配。

在上面的配置文件中，另一项有趣的设置是客户端入口文件。我们将它配置成 Mocha 并指出要使用 BDD 风格。

当 Karma 执行 Mocha 单元测试的时候，它自动在内部生成一个 HTML 页面并且加载所有 files 字段和 plugins 字段指定的文件。在上面的例子中，Karma 会生成一个 HTML 页面引用（使用<script>标签）Mocha、Chai、Sinon.JS（plugins 字段定义）、jQuery、Bootstrap 和 bdd.test.js 文件（files 字段定义）。

随书的源码中包含了 package.json 文件。可以使用这个文件运行 npm install 命令并下载所有的第三方依赖（包括 jQuery 和 Bootstrap）。

需要了解的很重要的一点是，只有通过 files 字段加载进来的文件在测试执行期间才是可用的，所有这些文件都会通过<script> 标签加载。有时，会遇到文件无法找到或者解析失败的错误（当一个非 JavaScript 文件通过<script>标签加载）。我们可以通过设置 pattern、included、served 和 watched 获得更好的文件引用流程控制。

设置	描述
pattern	匹配文件的方式
included	如果 autoWatch 选项是 true，所有 watched 选项被设置为 true 的文件将会被监视是否发生变更
served	文件是否被 Karma 的 webserver 服务
watched	文件是否被浏览器用<script>标签引用。如果手动加载这些文件则设置为 false（比如使用 RequireJS）

karma.conf.js 文件中也包含了一些生成测试覆盖率报告的设置，但我们现在先跳过这些，在本章的结尾处再关注它们。

记住，可以在 http://karma-runner.github.io/0.8/config/configuration-file.html 找到 karma.conf.js 中设置的所有字段的所有细节。

使用 Selenium 和 Nightwatch.js 运行 E2E 测试

Karma（结合了 Mocha、Chai 和 Sinon.JS）是编写和运行单元测试以及一部分集成测试的非常棒的工具。然而，Karma 并非编写 E2E 测试最好的工具，我们将使用另一个工具来写接下来的一些 E2E 测试：Selenium 和 Nightwatch.js。

要设置 Nightwatch.js，我们会从创建一个新的掌控 E2E 测试执行流程的 Gulp 任务开始。只需要指定一个额外的叫作 Nightwatch.json 的配置文件：

```
gulp.task('run-e2e-test', function(){
 return gulp.src('')
   .pipe(nightwatch({
    configFile: __dirname + '/nightwatch.json'
   }));
});
```

我们将要关注 Nightwatch.js，是因为它被设计为与主流的框架配合工作。但如果你使用的是 AngularJS，推荐你看一下 Protractor。Protractor 是一个非常棒的 E2E 测试框架，并且与 AngularJS 有极高的集成度。

nightwatch.json 文件包含了整个 E2E 测试所需的配置。我们需要指定 E2E 测试套件的位置和其他基础的 Selenium 配置。

我们可以这样看 Selenium，它或多或少与 Karma 类似，它是一个在浏览器中执行单元测试的工具。Selenium 与它最主要的区别在于，Selenium 允许以更好地模拟用户行为的方式来编写测试。要明白 Nightwatch.js 并不是一个直接控制测试运行的工具，Nightwatch.js 是用来编写 E2E 测试的框架，并与 Selenium 结合执行测试。

在这个例子里，我们将使用 nightwatch.json 中的 start_process 入口告诉 Nightwatch.js 不要为我们运行 Selenium。nightwatch.json 文件如下所示：

```
{
  "src_folders" : ["bundled/e2e-test/"],
  "output_folder" : "reports",
  "selenium" : {
    "start_process" : false
  },
  "test_settings" : {
    "default" : {
        "silent": true,
        "screenshots" : {
          "enabled" : true,
          "path" : "screenshots"
      },
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled" : true,
        "acceptSslCerts" : true
      }
    },
    "phantomjs" : {
      "desiredCapabilities": {
          "browserName": "phantomjs",
          "javascriptEnabled" : true,
          "acceptSslCerts" : true,
          "phantomjs.binary.path" : "./node_modules/phantomjs/bin/phantomjs"
      }
    },
    "chrome" : {
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    }
  }
}
```

我们将使用 selenium-standalone npm 包手动运行 Selenium（可以在准备阶段部分找到安装细节）。

```
selenium-standalone start
```

除了设置 Selenium，还需要设置在网络服务器上执行 E2E 测试的时候要使用哪个浏览器。

如果想获取更多关于 Nightwatch.js 配置的参数信息，可以访问官方文档：http://nightwatchjs.org/guide#settings-file。

最后，可以运行 E2E 测试了，需要在网络服务器上运行我们的程序。如在第 2 章中看到的，可以使用 browserSync 来达到这个目的。所以我们将增加一个任务部署 browserSync：

```
gulp.task('serve', function(cb) {
  browserSync({
    port: 8080,
    server: {
      baseDir: "./"
    }
  });
  gulp.watch([
    "./**/*.js",
    "./**/*.css",
    "./index.html"
  ], browserSync.reload, cb);
});
```

如果其中一个测试没有通过，而且不知道是什么原因导致的，可以在浏览器中运行程序来手动测试它。

这些任务正确的执行顺序非常重要。我们需要开启一个控制台或终端启动 Selenium：

```
selenium-standalone start
```

开启另一个控制台或终端运行下面的命令：

```
gulp build-source
```

```
gulp build-test
gulp bundle-source
gulp bundle-e2e-test
gulp serve
```

最后，开启第三个控制台运行下面的命令：

```
gulp run-e2e-test
```

使用 Mocha 和 Chai 创建测试断言、规范和套件

现在，测试的基础已经搭建完毕，可以开始写单元测试了。需要记住，我们将遵循 BDD 的测试风格，意味着要在写实际的代码之前就开始写测试。

我们要写一个网页版计算器，为保持简洁，将只实现它的一个功能。在做过一些分析之后，拿出了一个可以帮助理解需求的设计接口。在 interfaces.d.ts 文件中声明下面的接口：

```
interface MathInterface {
  PI : number;
  pow( base: number, exponent: number);
}
```

可以看到，这个计算器允许我们计算一个数的指数幂和获取数字 PI。现在我们知道了需求，可以开始写一些单元测试了。新建一个名为 bdd.test.ts 的文件，然后写入下面的代码:

```
///<reference path="../typings/tsd.d.ts" />
///<reference path="../source/interfaces.d.ts" />

import { MathDemo } from "../source/math_demo";

  var expect = chai.expect;

  describe('BDD test example for MathDemo class \n', () => {

    before(function(){ /* invoked once before ALL tests */ });
    after(function(){ /* invoked once after ALL tests */ });
```

```
    beforeEach(function(){ /* invoked once before EACH test */ });
    afterEach(function(){ /* invoked once before EACH test */ });

    it('should return the correct numeric value for PI \n', () => {
      var math : MathInterface = new MathDemo();
      expect(math.PI).to.equals(3.14159265359);
      expect(math.PI).to.be.a('number');
    });

    // ...
  });
```

在上面的代码片段中，我们引入了必要的描述文件和一个名为 MathDemo 的外部模块。这个外部模块会声明 MathDemo 类，它实现了我们将要测试的 MathInterface。

我们还看到有一个名为 expect 的快捷访问变量，这样就不需要在每次访问 expect 的时候都写 chai.expect 了。

```
var expect = chai.expect;
```

在快捷变量后面，可以看到第一个测试套件：

```
describe('BDD test example for MathDemo class \n', () => {
```

用 describe() 函数声明了这个套件，它包裹了一组单元测试，单元测试是被 it() 函数声明的：

```
it('should return the correct numeric value for PI \n', () => {
```

在单元测试内部，可以执行一个或多个断言。Chai 断言使用链式调用，能提供简单易读的代码：

```
expect(math.PI).to.equals(3.14159265359);
expect(math.PI).to.be.a('number');
```

这里要注意到一个场景，我们会重复地在一个测试套件的单元测试中写一些初始化测试的代码。有一些帮助函数可以避免代码重复。

before() 函数会在测试套件的所有测试前执行一次。after() 函数会在测试套件的所有测试执行后执行一次：

```
before(function(){ /* invoked once before ALL tests */ });
```

```
after(function(){ /* invoked once after ALL tests */ });
```

beforeEach()函数会在测试套件的每一个测试运行前执行一次，afterEach()函数会在测试套件的每一个测试运行完之后执行一次：

```
beforeEach(function(){ /* invoked once before EACH test */ });
afterEach(function(){ /* invoked once before EACH test */ });
```

如果现在运行这个测试，它将不能通过，因为要测试的功能（PI）还没有实现。我们新建一个名为math_demo.ts的文件，然后在里面增加下面的代码：

```
///<reference path="./interfaces.d.ts" />

class MathDemo implements MathInterface{
  public PI : number;
  constructor() {
    this.PI = 3.14159265359;
  }

  //...
}
export { MathDemo };
```

如果用 Karma 运行这个测试，将会无误地通过。以正确的顺序执行 Gulp 任务非常重要。我们需要打开一个控制台或者终端执行下面的命令：

```
gulp build-source
gulp build-test
gulp bundle-source
gulp bundle-test
```

最后，通过下面的命令执行单元测试：

```
gulp run-unit-test
```

在MathInterface中还有一个需求，所以需要再次重复 BDD 的整个流程，但这次需要测试一个名为pow的函数，而不是一个属性。我们将会在之前写的测试套件中加入一个新的测试：

```
it('should return the correct numeric value for pow \n', () => {
```

```
    var math : MathInterface = new MathDemo();
    var result = math.pow(3,5);
    var expected = 243;
    expect(result).to.be.a('number');
    expect(result).to.equal(expected);
});
```

如之前在 MathInterface 接口中看到的，即将测试的这个 pow 函数，接受两个数字类型的参数。因此我们创建了一个测试，它将创建新的 MathDemo 实例并且调用 pow 方法，传入 3 和 5 两个参数。我们期望的结果为 243，因此断言 pow() 函数返回一个数字类型的结果，为 243。

在这个时候，之前的测试将会无法通过，因为 pow 方法还没有被实现。让我们回到 math_demo.ts 文件并实现 pow 方法：

```
///<reference path="./interfaces.d.ts" />

class MathDemo implements MathInterface{
  public PI : number;

  constructor() {
    this.PI = 3.14159265359;
  }

  public pow(base: number, exponent: number) {
    var result = base;
    for(var i = 1; i < exponent; i++) {
      result = result * base;
    }
    return result;
  }

  // ...
}
  export { MathDemo };
```

如果再次运行测试，可以看到有多少个测试在执行，它们中有多少未通过，所有的测试运行完花了多长时间：

```
Executed 2 of 2 SUCCESS (0.007 secs / 0.008 secs)
```

测试异步代码

在第 3 章中，我们知道了如何编写异步代码；在第 6 章中，我们知道了使用异步代码是保证程序性能的黄金法则。应该尽可能地以编写异步代码为目标，因此学习如何测试异步代码就非常重要了。

让我们实现一个异步版本的 pow 函数来说明如何测试异步函数。从依赖开始：

```
interface MathInterface {
  // ..
  powAsync(base: number, exponent: number, cb : (result : number) => void);
}
```

我们需要实现一个名为 powAsync 的函数，它接受两个数字类型的参数（和以前一样）和一个回调函数。异步版本的测试和同步版本的测试几乎是一样的：

```
it('should return the correct numeric value for pow (async) \n',
(done) => {
  var math : MathInterface = new MathDemo();
  math.powAsync(3, 5, function(result) {
    var expected = 243;
    expect(result).to.be.a('number'); expect(result).to.equal(expected);
    done(); // invoke done() inside your call back or fulfilled promises
  });
});
```

需要注意是，这一次 it 方法传入了一个名为 done 的参数。这个参数是一个函数，需要执行它来表明测试已经完成。

默认情况下，it 方法会等待它的回调返回结果，但在测试异步代码时，函数可能会在测试执行完成之前返回。

```
public powAsyncSlow(base: number, exponent: number, cb : (result :
number) => void) {
  var delay = 45; //ms
  setTimeout(() => {
    var result = this.pow(base, exponent);
    cb(result);
```

```
  }, delay);
}
```

在测试异步代码时，当超过 2000 毫秒才调用 done 函数时，Mocha 会认为测试未通过（超时）。超时失败前的时间是可配置的，它是作为对那些运行缓慢的函数的警告。

Mocha 推荐这么做，当一个函数运行超过 40 毫秒时，我们就需要研究如何提升它的性能了。如果一个函数执行超过 100 毫秒，就必须研究如何提升它的性能。默认情况下执行时间超过 2000 毫秒时将不可忍受。

断言异常

如在上一个例子中看到的，断言一个变量的值或者类型是非常直接的。但是有一个场景可能不如上面的那么直接，这个场景是测试异常。

在 MathInterface 接口中加入一个新方法，它唯一的作用就是演示如何测试异常：

```
interface MathInterface {
  // ...
  bad(foo? : any) : void;
}
```

bad 方法在调用它传入一个非数字类型的参数时会抛出一个异常：

```
public bad(foo? : any) {
  if(isNaN(foo)){
    throw new Error("Error!");
  }
    else {
      //...
  }
}
```

在下面的测试中，将看到如何使用 Chai 的异常 API 来断言一个抛出的异常：

```
it('should throw an exception when no parameters passed \n', () => {
  var math : MathInterface = new MathDemo();
  var throwsF = function() { math.bad(/* missing args */) };
```

```
  expect(throwsF).to.throw(Error);
});
```

如果你想了解更多有关断言的信息，可访问 Chai 的在线官方文档 http:// chaijs.com/api/bdd/。

Mocha 和 Chai 的 TDD 与 BDD 对比

TDD 和 BDD 遵守很多一样的原则，但它们的风格有些许不同。这两种风格提供了同样的功能，BDD 更多地考虑到能被软件开发团队的不同角色读懂（不仅仅是开发人员）。

下面的表格对比了 TDD 和 BDD 的套件、测试和断言的名字和风格：

TDD	BDD
suite	describe
setup	before
teardown	after
suiteSetup	beforeEach
suiteTearDown	afterEach
test	it
assert.equal(math.PI, 3.14159265359)	expect(Math.PI).to.equals(3.14159265359)

在随书的示例代码中，可以找到用 BDD 和 TDD 风格写的本章的所有例子。

使用 Sinon.JS 编写测试监视和测试桩

我们正在实现 MathDemo 类，而且已经用单元测试和断言测试了它的功能。现在我们将要创建一个内部使用了 MathDemo 类的小组件来执行数学运算。可以将这个新的类看作 MathDemo 类的图形化的用户接口。需要如下的 HTML：

```
<div id="widget">
  <input type="text" id="base" />
  <input type="text" id="exponent" />
```

```
  <input type="text" id="result" />
  <button id="submit" type="submit">Submit</button>
</div>
```

在随书的示例代码中，上面的 HTML 代码中包含更多的节点，例如 CSS 类，但为了让例子更加清晰易读，在这里将它们移除了。

在 source 目录下新建一个名为 `calculator_widget.ts` 的文件。HTML 将会被存储在一个 string 类型的变量中，它位于这个组件的作用域内。新类的类名为 `CalculatorWidget`，它将实现 `CalculatorWidgetInterface` 接口：

```
interface CalculatorWidgetInterface {
  render(id : string);
  onSubmit() : void;
}
```

应该在实现 `CalculatorWidget` 类之前写好单元测试，但这次我们要打破 BDD 的规则，以便更方便地理解测试桩和测试监视：

```
///<reference path="./interfaces.d.ts" />
///<reference path="../typings/tsd.d.ts" />

var template = 'HTML...';

class CalculatorWidget implements CalculatorWidgetInterface{

  private _math : MathInterface;
  private _dom = {};
  private $base: JQuery;
  private $exponent: JQuery;
  private $result: JQuery;
  private $btn: JQuery;
  constructor(math : MathInterface) {
    if(math == null) throw new Error("Argument null exception!");
    this._math = math;
  }
```

```
    public render(id : string) {
      $(id).html(template);
      this._dom.$base = $("#base");
      this._dom.$exponent = $("#exponent");
      this._dom.$result = $("#result");
      this._dom.$btn = $("#submit");
      this._dom.$btn.on("click", e => {
        this.onSubmit();
    });
  }

  public onSubmit() {
    var base = parseInt(this._dom.$base.val());
    var exponent = parseInt(this._dom.$exponent.val());

    if(isNaN(base) || isNaN(exponent)) {
      alert("Base and exponent must be a number!");
    }
    else {
      this._dom.$result.val(this._math.pow(base, exponent));
    }
  }
}
export { CalculatorWidget };
```

可以看到，我们定义了一个包含在前面审查过的 HTML 中的变量，但为了使代码简洁没有显示它。新的名为 `CalculatorWidget` 的类也包含了一个构造函数，可以观察到这个类有两个属性，一个名为 `_dom` 的变量和一个实现了 `MathInterface` 接口的 `_math`。我们依赖的是一个接口，因为可以在第 4 章看到，这是一个好的原则（依赖反转原则）。

注意，这个类的构造函数接受实现 `MathInterface` 的唯一参数。将依赖作为构造函数的参数传递也是一个好的实践，这有助于降低组件间的耦合。

这个类中第一个名为 `render` 的方法接受一个 HTML 的 ID（string 类型）作为唯一参数。使用 jQuery 选择器来用这个 ID 选择一个节点。一旦选中，之前审查过的 HTML 会被插入到这个被选中的节点内。可以说这个组件就是负责渲染 HTML 的，只需要改变它的容器，它就能很简单地被复用。这就是前端组件通常的工作方式：它们是父程序内的可

复用的独立组件，而不是简单的一些组件。

在渲染了 HTML 后，render 方法创建了每一个表单组件的快捷引用，并且初始化了一个单击事件监听：

```
public render(id : string) {
  $(id).html(template);
  this._dom.$base = $("#base");
  this._dom.$exponent = $("#exponent");
  this._dom.$result = $("#result");
  this._dom.$btn = $("#submit");

  this._dom.$btn.on("click", e => {
    this.onSubmit();
  });
}
```

当用户单击了 ID 为 submit 的按钮时，一个事件被触发，然后事件监听就会调用我们接下来可以看到的 onSubmit 方法。这个方法会读取之前快捷引用中的指数和底数：

```
  public onSubmit() {
    var base = parseInt(this._dom.$base.val());
    var exponent = parseInt(this._dom.$exponent.val());
    if(isNaN(base) || isNaN(exponent)) {
      alert("Base and exponent must be a number!");
    }
    else {
      this._dom.$result.val(this._math.pow(base, exponent));
    }
  }
}
```

如果输入的值不是数字（底数和指数），会显示一个包含错误信息的警告提示用户。如果输入的值是数字则 MathDemo 类上的 pow 方法被执行，结果通过之前创建的一个引用被关联到 result 区域。

当被测试的组件被强关联到另一个组件时，编写单元测试会变成一项非常复杂的任务。在前一部分中，我们试图遵守一些好的实践，例如依赖反转或通过构造函数的依赖进行依赖注入，但有时候，即使遵循了优秀的实践，依然需要处理高度关联的代码。

测试监视、测试模拟和测试桩会解决一些高度耦合模块带来的痛苦。这些功能也可以帮助我们找到一些错误的根源。如果将依赖的组件全部替换为测试桩，测试仍然无法通过，我们就知道问题出在了这个组件内而不是外部依赖的组件。

比如，CalculatorWidget 类依赖 MathDemo 类。如果这个计算器网站出了问题，我们无法得知问题的根源在 CalculatorWidget 类还是 MathDemo 类。然而，如果为 CalculatorWidget 写一些独立的单元测试（将 MathDemo 依赖替换为测试桩），其中的一些测试失败了，我们就知道问题出在了 CalculatorWidget 类而不是 MathDemo 类。

来看一些测试示例。

测试监视

我们通过新建一个测试套件来学习如何使用测试监视。这次我们会使用 before() 和 beforeEach() 函数。当 before() 函数被调用时（在单元测试开始执行前），会创建一个新的 HTML 节点容纳组件的 HMTL。

beforeEach() 函数用来在每个测试前重置 HTML 容器。用这种方法，可以保证测试套件内每一个测试的组件都是全新的。这是一个好办法，因为它防止了一个测试对其他测试结果的可能的修改。

```
describe('BDD test example for CalculatorWidget class \n', () => {

  before(function() {
   $("body").append('<div id="widget"/>');
  });
  beforeEach(function() {
    $("#widget").empty();
  });
});
```

通常情况下，测试框架（不管哪一种语言的测试框架）不允许我们控制单元测试和测试套件的执行顺序。测试甚至可以同时通过多线程运行。因此，保证测试套件中的单元测试的独立性非常重要。

现在测试套件已经准备好了，可以为 render() 和 onSubmit() 方法新建单元测试了。先新建一个 MathDemo 的实例，它随后会传入 CalculatorWidget 的构造函数中新建的一

个名为 calculator 的实例。

render 方法随后通过 widget ID 被调用来渲染 HTML 节点内的组件。在组件渲染完成后，ID 为 base 和 exponent 的输入框被设置了一个值。

测试规范（onSubmit should be invoked when #submit is clicked）可以帮助我们了解正在测试的是单击事件。我们将使用一个测试监视观察 onSubmit()方法，所以当 ID 为 submit 的按钮被单击后，测试监视会察觉到 onSubmit()方法被调用了。

为了完成测试，将在 ID 为 submit 的按钮上触发一个单击事件，并断言 onSubmit()方法只被调用了一次：

```
it('onSubmit should be invoked when #submit is clicked', () => {
  var math : MathInterface = new MathDemo();
  var calculator = new CalculatorWidget(math);
  calculator.render("#widget");
  $('#base').val("2");
  $('#exponent').val("3");

  // 监视onSubmit
  var onSubmitSpy = sinon.spy(calculator, "onSubmit");
  $("#submit").trigger("click");

  // 当#submit被单击时，断言calculator.onSubmit会被执行
  expect(onSubmitSpy.called).to.equal(true);
  expect(onSubmitSpy.callCount).to.equal(1);
  expect($("#result").val()).to.equal("8");
});
```

测试监视允许进行多种操作：从检查一个函数被调用多少次到检查它是否被 new 操作符调用，或调用它的时候是否传入一组特定的参数。

最后一个断言帮助确保 onSubmit()在 result 输入框中能输出正确的结果。

所有可用的操作符的详细介绍在 Sinon.JS 的官方在线文档 http://sinonjs. org/docs/#sinonspy 可以找到。

测试桩

看起来我们已经测试了整个应用，但事实似乎并非如此，分析一下到目前为止测试过哪些东西：

- 已经测试了整个MathDemo类，并且知道了当pow方法被调用时它会返回正确的值。
- 知道了 CalculatorWidget 类能正确渲染 HTML。
- 知道了 CalculatorWidget 类设置了一些事件监听并如预期从 HTML 输入中读取一些值。

到目前为止，我们已经为 MathDemo 和 CalculatorWidget 类创建了一些测试，但并没有测试集成时的情况。

我们已经测试了使用 2 为底数 3 为指数的情况，但如果错误地使用了同样的指数和底数，就可能错过一个可能的错误：在 MathDemo 类的 onSubmit() 函数被调用的时候，可能 CalculatorWidget 类传入参数的顺序有误。

在本章后面，我们会看到如何生成一种报告（测试覆盖率报告），帮助找出程序中有哪些地方没有被测试。

可以通过将 CalculatorWidget 类从它的依赖 MathDemo 中独立出来测试这个可能遇到的场景，我们用测试桩来实现它。看一下下面的单元测试，实践一下测试桩。

在这个方法开始的时候，一个 MathDemo 的实例被创建，用一个测试桩靠在它的 pow 方法上。这个测试桩会将 pow 方法替换成一个新的方法。这个新的方法会断言传入参数的顺序是正确的：

```
it('pass the right args to Math.pow', (done) => {
  var math : MathInterface = new MathDemo();

  // replace pow method with a method for testing
  sinon.stub(math, "pow", function(a, b) {
    // assert that CalculatorWidget.onSubmit invokes
    // math.pow with the right arguments
    expect(a).to.equal(2);
    expect(b).to.equal(3);
    done();
```

```
    });

    var calculator = new CalculatorWidget(math);
    calculator.render("#widget");
    $('#base').val("2");
    $('#exponent').val("3");

    $("#submit").trigger("click");
});
```

当测试桩准备好的时候，一个 CalculatorWidget 实例被创建，但并不是传入一个普通的 MathDemo 实例作为它的唯一参数，我们将这个实例注入了测试桩。这样做之后，我们就不再测试 MathDemo 类了，而是在独立的环境里测试 CalculatorWidget 类。如果不通过构造函数参数进行依赖注入的话，事情会变得复杂得多。

为了完成测试，我们需要渲染计算器组件，为 ID 为 base 和 exponent 的输入框设置值，然后在 ID 为 submit 的按钮上触发单击事件。事件监听会调用 onSubmit 方法，然后它会调用 pow 方法。当参数传入的顺序不正确时，可以百分之百地确定问题的根源在 onSubmit 函数。

使用 Nightwatch.js 创建端对端测试

使用 Nightwatch.js 编写 E2E 测试是一个非常符合直觉的过程。即使是第一次遇到它，我们也可以非常容易读懂 E2E 测试并且理解它。

看看下面的代码片段，当进入这个页面的时候，测试会花 1 秒等待页面的主体显示。元素可以通过 CSS 选择器或 XPath 语法选中。如果元素可视，setValue 方法会将 2 插入到 ID 为 base 的文本输入框中，将 3 插入到 ID 为 exponent 的文本输入框中：

```
var test = {
  'Calculator pow e2e test example' : function (client) {
    client
      .url('http://localhost:8080/')
      .waitForElementVisible('body', 1000)
      .assert.waitForElementVisible('TypeScriptTesting', 100)
      .assert.waitForElementVisible('input#base' ,100)
      .assert.waitForElementVisible('input#exponent', 100)
```

```
        .setValue('input#base', '2')
        .setValue('input#exponent', '3')
        .click('button#submit')
        .pause(100)
        .assert.value('input#result', '8')
        .end();
  }
};

export = test;
```

随后测试会找到 ID 为 submit 的按钮并触发一个单击事件。在 0.1 秒后，测试会断言正确的结果被显示在了 ID 为 result 的输入框中。在测试执行的过程中，我们可以在控制台中看到这些步骤。

可以通过下面的命令运行测试：

```
gulp run-e2e-test
```

记住，必须像在网络服务器上使用 BrowserSync 运行应用和执行 Selenium 一样，在运行 E2E 测试前先运行任务编译打包我们的代码。

生成测试覆盖率报告

在本章的前面，设置 Karma 的时候，我们已经增加了一些生成测试覆盖率报告的设置。来看一下 karma.conf.js 文件，并找出与测试覆盖率报告相关的设置：

```
module.exports = function(config) {
  'use strict';

  config.set({
    basePath: '',
    frameworks: ['mocha', 'chai', 'sinon'],
    browsers: ['PhantomJS'],
    reporters: ['progress', 'coverage'],
    coverageReporter: {
      type: 'lcov',
```

```
      dir: __dirname + '/coverage/'
    },
    plugins: [
      'karma-coverage',
      'karma-mocha',
      'karma-chai',
      'karma-sinon',
      'karma-phantomjs-launcher
    '],
    preprocessors: {
      '**/bundled/test/bdd.test.js': 'coverage'
    },
    files: [{
      pattern: "/bundled/test/bdd.test.js",
      included: true
    },
    {
      pattern: "/node_modules/jquery/dist/jquery.min.js",
      included: true
    }, {
      pattern:
      "/node_modules/bootstrap/dist/js/bootstrap.min.js",
      included: true
    }
  ],
  client: {
    mocha: {
      ui: "bdd"
    }
  },
      port: 9876,
      colors: true,
       autoWatch: false,
      logLevel: config.DEBUG
  });
};
```

可以看到，需要设置一个文件夹存储测试覆盖率报告。还需要在 reporter 设置中增加一个 `coverageReport` 的设置来指定报告的格式。

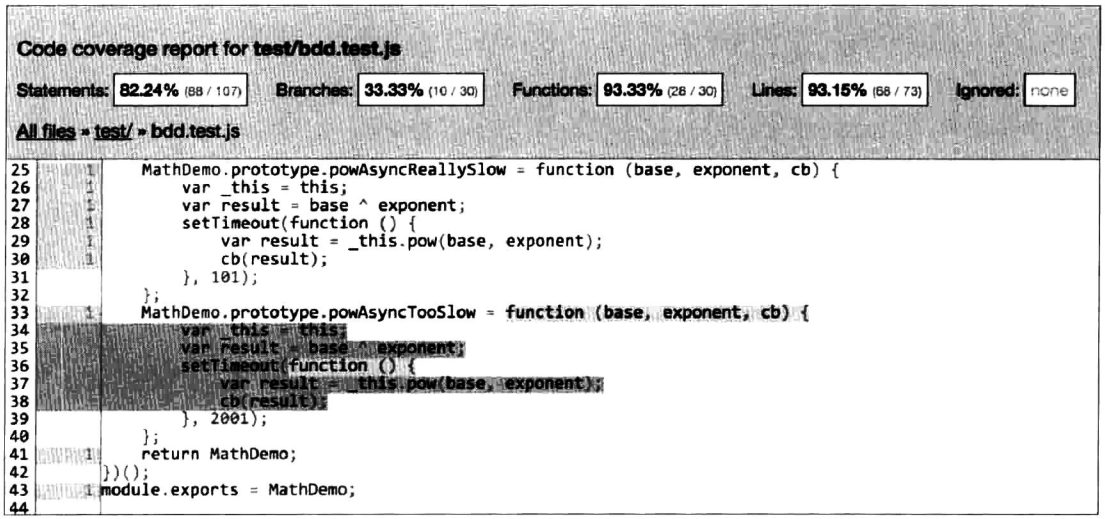
不能忘记要使用 npm 安装 `karma-coverage` 插件并且在 `karma.conf.js` 的 plugins 字段引用它。最后，需要将覆盖率报告增加到预处理字段：

```
npm karma-coverage
```

要生成覆盖率报告，只需在程序内运行所有单元测试的 Gulp 任务。可通过下面的命令执行：

```
gulp run-unit-test
```

一旦所有测试执行完毕，我们就可以进入存储覆盖率报告的文件夹，用浏览器打开其中的 `index.html` 文件。通过单击 source 下的文件名，这个 HTML 报告可以导航到各个文件的覆盖率细节。



这个报告可以帮助我们方便地找到代码中哪些地方没有被测试（红色高亮的行）。测试覆盖率报告还计算了被测试的代码占程序总代码行数的比例。可以在上面的截图中看到，例子中只有 82.24%的情况被测试了。

如果想更多地了解本章讨论的工具，强烈推荐你看一下 Ryan Roemer 所著的 *Bachbone.js Testing* 这本书。

小结

在本章中，我们讨论了一些测试的核心概念（包括测试桩、测试模拟、测试套件等），了解了测试驱动开发和行为驱动开发两种实践方式，学习了如何使用一些常用的 JavaScript 测试框架，比如 Mocha、Chai、Sinon.JS、Karma、Selenium 和 Nightwatch.js。

在本章的结尾，我们还学习了如何跨终端测试以及如何生成测试覆盖率报告。

在下一章中，我们会看到 TypeScript 1.5 引入的两个新 API——装饰器和元数据反射。

8

装饰器

在本章中，你将学习到注解和装饰器这两个 ECMAScript 6 标准中的特性，但现在，我们已经可以在 TypeScript 1.5 中使用它们了。

本章将包含以下话题：

- 注解和装饰器
  - 类装饰器
  - 方法装饰器
  - 属性装饰器
  - 参数装饰器
  - 装饰器工厂
  - 带有参数的装饰器
- 元数据反射 API

条件准备

本章中所使用的特性只有在 TypeScript 1.5 或更高的版本中才可用。我们可以像之前的章节中一样，使用 Gulp，但必须保证 `gulp-typescript` 包中运行的是最新版本的 TypeScript。让我们从创建一个 `package.json` 文件开始，然后安装会使用到的包：

```
npm init
npm install --save-dev gulp gulp-typescript typescript
```

```
npm install --save reflect-metadata
```

在安装完毕之后，我们可以创建一个 gulpfile.js 文件，然后添加一个编译代码的任务。

下面的代码展示了必需的编译器配置项。target 属性必须为 ES5，emitDecoratorMetadata 属性必须为 true。并且为了保证使用的是最新版本的 TypeScript，需要手动提供 TypeScript 的编译器包：

```
var gulp         = require("gulp"),
    tsc          = require("gulp-typescript"),
    typescript   = require("typescript");

var tsProject = tsc.createProject({
  removeComments: false,
  noImplicitAny: false,
  target: "es5",
  module: "commonjs",
  declarationFiles: false,
  emitDecoratorMetadata: true,
  typescript: typescript
});
```

在编译器设置配置完毕后，我们可以使用 gulp-typescript 插件写一个 gulp 任务：

```
gulp.task("build-source", function () {
  return gulp.src(__dirname + "/file.ts")
    .pipe(tsc(tsProject))
    .js.pipe(gulp.dest(__dirname + "/"));
});
```

注解和装饰器

注解是一种为类声明添加元数据的方法。然后，元数据就可以被诸如依赖注入容器这样的工具所使用。

注解的 API 最早由 Google 的 AtScript 团队提出，但是注解最终并没有成为语言标准。但是，装饰器由 Yehuda Katz 提出，已是 ECMAScript 7 标准的特性。它用来在代码

的设计时注释和修改类和类的属性。

注解和装饰器在外观上十分相似：

> 注解和装饰器很像是同一个东西。在使用者的角度来看，它们有相同的语法。唯一的区别就是，在使用注解时，不用去关心它是如何将元数据加入到代码里来的。装饰器则更像是一个接口，用来构建一个以注解功能结尾的东西。
>
> 目前看来，我们只需要关注装饰器，因为它才是真正的语言标准。AtScript 是 TypeScript 和 TypeScript 实现的装饰器的结合。
>
> — Pascal Precht，《注解和装饰器的区别》

我们将使用下面这个类，来说明如何使用装饰器：

```
class Person {

  public name: string;
  public surname: string;

  constructor(name: string, surname: string) {
    this.name = name;
    this.surname = surname;
  }

  public saySomething(something: string): string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

可供使用的装饰器一共有 4 种，它们分别用来装饰：类、属性、方法和参数。

类装饰器

在官方的 TypeScript 装饰器提案中，是这样定义类装饰器的：

> 类装饰器是指接受一个类构造函数作为参数的函数，并且返回 undefined、参数中提供的构造函数或一个新的构造函数。返回 undefined 等同于返回参数中提供的构造函数。
>
> — Ron Buckton，《装饰器提案-TypeScript》

类装饰器用来修改类的构造函数。如果装饰器函数返回 undefined，那么类仍然使用原来的构造函数。如果装饰器有返回值，那么返回值会被用来覆盖类原来的构造函数。

我们将要创建一个名为 logClass 的类装饰器。使用如下代码开始创建一个装饰器：

```
function logClass(target: any) {
  // ...
}
```

上面的类装饰器并没有任何逻辑，但是我们已经可以使用它装饰一个类。若想使用装饰器，需要使用@符号：

```
@logClass
class Person {
  public name: string;
  public surname: string;
  //...
```

如果已经声明并使用了一个装饰器，那么在经过 TypeScript 编译器编译后的代码中，将会有一个名为__decorate 的函数。在这里，我们不会去探究这个函数的内部实现，但是需要理解它用来在运行时装饰类。在编译了上面的 Person 类代码后，我们可以看到它在运行：

```
var Person = (function () {
  function Person(name, surname) {
    this.name = name;
    this.surname = surname;
  }
  Person.prototype.saySomething = function (something) {
      return this.name + " " + this.surname + " says: " +
      something;
  };
  Person = __decorate([
    logClass
  ], Person);
  return Person;
})();
```

现在我们已经知道了类装饰器是如何被调用的，下面来实现它的逻辑部分：

```
function logClass(target: any) {

  // 保存原构造函数的引用
  var original = target;

  // 用来生成类的实例的工具方法
  function construct(constructor, args) {
    var c: any = function () {
      return constructor.apply(this, args);
    }
    c.prototype = constructor.prototype;
    return new c();
  }

  // 新的构造函数行为
  var f: any = function (...args) {
     console.log("New: " + original.name);
     return construct(original, args);
    }

  // 复制原型，使 instanceof 操作能正常使用
  f.prototype = original.prototype;

  // 返回新的构造函数（将会覆盖原构造函数）
  return f;
}
```

类装饰器接受一个参数，即类的构造函数。这意味着 target 参数就是 Person 类的构造函数。

这个装饰器先复制了类的原构造函数，然后它定义了一个名为construct的工具函数，来生成类的实例。

装饰器用来为元素添加一些额外的逻辑或元数据。当我们想要拓展一个函数（类的方法或构造函数）的功能时，需要往原函数上包一个新函数，新函数里有额外的逻辑，且能执行原函数里的方法。

在上面的装饰器里，我们为类的构造函数添加了一些额外的逻辑，即在类的实例被创

建时，在控制台打印类的名字。为了实现它，我们创建了一个名为 f 的新构造函数。新的构造函数包含了新添加的逻辑，并且使用 construct 函数来执行原构造函数。

在装饰器函数的最后，原构造函数的原型被赋值到了新的构造函数上，来保证 instanceof 操作符使用在新的构造函数上时，也会返回正确的结果。最后，新的构造函数被返回，并且 TypeScript 编译后的代码会使用它来覆盖原构造函数。

在装饰了类的构造函数后，我们创建一个新的类实例：

```
var me = new Person("Remo", "Jansen");
```

这么做时，以下内容会被打印在控制台中：

```
"New: Person"
```

方法装饰器

在官方的 TypeScript 装饰器提案中，方法装饰器的定义如下：

> 方法装饰器函数是一个接受三个参数的函数：包含这个属性的对象、属性名（一个字符串或一个符号），和一个可选参数（属性的描述对象）。这个函数会返回 undefined、参数里提供的属性描述对象或一个新的属性描述对象。返回 undefined 等同于返回参数里提供的属性描述对象。
>
> — Ron Buckton，《装饰器提案-TypeScript》

方法装饰器和类装饰器十分相似，但它不是用来覆盖类的构造函数的，而是用来覆盖类的方法的。

如果方法装饰器返回的不是 undefined，那么返回值将会覆盖方法的属性描述对象。

属性描述对象是一个可以通过 Object.getOwnPropertyDescriptor() 方法获取到的对象。

我们先定义一个没有任何行为的名为 logMethod 的方法装饰器：

```
function logMethod(target: any, key: string, descriptor: any) {
  // ...
}
```

可以使用它装饰 Person 类中的一个方法：

```
//...
@logMethod
public saySomething(something : string) : string {
  return this.name + " " + this.surname + " says: " + something;
}
// ...
```

方法装饰器被调用时，带有以下参数：

- 包含了被装饰方法的类的原型，即 `Person.prototype`。
- 被装饰方法的名字，即 `saySomething`。
- 被装饰方法的属性描述对象，即 `Object`。

  `getOwnPropertyDescriptor(Person.prototype, saySomething)`

现在我们已经知道了各个参数的值，下面来实现这个装饰器的逻辑部分：

```
function logMethod(target: any, key: string, descriptor: any) {

  // 保存原方法的引用
  var originalMethod = descriptor.value;

  // 编辑 descriptor 参数的 value 属性
  descriptor.value = function (...args: any[]) {

    //将方法参数转化为字符串
    var a = args.map(a => JSON.stringify(a)).join();

    // 执行方法，得到其返回值
    var result = originalMethod.apply(this, args);

    // 将返回值转化为字符串
    var r = JSON.stringify(result);

    // 将函数的调用细节打印在控制台中
    console.log(`Call: ${key}(${a}) => ${r}`);

    // 返回方法的调用结果
```

```
      return result;
    }

    // 返回编辑后的属性描述对象
    return descriptor;
  }
```

和实现类装饰器时一样，以创建被装饰元素的副本开始。我们没有直接通过类的原型（`target["key"]`）来访问方法，而是通过属性描述对象（`descriptor.value`）。

然后创建了一个新函数来替代被修饰的函数。新的函数除了调用了原函数之外，还包含一些额外逻辑把函数名和每次调用时包含的参数打印到控制台里。

调用被装饰的方法后，方法名和参数将会被打印在控制台里：

```
var me = new Person("Remo", "Jansen");
me.saySomething("hello!");
// Call: saySomething("hello!") => "Remo Jansen says: hello!"
```

属性装饰器

在官方的 TypeScript 装饰器提案中，属性装饰器的定义如下：

> 属性装饰器函数是一个接受两个参数的函数：包含了这个属性的对象和这个属性的属性名（一个字符串或一个符号），不会返回一个属性描述对象。
>
> — Ron Buckton，《装饰器提案-TypeScript》

属性装饰器和方法装饰器十分相似。主要的区别在于，一个属性装饰器没有返回值且没有第三个参数（属性描述对象）。

我们创建一个名为 `logProperty` 的属性装饰器：

```
function logProperty(target: any, key: string) {
  // ...
}
```

可以使用它来装饰一个 `Person` 类的属性：

```
class Person {
  @logProperty
  public name: string;
  // ...
```

和之前一样，我们将使用一个新属性来替代原来的属性，新属性会表现得与原属性一致，除了在更改时会将改变的值打印到控制台中：

```
function logProperty(target: any, key: string) {

  // 属性值
  var _val = this[key];

  // 属性的 getter
  var getter = function () {
    console.log(`Get: ${key} => ${_val}`);
    return _val;
  };

  // 属性的 setter
  var setter = function (newVal) {
    console.log(`Set: ${key} => ${newVal}`);
    _val = newVal;
  };

  // 删除属性，在严格模式下，如果对象是不可配置的，
  // 删除操作将会抛出一个错误。在非严格模式下，则会
  // 返回 false 。
  if (delete this[key]) {
    Object.defineProperty(target, key, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true
    });
  }
}
```

在上面的装饰器里，我们创建了一个原属性的副本，并且分别声明了两个函数：getter（在读取属性值时会被调用）和 setter（在访问属性值时会被调用）。

在之前的装饰器中，返回值总被用来覆盖被修饰的元素，但属性装饰器没有返回值，我们不能使用返回值来覆盖被修饰属性的值，所以我们手动删除了原属性，并使用

Object.defineProperty 函数创建了新属性。

在使用上述装饰器装饰了 name 属性后，可以在控制台里观察到属性的每一次改变：

```
var me = new Person("Remo", "Jansen");
// Set: name => Remo
me.name = "Remo H.";
// Set: name => Remo H.
var n = me.name;
// Get: name Remo H.
```

参数装饰器

在官方的 TypeScript 装饰器提案中，参数装饰器的定义如下：

> 参数装饰器函数是一个接受三个参数的函数：一个包含了被装饰参数的方法的对象、方法的名字（或 undefined）和参数在参数列表中的索引。这个装饰器的返回值将会被忽略。
>
> —Ron Buckton，《装饰器提案-TypeScript》

让我们创建一个名为 addMetadata 的参数装饰器：

```
function addMetadata(target: any, key : string, index : number) {
  // ...
}
```

可以使用这个装饰器装饰一个参数：

```
public saySomething(@addMetadata something : string) : string {
  return this.name + " " + this.surname + " says: " + something;
}
```

参数属性没有返回值，这意味着我们将不能覆盖这个包含被修饰参数的方法。

可以使用参数装饰器来为类的原型（target）添加一些元数据。在下面的实现中，我们将会在类的原型上添加一个名为 log_${key}_ parameters 的数组，key 为包含被装饰参数的方法的名字：

```
function addMetadata(target: any, key : string, index : number) {
  var metadataKey = `_log_${key}_parameters`;
  if (Array.isArray(target[metadataKey])) {
```

```
    target[metadataKey].push(index);
  }
  else {
    target[metadataKey] = [index];
  }
}
```

为了让更多参数可以被装饰，我们将检查这个新属性是否为一个数组，如果不是，那么将初始化它为一个包含参数位置索引的数组，否则就只将参数的位置索引推入数组。

单独的参数装饰器并不是很有用，它需要和方法装饰器结合，参数装饰器用来添加元数据，然后通过方法装饰器来读取它：

```
@readMetadata
public saySomething(@addMetadata something : string) : string {
  return this.name + " " + this.surname + " says: " + something;
}
```

下面这个方法装饰器和我们在上文中实现的方法装饰器十分类似，不过它将读取通过参数装饰器添加的元数据，并且它在执行时，不再展示所有的参数，而是仅仅打印被装饰的参数：

```
function readMetadata(target: any, key: string, descriptor: any) {

  var originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {

    var metadataKey = `_log_${key}_parameters`;
    var indices = target[metadataKey];

    if (Array.isArray(indices)) {

      for (var i = 0; i < args.length; i++) {

        if (indices.indexOf(i) !== -1) {

          var arg = args[i];
          var argStr = JSON.stringify(arg) || arg.toString();
          console.log(`${key} arg[${i}]: ${argStr}`);
```

```
            }
          }
          var result = originalMethod.apply(this, args);
          return result;
        }
      }
      return descriptor;
    }
```

执行 saySomething 方法：

```
var person = new Person("Remo", "Jansen");

person.saySomething("hello!");
```

readMetadata 装饰器将会展示被添加到元数据（类原型上的_log_saySomething_parameters 属性）中的参数的值：

```
saySomething arg[0]: "hello!"
```

需要注意的是，在上面的例子里，我们使用了类原型的属性来保存原数组。在本章的后面，将会学习到使用反射元数据 API，这个 API 用来专门生成和读取元数据，在处理装饰器和元数据时，使用它是更推荐的做法。

装饰器工厂

在官方的 TypeScript 装饰器提案中，装饰器工厂的定义如下：

> 装饰器工厂是一个接受任意数量参数的函数：并且必须返回上述的任意一种装饰器。
>
> — Ron Buckton，《装饰器提案-TypeScript》

已经学习了如何实现一个类、属性、方法和参数装饰器。在大多数情况下，我们只会去使用装饰器，而不会去实现它。比如，在 Angular 2.0 中，我们将会使用 @view 装饰器来装饰一个表现为视图的类，但不需要自己来实现这个@view 装饰器。

可以使用装饰器工厂来使装饰器更容易被使用，让我们来看下面的代码：

```
@logClass
class Person {

  @logProperty
  public name: string;

  @logMethod
  public saySomething(@logParameter something: string): string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

上面代码的问题在于，我们作为开发者，需要知道 logMethod 只能被用于装饰一个方法。例子中只是因为装饰器的名字，让这个问题看上去不那么严重。

一个更好的解决方案是，让开发者使用一个@log 装饰器，无须担心自己是否使用了正确类型的装饰器：

```
@log
class Person {

  @log
  public name: string;
  public surname: string;

  constructor(name: string, surname: string) {
    this.name = name;
    this.surname = surname;
  }

  @log
  public saySomething(@log something: string): string {
    return this.name + " " + this.surname + " says: " + something;
  }
}
```

可以通过创建一个装饰器工厂来实现它。装饰器工厂指能鉴别该使用哪种装饰器并返回它的函数：

```
function log(...args: any[]) {
  switch (args.length) {
      case 1:
         return logClass.apply(this, args);
      case 2:
        // 由于属性装饰器没有返回值
        // 所以使用 break 取代 return
        logProperty.apply(this, args);
        break;
      case 3:
        if (typeof args[2] === "number") {
          logParameter.apply(this, args);
        }
        return logMethod.apply(this, args);
      default:
        throw new Error("Decorators are not valid here!");
      }
}
```

正如我们看到的，装饰器工厂通过参数的数量和类型来判断返回的装饰器类型。

带有参数的装饰器

我们可以使用一种特殊的装饰器工厂来配置装饰器的行为。例如，可以为一个类装饰器传递一个字符串参数：

```
@logClass("option")
class Person {
// ...
```

为了给装饰器传递参数，需要使用一个函数来包裹装饰器。这个包裹函数接受参数并返回一个装饰器：

```
function logClass(option: string) {
  return function (target: any) {

    // 类装饰器的逻辑
    // 我们可以访问到装饰器的参数
```

```
    console.log(target, option);
  }
}
```

这可以运用到在上文中讨论过的任意一种装饰器中。

反射元数据 API

我们已经学习了如何使用装饰器来修改和拓展类的方法与属性的行为，也学习了如何使用装饰器来为一个被装饰的类添加元数据。

对于经验不足的开发者，为类添加元数据看上去并非那么有用，但这是过去几年中，在 JavaScript 中实现的最伟大的事情之一。

我们知道，TypeScript 仅在代码设计时使用类型，但一些特性，诸如依赖注入、运行时类型断言、反射和测试，需要运行时的类型信息才可以实现。

这将不再是问题，因为我们可以使用装饰器来生成元数据，这些元数据会携带类型信息，它们可以在运行时被处理。

当 TypeScript 团队思考怎么让开发者生成类型元数据时，他们保留了一些特殊的装饰器名。

当时的想法是，当一个元素被保留的装饰器装饰后，编译器会自动地添加类型信息到元素上。这些保留的装饰器如下所示。

> TypeScript 编译器将会在这些保留装饰器的参数中添加额外的信息。
>
> @type：被装饰目标序列化后的类型。
>
> @returnType：被装饰目标若为函数，则为其序列化后的返回类型，否则为 undefined。
>
> @parameterTypes：如果它是一个函数、undefined 或其他类型，这是被装饰目标序列化后的参数类型。
>
> @name：被装饰目标的名字。
>
> Jonathan Turner，《装饰器头脑风暴》

不久之后，TypeScript 团队决定使用反射元信息 API（ES7 特性之一）来替代这些保留装饰器。

它的思想与使用保留装饰器十分相似，但使用了反射元信息 API 替代保留装饰器，来获取元信息。TypeScript 文档中定义了三种保留元数据键：

> 类型元数据使用元数据键：design:type。
> 参数类型元数据使用元数据键：design:paramtypes。
> 返回值元数据使用元数据键：design:returntype。
>
> Issue #2577，TypeScript 官方 GitHub 仓库

让我们来看看如何使用反射元数据 API，这需要引用并导入一个名为 `reflect-metadata` 的 npm 包：

```
/// <reference path="./node_modules/reflect-metadata/reflect-
metadata.d.ts"/>

import 'reflect-metadata';
```

为了测试，我们将创建一个类，而在运行时，去获取类的一个属性的类型。使用一个名为 `logType` 的属性装饰器来装饰类属性：

```
class Demo {
  @logType
  public attr1: string;
}
```

我们将调用 `Reflect.getMetadata()`方法并传入 `design:type` 键，而不是使用保留装饰器`@type` 来获取属性类型。类型将以函数的方式返回，例如，若类型是字符串，将会返回 `function String(){}`函数。可以使用 `function.name` 属性来以字符串的形式获取到类型：

```
function logType(target: any, key: string) {
  var t = Reflect.getMetadata('design:type', target, key);
  console.log(`${key} type: ${t.name}`);
}
```

如果编译上述代码，然后在浏览器中执行编译后的 JavaScript 代码，将会在控制台看到 `attr1` 属性的类型：

```
'attr1 type: String'
```

注意，上述例子只有在导入了 reflect-metadata 库后才可运行。

也可以用同样的方式来使用其他保留元数据键。让我们创建一个包含许多参数的方法，然后使用保留元数据键 design:paramtypes 来获取这些参数的类型：

```
class Demo {
  @logParamTypes
  public doSomething(
    param1: string,
    param2: number,
    param3: Foo,
    param4: { test: string },
    param5: IFoo,
    param6: Function,
    param7: (a: number) => void
  ): number {
    return 1;
  }
}
```

这次，将使用保留元数据键 design:paramtypes，因为我们查询的是多个参数的类型，所以 Reflect.getMetadata() 函数会返回一个数组：

```
function logParamTypes(target : any, key: string) {
  var types = Reflect.getMetadata('design:paramtypes', target, key);
  var s = types.map(a => a.name).join();
  console.log(`${key} param types: ${s}`);
}
```

如果我们在浏览器中执行编译后的代码，将能在控制台中看到各个参数的类型：

```
'doSomething param types: String, Number, Foo, Object, Object,
Function, Function'
```

类型以一定的规则被序列化。可以看到，函数被序列化为 Fucntion、对象字面量（{test : string}），接口被序列化为 Object，等等。

类型	序列化
void	undefined
string	String
number	Number
boolean	Boolean
symbol	Symbol
any	Object
enum	Number
Class C{}	C
Object literal {}	Object
interface	Object

值得注意的是，一些开发者已经要求 TypeScript 拥有通过元数据访问接口的类型和类的继承树上的类型的特性。这个特性被称为复杂类型序列化，在写本书之时，该特性尚不可用。但 TypeScript 团队已经着手开始实现它。

最后，我们将创建一个拥有返回类型的方法，然后使用保留元数据键 design:returntype 来获取返回值的类型：

```
class Demo {
  @logReturntype
  public doSomething2(): string {
    return "test";
  }
}
```

和上面两个例子一样，我们需要执行 Reflect.getMetadata()函数，传入保留元数据键 design:returntype：

```
function logReturntype(target, key) {
  var returnType = Reflect.getMetadata('design:returntype', target, key);
  console.log(`${key} return type: ${returnType.name}`);
}
```

如果在浏览器中执行编译后的代码，我们将能在控制台中看到方法返回值的类型：

```
'doSomething2 return type: String'
```

小结

在本章中，我们学习了如何使用和实现 4 种可用的装饰器（类、方法、属性和参数），如何创建一个装饰器工厂来抽象不同类型的装饰器。

还学习了如何使用反射元数据 API 在运行时访问类型信息。

在下一章中，我们将会学习一个 TypeScript 应用的结构，还会学习如何使用一些设计模式，以及如何创建一个单页面 Web 应用。

9

应用架构

在前面几章中，我们已经了解了 TypeScript 许多方面的特性，现在应该有充足的自信去创建一个小型应用了。

我们知道，TypeScript 是微软为方便开发大型 JavaScript 应用而创造的。一些 TypeScript 特性例如模块和类，可以为创建大型应用提供便利，但这些还不够。从长远来看，如果想要成功创建大型应用，还需要一个优秀的程序架构。

本章将深入两个主要的部分。第一个部分，我们将探索单页应用（SPA）架构和一些能帮我们创建可扩展、可维护程序的设计模式。这个部分涵盖了以下的主题：

- 单页应用架构
- MV* 架构
- 数据模型（model）和数据模型集合（collection）
- 单数据视图（item view）和集合数据视图（collection view）
- 控制器（controller）
- 事件（events）
- 路由（router）和 hash 导航
- 中介器（mediator）
- 客户端渲染和 virtual DOM
- 数据绑定和数据流
- Web 组件和 shadow DOM
- 选择一个 MV* 框架

在本章的第二个部分中，我们会将第一部分的理论和概念付诸实践。我们将从零开始

开发一个单页应用框架，它将会被用在第 10 章中。

单页应用架构

我们从探索*单页应用*（SPA）和它是如何运作的开始。许多单页应用框架可以帮我们用优秀的架构开发应用。

也可以直接深入其中一个框架，但在用之前应理解它的组件是如何运作的，我们将在第一部分研究一个 SPA 框架的内部架构。

一个 SPA 就是一个 Web 应用，它所需的资源（HTML、CSS、JavaScript 等）在一次请求中就加载完成，或不需刷新地动态加载。我们使用术语“单页”来指这种应用，因为页面在初始化加载后就永远不会重新加载（刷新）。

在过去，Web 应用仅仅是一组静态 HTML 文件和一些超链接，每当单击一个超链接时，一个新的页面就被加载进来。这会影响 Web 应用的性能，因为许多内容会重复载入（比如页头、页脚、侧边栏、脚本文件）。

当浏览器支持 AJAX 之后，开发者开始使用 AJAX 请求动态加载一些页面内容，以避免不必要的网页重载，从而提供更好的用户体验。AJAX 应用和 SPA 的工作原理非常类似。它们之间最显著的区别是，AJAX 应用是以 HTML 的形式加载应用的一部分，这些部分在它们加载完成之后就被插入到 DOM 中。而另一方面，SPA 会避免加载 HTML，而加载数据和客户端模板，这些模板和数据通过*客户端渲染*的过程在浏览器中被处理和转化成 HTML。这些数据通常使用 XML 或 JSON 格式，它们被许多客户端模板语言所支持。

让我们比较一下两者的实现。比如，AJAX 程序使用 HTML 展示一个客户和订单的列表的实现，我们会先加载一个包含了客户列表的 HTML 格式的初始页面。在这个表格中，每个客户占一行：

```
<tr>
  <td>Client Name 1</td>
  <td>
    <a href="javascript: void(0);" class="orders_link" data-client-id="1">
    View Orders
    </a>
  </td>
  <!-- more columns... -->
</tr>
```

现在不需要新建任何文件或文件夹。这些例子都是为了证明理论，而不是为实现或者运行。

在用户单击 `View Orders` 链接后，我们需要一些 JavaScript 代码通过 AJAX 来加载客户订单：

```
$(document).ready(){

  // 加载和显示客户订单
  function displayOrders(userId){
    $.ajax({
      method: "GET",
      url: `/client/orders.aspx?id=${userId}`,
      dataType: "html",
      success : function(html) {
        $("#page_container").html(html);
      },
      error : function(e) {
        var msg = "<h1>Sorry, there has been an error!</h1>";
        $("#page_container").html(msg);
      }
    });
  }

  // 设置单击事件
  $('.orders_link').on('click', function(e) {
    var userId = $(e.currentTarget).data("client-id");
    displayOrders(userId);
  });
}
```

如果你不是很理解上面的例子，可以访问 Handlebars.js（`http://handlebarsjs.com/`）和 jQuery AJAX（`http://api.jquery.com/jquery.ajax/`）在线文档获得帮助。

上面的代码片段使用了 document-ready 事件等待页面加载完成。随后在 class 标签为 orders_link 的元素上增加了一个单击事件处理函数。

事件处理函数从 data-client-id 标签上获取用户 ID 并将它传入 displayOrders 函数。displayOrders 函数使用 AJAX 请求订单列表。订单列表是 HTML 格式的，且不需要格式化就被插到 DOM 中。

在 SPA 中，和上面的流程非常相似。初始化页面的加载和 AJAX 程序完全一样。在 SPA 中，导航到一个新的页面由 JavaScript 事件控制，通常被一个叫作路由的模块管理。

让我们暂时忽略 SPA 中的导航，而关注数据加载和渲染。在 SPA 中，我们不会以 HTML 的形式加载订单列表，将使用 JSON 或 XML 格式加载它。如果使用 JSON，请求的响应看起来应该是下面这个样子：

```
{
  "orders" : [
    {
      "order_id" : 32423234,
      "currency" : "EUR",
      "date" : "13-02-2015",
      "items" :[
        { "product_id" : 13223523, "price" : 150.00, "quantity": 2 },
        { "product_id" : 62352355, "price" : 50.00, "quantity": 1 }
      ]
    },
    {
      "order_id" : 32423786,
      "currency" : "EUR",
      "date": "13-02-2015",
      "items" :[
        { "product_id" : 13228898, "price" : 60.00, "quantity" : 1 }
      ]
    }
  ]
}
```

可以像上一个例子那样使用 AJAX 请求：

```
function getOrdersData(userId : number, cb){
  $.ajax({
```

```
    method: "GET",
    url: `/api/orders/${userId}`,
    dataType: "json",
    success : function(json) {
      cb(json);
    },
    error : function(e) {
      var msg = "<h1>Sorry, there has been an error!</h1>";
      $("#page_container").html(msg);
    }
  });
}
```

在浏览器能展示订单列表之前，我们需要通过一个模板系统将它转化成 HTML，目前有很多模板系统可供选择，我们在例子中将使用 Handlebars。下面来看一下其中一个模板的语法：

```
{{#each orders}}
  <tr>
    <td>{{order_id}}</td>
    <td>{{date}}</td>
    <td>
      <ul>
        {{#each items}}<li> {{product_id}} x {{quantity}}
        </li>{{/each}}
      </ul>
    </td>
  </tr>
{{/each}}
```

Handlebars 模板语言中的元素被双括号包裹（{{和}}）。上面的模板的开始部分是一个 each 流程控制语句。each 语句会为数组中的每一个元素重复一些命令。如果我们看一下响应的 JSON，会发现订单列表是一个数组。模板会为 orders 数组的每一个对象重复{{#each}}和{{/each}}之间的操作。

每一次迭代创建一个新的 HTML 表格行。为了在 HTML 输出中显示 JSON 字段的一个值，需要将这些字段使用双括号包裹起来。比如，当需要渲染一个包含订单编号的单元格时，需要用{{order._id}}。

当在模板中引用一个 JSON 的字段时，这个字段必须在当前的作用域内。作用域可以通过 this 关键字访问，比如{{this.order_id}}等价于{{order_id}}。在使用一些流程控制语法时，模板中的作用域会改变。比如，{{# each orders}}语句会将 this 指向数组内当前的元素。

为了使用 Handlebars 模板，需要加载并编译它。我们通过一个普通的 AJAX 请求加载它：

```
function getOrdersTemplate(cb){
  $.ajax({
    method: "GET",
    url: "/client/orders.hbs",
    dataType: "text",
    success : function(templateSource) {
      var template = Handlebars.compile(source);
      cb(template);
    },
    error : function(e) {
      var msg = "<h1>Sorry, there has been an error!</h1>";
      $("#page_container").html(msg);
    }
  });
}
```

在上面这个例子中，我们使用 AJAX 请求加载了一个模板，并使用 Handlebars 的编译方法进行编译。

在一个真实生产环境中的网站，模板通常由持续集成系统编译。模板在它们加载完之后就可以使用了。预编译模板可以提升应用性能。

我们已经创建了两个函数：一个用来加载模板并编译它，另一个用来加载 JSON 数据。最后一步是将它们放在一起并生成包含订单列表的 HTML 表格：

```
function displayOrders(userId){
  getOrdersData(userId, function(data) {
```

```
      getOrdersTemplate(data, function(template) {
        var html = template(json);
        $("#page_container").html(html);
      });
    });
  }
```

SPA 看起来可能需要更多的工作量，并且它与 AJAX 应用相比可能性能更差，因为它既需要更多的操作，也需要更多的请求。然而，这和真实的情况相差甚远。要理解 SPA 的好处，我们需要先了解为什么会出现 SPA。

SPA 的出现受到两个事件的深度影响：其一是功能强大的移动和平板设备数量的迅速增长，其二是同一时期的 JavaScript 性能的迅速增长。

当移动设备变得流行起来，许多公司被迫开发 Web 应用的移动版本。它们开始开发提供 JSON 或 XML 格式数据的网络服务，供不同的客户端使用，这样做可以减少公司的开销。

问题在于，如果没有一个客户端渲染系统，现存的 AJAX 应用就不能享受到 Web 服务的好处。Mustache（Handlebars 的前身）这样的模板系统在这个时候出现，解决了这个问题。

SPA 的主要好处之一是我们需要一个 HTTP API。一个 HTTP API 相比在服务端渲染一个 HTML 页面有诸多好处，比如为网络服务编写单元测试会简单得多，因为断言数据的正确比断言用户交互函数简单得多。HTTP API 还可以被其他很多客户端程序所用，这样可以节约大量的成本并能开拓新的业务线，比如将 HTTP API 作为产品销售。

另一个 SPA 的重要好处是大量的工作都在浏览器中完成，服务端承担更少的工作，这样就可以处理更多的请求。而客户端的性能并没有受到显著影响，因为近年来设备的性能与 JavaScript 性能都有了显著的提升。

SPA 的网络性能与传统 AJAX 应用相比有好有坏。请求的响应格式是 HTML，响应的数据量有时候比 JSON 和 XML 大。

使用 XML 和 JSON 额外的开销是，我们需要一个额外的请求获取模板。可以通过预编译模板、缓存机制和将多个模板拼接成一个大的模板来减少请求数量。

MV* 架构

可以看到，在 SPA 中有很多传统在服务端的任务转移到了客户端，这样就增加了 JavaScript 代码的数量，从而我们需要更好地组织代码。

近几年，开发者们开始在前端使用后端领域较为成功的设计模式。其中，我们要重点关注 Model-View-Controller（MVC）设计模式和一些衍生版本，例如 Model-View-ViewModel（MVVM）和 Model-View-Presenter（MVP）。

全世界的开发者开始分享一些通过某种方式实现 MVC 但又无须严格遵循 MVC 模式的 SPA 框架。这些框架主要实现了 Model 和 View，但它们之中不是所有的都实现了 Controller，我们将这些类似的框架称为 MV*。

我们将在本章的稍后部分解释 MVC、Model 和 View 等概念。

现在来看一些 MV* 框架中的架构原则、设计原则和组件。

MV* 框架中的组件和功能

我们已经知道了单页应用通常使用 MV* 之类的框架开发，也浏览了一些基本的 SPA 架构设计原则。

下面来深入了解一些 MV* 框架中常见的组件和功能。

在这部分中，我们将会展示一些框架中的代码片段，并不是尝试学习如何使用这些框架，而且也不需要你有任何这些框架的使用经验。我们的目标是理解 MV* 框架中常见的模块，而不是专注于某一个框架。

model

model 是一个用来存储数据的组件。这些数据通常从 HTTP API 请求过来并显示在 view 上。一些框架包含了一个 model 实现，我们开发时需要继承它。比如，使用 Backbone.js（一个流行的 MV* 框架）时，model 必须继承 `Backbone.Model` 类：

```
class TaskModel extends Backbone.Model{
  public created : number;
  public completed : boolean;
  public title : string;
```

```
    constructor() {
      super();
    }
}
```

一个 model 继承了一些方法，可以帮助我们与网络服务进行通信。比如，以 Backbone.js 的 model 为例，我们可以使用 fetch 方法从网络服务请求数据，并将它设置到 model 中。在一些框架中，model 包含了与网络服务进行通信的方法，而另一些框架中则有单独的模块负责与 HTTP API 通信。

在其他框架中，model 可能就是一个纯粹的类，它不需要继承或实例化框架中的类:

```
class TaskModel {
  public created : number;
  public completed : boolean;
  public title : string;
}
```

collection

collection 用来表示一组 model。在上一小节中，我们看到了名为 TaskModel 的例子。model 表示列表中的一个任务，而一个 collection 表示一个任务的列表。

在支持 collection 的主流 MV* 框架中，我们需要在 collection 声明时指定 collection 承载的元素类型。比如，以 Backbone.js 为例，Task Collection 像下面这样声明：

```
class TaskCollection extends Backbone.Collection<TaskModel> {
  public model : TaskModel;
  constructor() {
    this.model = TodoModel;
  super();
  }
}
```

如同 model 的例子，一些框架中的 collection 就是纯粹的数组，我们也不需要继承或实例化框架中的任何类。collection 也可以继承一些方便与网络服务进行通信的方法。

item view

框架提供给我们最主要的功能就是 item view（或者就叫 view）组件。view 负责将存储

在 model 中的数据渲染成 HTML。view 通常依赖在构造函数、属性或设置中传入一个 model、一个模板和一个容器。

- model 和模板用来生成 HTML，就像本章前面提到过的。
- 容器通常是一个 DOM 元素选择器，被选中的 DOM 元素作为 HTML 的容器，HTML 将会被插入或附加进去。

比如，在 Marionette.js（一个流行的基于 Backbone.js 的 MV* 框架）中，一个 view 被这样声明：

```
class NavBarItemView extends Marionette.ItemView {
  constructor(options: any = {}) {
    options.template = "#navBarItemViewTemplate";
    super(options);
  }
}
```

collection view

一个 collection view 是一种特殊的 view。它与 view 的关系就好比 model 与 collection 的关系。collection view 通常依赖在构造函数、属性或者设置中传入一个 collection、一个 item view 和一个容器。

一个 collection view 迭代 collection 里面存储的 model，使用 item view 渲染它，然后将结果追加到容器尾部。

在主流的框架中，渲染一个 collection view 实际上是为每一个 collection 中的 model 渲染一个 item view，这可能会造成性能瓶颈。一种替代的方案是，使用一个 item view 和一个属性为数组的 model，然后使用{{# each}}语句在 view 的模板中渲染这个列表，而不是为 collection 中的每一个元素都渲染一个 view。

下面的代码是 Marionette.js 中的一个 collection view：

```
class SampleCollectionView extends Marionette.
CollectionView<SampleModel> {
  constructor(options: any = {}) {
```

```
    super(options);
  }
}
var view = new SampleCollectionView({
  collection : collection,
  el:$("#divOutput"),
  childView : SampleView
});
```

controller

一些框架提供了 controller 功能，controller 通常负责管理特定的 model 和相关 view 的生命周期。它的职责是实例化 model 和 collection，将它们关联起来，并与相关的 view 联系起来，在将控制权交给其他 controller 前销毁它们。

MVC 应用的交互是通过组织 controller 和它的方法。controller 在需要的时候可以拥有许多方法，而这些方法和用户的行为一一对应。

接下来看一段代码示例，它使用了 Chaplin 框架。和 Marionette.js 类似，它也基于 Backbone.js。这段代码继承了 Chaplin 的 Controller 类：

```
class LikesController extends Chaplin.Controller {
  public beforeAction() {
    this.redirectUnlessLoggedIn();
  }
  public index(params) {
    this.collection = new Likes();
    this.view = new LikesView({collection: this.collection});
  }
  public show(params) {
    this.model = new Like({id: params.id});
    this.view = new FullLikeView({model: this.model});
  }
}
```

在上面的代码片段中，可以看到，名为 `LikesController` 的 controller，它拥有 `index` 和 `show` 两个方法。可以看到在每个方法执行前，一个名为 `beforeAction` 的方法被 Chaplin 执行。

事件

事件是指被程序发现的行为或发生的事情，而且它可能会被程序处理。MV*框架通常区分两种事件。

- **用户事件**：程序允许用户通过触发和处理事件的形式沟通，比如单击一个按钮、滚动屏幕或提交一个表单。用户事件通常在 view 中处理。
- **程序事件**：应用自身也可以触发和处理一些事件。比如，一些程序在 view 渲染后触发 `onRender` 事件，或在 controller 的方法调用前触发 `onBeforeRouting` 事件。

程序事件是遵循 SOLID 原则中的开/闭原则的一个好的方式。可以使用事件来允许开发者扩展框架，而不需要对框架做任何修改。

程序事件也可以用来避免组件间的直接通信。我们将在本章后面部分在讲解中介器的时候涵盖这些内容。

路由和 hash (#) 导航

路由负责观察 URL 的变更，并将程序的执行流切换到 controller 的相应方法上。

主流框架使用了一种叫作 hash 导航的混合技术，它使用 HTML5 的 History API 在不重载页面的情况下处理页面 URL 的变更。

在 SPA 中，链接通常包含一个 hash (#)字符。这个字符原本的设计是导航到页面的一个 DOM 元素上，但它被 MV* 框架用来做无须刷新的导航。

为了理解这个概念，我们将从头开始实现一个基本的路由。我们从了解一个 MV* 框架中的路由——一个代表 URL 的纯对象是什么样子的开始：

```
class Route {
  public controllerName : string;
  public actionName : string;
  public args : Object[];

  constructor(controllerName : string, actionName : string, args :
  Object[]){
    this.controllerName = controllerName;
    this.actionName = actionName;
    this.args = args;
  }
}
```

路由观察浏览器 URL 的变更。当 URL 变更时，路由会解析它并生成一个新的路由实例。

一个最基本的路由是这样的：

```
class Router {
  private _defaultController : string;
  private _defaultAction : string;

  constructor(defaultController : string, defaultAction : string) {
    this._defaultController = defaultController || "home";
    this._defaultAction = defaultAction || "index";
  }

  public initialize() {

    // 观察用户改变URL的行为
    $(window).on('hashchange', () => {
      var r = this.getRoute();
      this.onRouteChange(r);
    });
  }

  // 读取URL
  private getRoute() {
    var h = window.location.hash;
    return this.parseRoute(h);
  }

  // 解析URL
  private parseRoute(hash : string) {
    var comp, controller, action, args, i;
    if (hash[hash.length - 1] === "/") {
        hash = hash.substring(0, hash.length - 1);
    }

    comp = hash.replace("#", '').split('/');
```

```
        controller = comp[0] || this._defaultController;
        action = comp[1] || this._defaultAction;

        args = [];
        for (i = 2; i < comp.length; i++) {
          args.push(comp[i]);
        }
        return new Route(controller, action, args);
      }

      private onRouteChange(route : Route) {
        // 在此处执行控制器
      }
    }
```

在本章的后面部分中，我们将从零开始开发一个 MV* 框架，到时候将会使用上面这个类的扩展版。

上面这个类使用默认 controller 和默认方法的名字作为它的构造函数的参数。当没有参数被传入时，home 和 index 作为默认 controller 名和默认方法名。

initialize 方法被用来创建 hashchange 事件的监听。浏览器会在 window.location.hash 变更的时候触发这个事件。

比如，当前页面的 URL 是 http://localhost:8080。一个用户单击了下面的链接：

```
<a href="#tasks/index">View Tasks</a>
```

当这个链接被单击时，window.location.hash 的值会变成"/task/index"。浏览器地址栏中的地址会变更，但 hash 字符会阻止浏览器重载当前页面。随后路由会使用 parseRoute 调用 getRoute 方法将 URL 转变成一个新的 Route 类实例。

URL 遵循下面的命名规则：

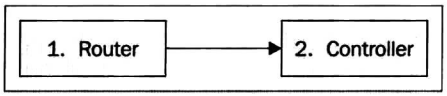
```
#conrollerName/actionName/arg1/arg2/arg3/argN
```

这意味着 task/index URL 会被转换成：

```
new Route("task", "index", []);
```

主流的 MV* 框架会使用 HTML History API 隐藏 URL 上的 hash 字符，但我们不会在框架中实现这个功能。

Route 类的实例被传入 onRouteChange 方法中，它将负责调用处理这个路由的 controller。



我们没有介绍 onRouteChange 的实现，因为在本章后面的部分，讲中介器和调度器的时候会与这个函数相关。

这就是 hash 导航和路由的基本用法。可以预测到，在一个真实的框架中，一个路由有许多其他功能，但上面的例子可以帮助我们理解主流框架中的路由是如何工作的。

中介器

一些 MV* 框架引入了一个叫作中介器的组件。中介器是一个简单的对象，所有其他的模块都通过它与其他部分进行通信。

中介器通常实现于发布/订阅设计模式（也叫 pub/sub），这种模式可以让模块之间不用相互依赖。模块之间通过事件通信，而不是直接使用程序中其他的部分。

模块可以监听一个事件并处理它，也可以发布一个事件让其他模块响应这个事件。这保证了程序模块间的低耦合，也能轻松实现信息交换。

中介器还能让开发者轻松扩展（通过订阅事件）框架而不需要对框架代码做任何改动。如我们在第 4 章中看到的，这是非常好的特性，因为它遵循了 SOLID 原则中的开/闭原则。

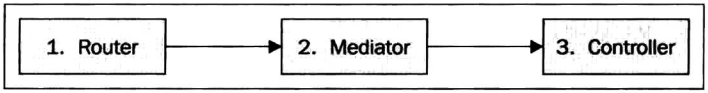
我们不深入研究中介器的实现，但可以看一个中介器的接口示例：

```
interface IMediator {
  publish(e : IAppEvent) : void;
  subscribe(e : IAppEvent) : void;
  unsubscribe(e : IAppEvent) : void;
}
```

在上一节中，我们没有介绍路由是如何调用一个 controller 的，因为框架将使用中介器来开发这个功能：

```
class Router {
  // ...
  private onRouteChange(route : Route) {
    this.meditor.publish(new AppEvent("app.dispatch", route, null));
  }
}
```

在上面的代码中，路由避免了直接调用 controller，而是使用中介器发布一个事件。



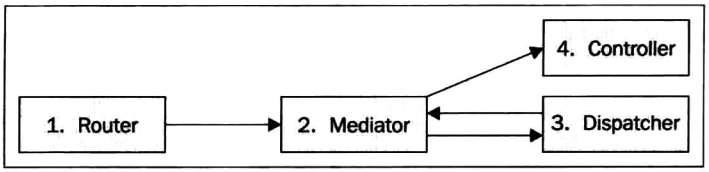
调度器

在上面的代码中可能会有一些东西引起了你的注意：app.dispatch 这个事件名。

app.dispatch 事件指向一个叫作调度器的东西。这意味着路由在向调度器发送事件而不是 controller：

```
class Dispatcher {
  // ...
  public initialize() {
    this.meditor.subscribe(
      new AppEvent("app.dispatch", null, (e: any, data? : any) => {
        this.dispatch(data);
      })
    );
  }

  // 创建和销毁controller实例
  private dispatch(route : IRoute) {
    // 1. 销毁旧的controller
    // 2. 创建新的控制器实例
    // 3. 通过中介器触发控制器的action
  }
  // ...
}
```

可以在代码片断中看到，调度器的职责是创建新的 controller 和销毁旧的 controller。当路由完成对 URL 的解析后，它将会通过中介器向调度器传入一个新的路由实例。然后调度器会销毁旧的 controller 并创建一个新的 controller，并使用中介器调用 controller 上的方法。



客户端渲染和 Virtual DOM

我们已经熟悉了基本的客户端渲染，知道了客户端渲染需要一个模板和一些数据来生成 HTML，但还没有意识到一些关于性能的细节，这些都是我们选择一个 MV* 框架时需要考虑到的。

操作 DOM 是造成 SPA 性能瓶颈的主要原因之一。因此，在决定使用哪种 MV* 框架之前，比较它们的渲染原理是非常有用的。

有一些框架在 model 发生更改的时候进行渲染，有两种可能的方式可以知道一个 model 是否发生了改变：

- 第一种是使用定时器检测变更，这个操作有时候被称为脏检测（但不是 Angular.js 中的脏检测）。
- 第二种是使用 observable model。

observable 的实现比使用定时器更高效，因为 observable 仅在变更发生的时候触发。而定时器会在时间符合条件的时候触发，不管是否有变更发生。

何时渲染非常重要，如何渲染也非常重要。一些框架直接操作 DOM，而另一些框架在内存中操作被称为 Virtual DOM 的 DOM 映射。Virtual DOM 更加高效，因为 JavaScript 对内存的操作比对 DOM 的操作更加快速。

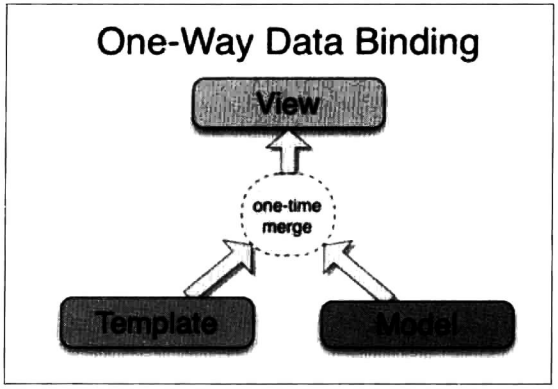
用户界面数据绑定

用户界面（UI）数据绑定是一种旨在简化图形界面开发的设计模式。UI 数据绑定将一个 UI 元素和程序 model 绑定在一起。

一个绑定会将两个属性关联在一起，当其中一个改变时，另外一个的值将自动更新。绑定可以将同一对象或不同对象上的元素联系在一起。大多数 MV* 框架都实现了某种 view 和 model 间的绑定。

单向数据绑定

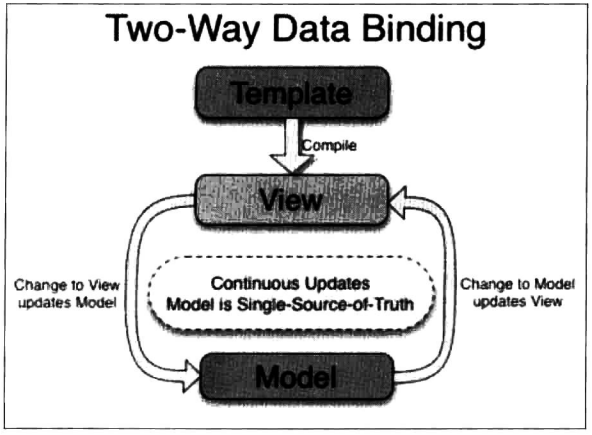
单向数据绑定是一种 UI 数据绑定的类型，这种类型的数据绑定仅单向传播变更。



在主流的 MV* 框架中，单向数据绑定意味着只有 model 的变更会传递到 view 中，而 view 上的变更不会被传递给 model。

双向数据绑定

双向数据绑定用来确保 model 和 view 中所有的变更都会传递给对方。

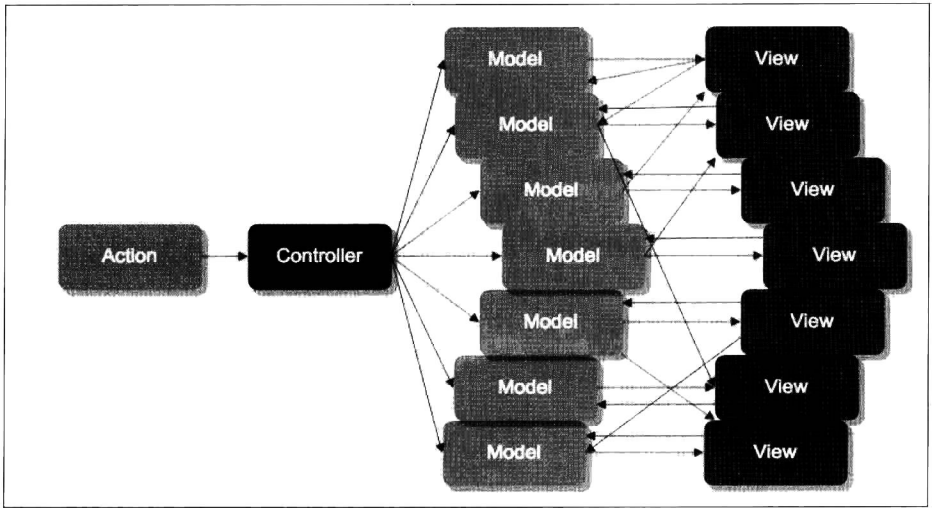


数据流

一些最新的 MV* 框架引入了新的实现和技术，其中一个新概念就是单向数据流（由 Flux 提出）。

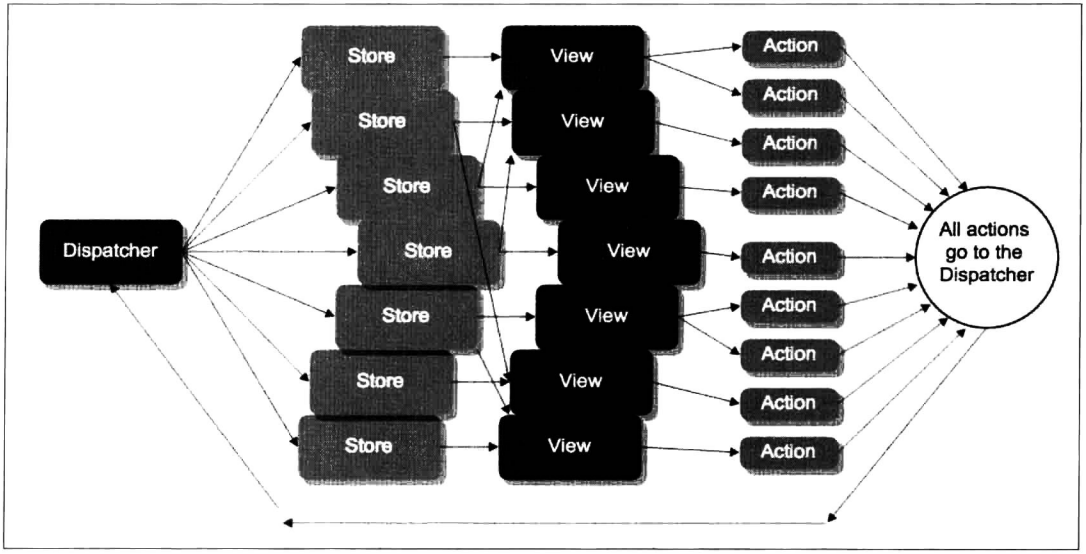
单向数据流基于这样的想法——一个变量值每次改变，都会导致依赖于该变量的其他变量重新计算自己的值。

在 MVC 应用中，一个 controller 处理多个 model 和 view。有时，一个 view 使用不止一个 model，当使用双向数据绑定时，我们最终会面临非常复杂的数据流。下图显示了这样的场景：



在这张图中，Action 不是指 controller 中的 Action，指的是程序中的用户事件或程序事件。

数据流架构试图通过把数据的流动限制为唯一的渠道和方向，来解决这个问题。这样做之后，程序组件内的数据流动就非常容易跟踪了。下面是一张展示了基于单项数据流的程序的数据流动示意图：



上图显示了数据是如何单向流动的。

在 Flux 的单向数据流架构中，所有的 Action 都直接发送到 Dispatcher 中。Flux 中的 Dispatcher 和我们框架中的调度器非常相似，但它并不是将执行流转移给 controller，而是交给一个 Store。

Store 用来存储和操作数据，类似于 MVC 中的 model。当数据被修改时，它就会被传递给 view。

View 和 MVC 中的 view 一样，负责将数据渲染成 HTML 并处理用户事件（Action）。如果一个事件需要修改一些数据，View 会将这个 Action 送入 Dispatcher 中，而不是直接对 model 进行修改，而这种场景会在双向数据绑定的框架中发生。

数据永远朝着一个方向移动，并且形成一个环，与一些提供双向数据绑定的 MVC 程序相比，这让程序的执行流非常清晰且可预测。

Web component 和 shadow DOM

一些框架使用 Web component 术语来指代那些可以重用的 UI 组件。Web component 允许用户自定义 HTML 元素，比如可以定义一个新的 HTML `<map>`标签来显示一个地图。Web component 可以单独引入它们自己的依赖并且使用一种叫 shadow DOM 的客户端模板渲染 HTML。

shadow DOM 让浏览器能在 Web component 中使用 HTML、CSS 和 JavaScript。

shadow DOM 技术在开发大型应用时非常有用，因为它可以避免模块之间的 CSS、HTML 和 JavaScript 冲突。

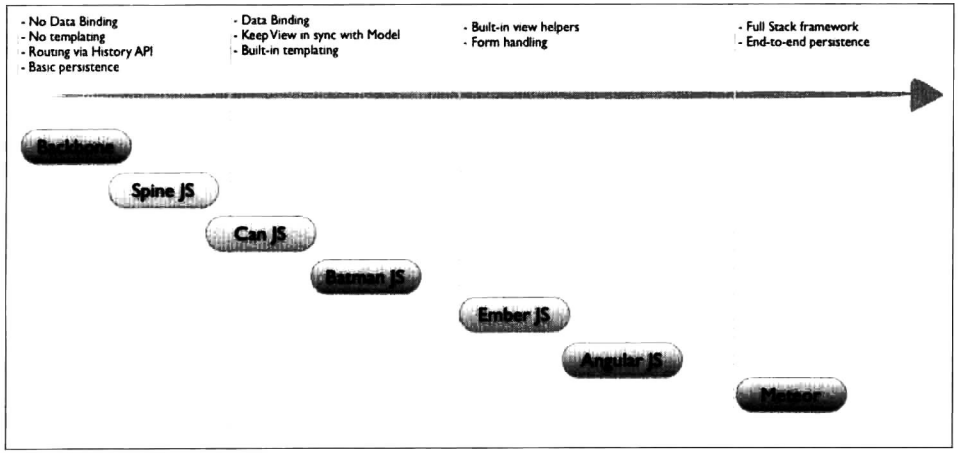
一些现存的 MV*框架（例如 Polymer），可以用来实现真正的 Web component。而一些其他框架比如 React，使用 Web component 术语指代那些可复用的 UI 组件，这些组件并非真正的 Web component，因为它们没有用到 Web component 的相关技术（自定义元素、HTML 模板、shadow DOM 和 HTML 导入）。

选择一个程序框架

我们可以从零开始开发一个 SPA，但在实现自己的框架之前通常需要先选择一个框架。选择 JavaScript SPA 框架唯一的问题就是框架实在是太多了。

> 最新最好的 JavaScript 框架每 6 分钟就会出现一个。
>
> —*Allen Pike*

我个人建议从需要实现的功能来考虑框架和其他依赖。



比如我们的应用没有复杂的 view 和表单，Backbone.js 和基于它的实现（Marionette.js、Chaplin 等）就能很好地满足我们的需求。然而，如果程序由复杂的 view 和大量的表单组成，Ember.js 或 Angular.js 可能更合适。

如果在选择框架时需要额外的帮助，可以访问 `http://todomvc.com`。TodoMVC 是一个提供使用不同 MV* 框架实现相同程序（一个任务管理应用）的项目。

从零开始实现一个 MVC 框架

我们已经对一个常见的 MV* 程序中的组件有了很好的概念性了解，现在即将从零开始实现自己的框架。

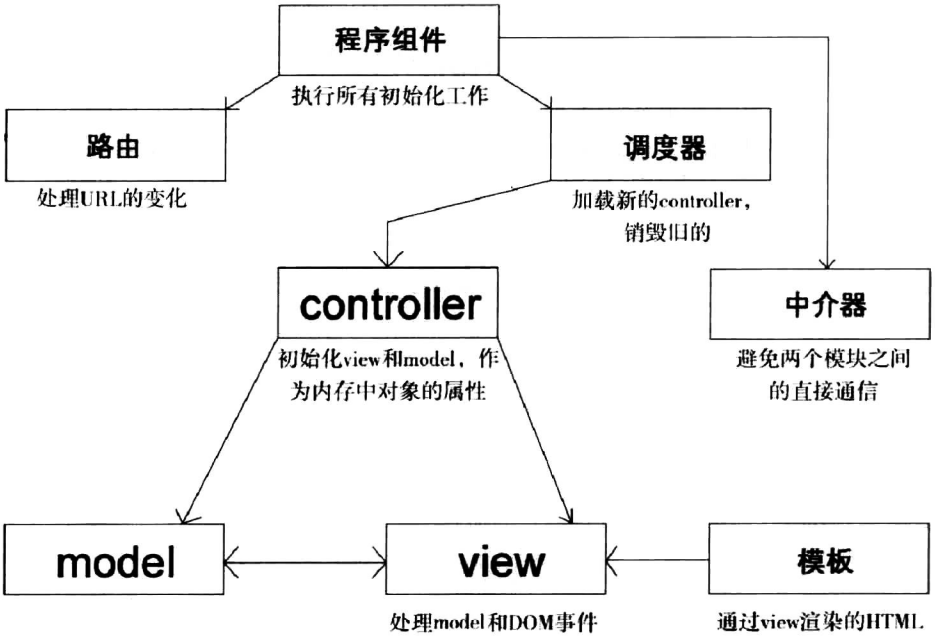
我们即将开发的框架并不是设计用在生产环境中。真实的 MV* 框架有着上千的功能并被开发了几个月甚至几年才趋于稳定。
我们即将开发的框架并不是最高效的、最易于维护的 MV* 框架，但它是极好的学习素材。

程序将实现 controller、模板、view 和 model，还有一个路由、一个中介器和一个调度器。先来看看它们在框架中的职责。

- **程序组件**：这是一个程序的根组件。程序组件负责初始化框架内所有的内部组件（中介器、路由和调度器）。
- **中介器**：中介器负责程序中所有其他模块间的通信。
- **程序事件**：程序事件被用来将信息从一个组件发送到另一个。组件可以发布程序事件也可以订阅或取消订阅一个程序事件。
- **路由**：路由观察浏览器 URL 的变更，并在变更时创建一个 `Route` 类的实例，通过程序事件传递给调度器。
- **路由表**：它被用来表示一个 URL。URL 命名规则可以指明哪一个 controller 的方法在特定路由下被调用。
- **调度器**：调度器接收一个 `Route` 类的实例，这个实例被用来指定依赖的 controller。如果需要的话，调度器会销毁上一个 controller 并新建一个。一旦 controller 被创建，调度器使用程序事件将程序执行流交给 controller。
- controller：controller 被用来初始化 view 和 model。一旦 view 和 model 初始化完成，controller 就将执行流交给一个或多个 model。

- **model**：model 负责与 HTTP API 通信，并在内存中维护这些数据，这涉及数据的格式化和对数据的增减。一旦 model 完成了对数据的操作，它就将被传递到一个或者多个 view 中。
- **view**：view 负责加载并编译模板。一旦模板编译完成，它就会等待 model 传入数据。当收到数据后，它会和模板一起被编译成 HTML 代码并插入 DOM。view 也负责绑定和解绑 UI 事件（click、focus 等）。

下图可以帮助我们理解上面介绍的组件之间的联系。



现在已经有了对框架整体架构的思路，让我们开始一个新的项目。

准备工作

在本书的前几章中提到过，建议使用 Gulp 来配置一个新项目的自动化工作流。

你可以尝试跟着下面的步骤一步步地创建这个框架，也可以在随书的代码中获取一个已完成的程序。

我们从用 npm 安装下面的项目依赖开始：

```
npm init
npm install animate.css bootstrap datatables handlebars jquery q --
save
```

还需要安装下面的开发依赖：

```
npm browser-sync browserify chai gulp gulp-coveralls gulp-tslint
gulp-typescript gulp-uglify karma karma-chai karma-mocha karma-sinon
mocha run-sequence sinon vinyl-buffer vinyl-source-stream --save-dev
```

现在让我们通过 tsd 安装必需的类型描述文件：

```
tsd init
tsd install jquery bootstrap handlebars q chai sinon mocha
jquery.dataTables highcharts --save
```

程序使用了下面的目录结构:

```
├── LICENSE
├── README.md
├── css
│   └── site.css
├── data
│   ├── nasdaq.json
│   └── nyse.json
├── gulpfile.js
├── index.html
├── karma.conf.js
├── node_modules
├── package.json
├── source
│   ├── app
│   │   └── // Chapter 10
│   └── framework
│       ├── app.ts
│       ├── app_event.ts
│       ├── controller.ts
│       ├── dispatcher.ts
│       ├── event_emitter.ts
```

```
|       ├── framework.ts
|       ├── interfaces.ts
|       ├── mediator.ts
|       ├── model.ts
|       ├── route.ts
|       ├── router.ts
|       ├── tsconfig.json
|       └── view.ts
├── test
├── tsd.json
└── typings
```

在本章，我们将主要在 source 目录下工作。在下一章，将使用我们开发的框架开发一个应用，大部分的程序文件将在 app 目录下。

让我们开始一步步实现框架的组件吧。

最终版本的框架和程序包含在随书的代码中。

程序事件

我们使用程序事件来让两个组件进行通信。比如，当一个 model 完成了从 HTTP API 接收数据，这个请求的响应将被 model 使用程序事件传递到 view。

我们在第 4 章中已经看到过了，SOLID 原则中有一个依赖反转原则，这意味着不应该依赖一个具体的类而是依赖它的抽象（接口）。我们将尝试遵循 SOLID 原则，所以在 framework 文件夹内新建一个 `interfaces.ts` 文件并加入 `IAppEvent` 接口：

```
interface IAppEvent {
  topic : string;
  data : any;
  handler: (e: any, data : any) => void;
}
```

一个应用事件包含了一个标志、主题、一些数据或一个事件处理函数。我们将在发布

或者订阅事件后更加了解这些属性。

在 framework 文件夹内创建一个名为 app_event.ts 的文件继续创建框架，可以将下面的代码复制进去：

```
/// <reference path="./interfaces"/>

class AppEvent implements IAppEvent {
  public guid : string;
  public topic : string;
  public data : any;
  public handler: (e: Object, data? : any) => void;

  constructor(topic : string, data : any, handler: (e: any, data? :
  any) => void) {
    this.topic = topic;
    this.data = data;
    this.handler = handler;
  }
}
export { AppEvent };
```

上面的代码声明了一个 AppEvent 类并实现了 IAppEvent 接口。

中介器

我们已经知道了中介器是一个用来实现发布/订阅设计模式的组件，它用来避免模块间的直接调用。

在 interfaces.ts 文件中加入一个新的接口：

```
interface IMediator {
  publish(e : IAppEvent) : void;
  subscribe(e : IAppEvent) : void;
  unsubscribe(e : IAppEvent) : void;
}
```

从上面的代码中可以看到，IMediator 暴露了三个方法来实现发布/订阅模式，它们分别如下所示。

- publish：它用来触发事件。当发布一个事件时，所有订阅事件的地方都会收到通知。
- subscribe：它用来订阅一个事件，换句话说，为一个事件设置事件处理。
- unsubscribe：它用来取消订阅一个事件，换句话说，移除一个事件的处理函数。

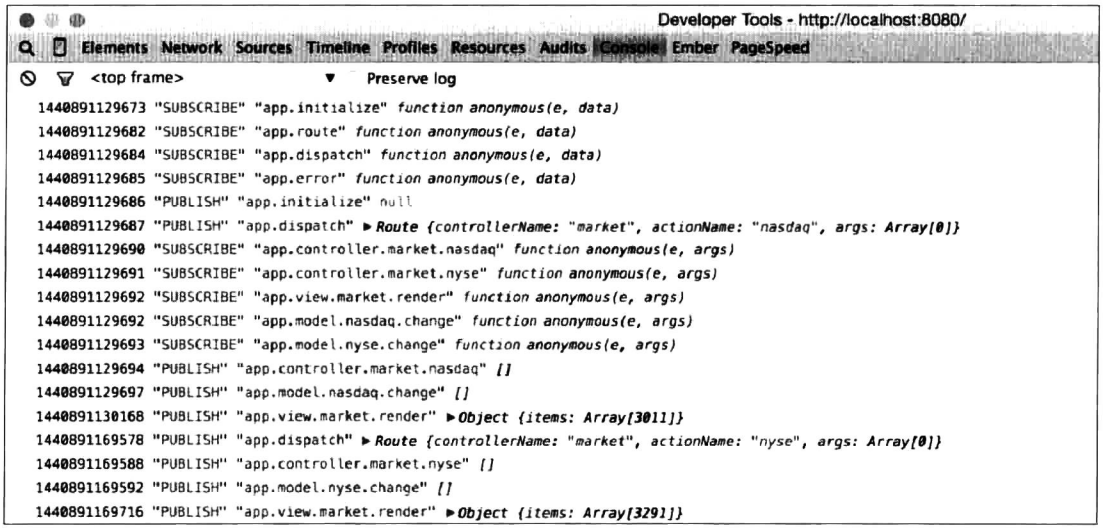
现在让我们在 framework 文件夹下新建一个名为 mediator.ts 的文件，并将下面的代码加入其中：

```
/// <reference path="./interfaces"/>

class Mediator implements IMediator {
  private _$ : JQuery;
  private _isDebug;

  constructor(isDebug : boolean = false) {
    this._$ = $({});
    this._isDebug = isDebug;
  }

  public publish(e : IAppEvent) : void {
    if(this._isDebug === true) { console.log(new Date().getTime(),
      "PUBLISH", e.topic, e.data);
    }
    this._$.trigger(e.topic, e.data);
  }

  public subscribe(e : IAppEvent) : void {
    if(this._isDebug === true) { console.log(new Date().getTime(),
      "SUBSCRIBE", e.topic, e.handler); }
    this._$.on(e.topic, e.handler);
  }

  public unsubscribe(e : IAppEvent) : void {
    if(this._isDebug === true) { console.log(new Date().getTime(),
    "UNSUBSCRIBE", e.topic, e.data); }
    this._$.off(e.topic);
```

```
  }
}
export { Mediator };
```

上面的代码声明了一个名为 Mediator 的类并实现了 IMediator 接口。Mediator 的构造函数有一个默认（false）的参数来指定是否开启调试模式。

调试模式非常有用，因为当它开启时，我们可以观察到所有对 publish、subscribe 和 unsubscribe 的调用。在下面的截图中，可以在浏览器控制台看到调试模式开启时的输出：



publish、subscribe 和 unsubscribe 方法使用了 jQuery 的 trigger、on 和 off 方法，分别用来执行事件处理函数、增加和移除事件处理。

默认构造函数还初始化了一个名为_$的私有属性。这个属性的值是一个在内存中的空的 jQuery 对象。这个对象被 jQuery 用来在 trigger、on 和 off 被调用时增加和移除事件处理。

值得注意的是，如果中介器在内存中被移除，它的_$属性也会被移除，所有的事件处理都会丢失。在下面的章节中，我们可以看到 App 类如何保证中介器永远也不会从内存中被清除。

程序组件

application 类是整个应用的根组件，这个类主要负责初始化框架的主要组件（路由、中介器和调度器）。

我们从声明 application 需要的接口开始，在 interfaces.ts 中加入以下接口：

```
interface IAppSettings {
  isDebug : boolean,
  defaultController : string;
  defaultAction : string;
  controllers : Array<IControllerDetails>;
  onErrorHandler : (o : Object) => void;
}

interface IControllerDetails {
  controllerName : string;
  controller : { new(...args : any[]): IController ;};
}
```

IAppSettings 接口用来指明可用的程序设置。我们可以使用程序设置来打开调试模式，设置默认 controller 的名字和它的默认方法名，设置可用的 controller，设置全局错误处理函数。来看一下实际的 application 类的实现：

```
/// <reference path="./interfaces"/>

import { Dispatcher } from "./dispatcher";
import { Mediator } from "./mediator";
import { AppEvent } from "./app_event";
import { Router } from "./router";

class App {
  private _dispatcher : IDispatcher;
  private _mediator : IMediator;
  private _router : IRouter;
  private _controllers : IControllerDetails[];
  private _onErrorHandler : (o : Object) => void;
```

```
  constructor(appSettings : IAppSettings) {
    this._controllers = appSettings.controllers;
    this._mediator = new Mediator(appSettings.isDebug || false);
    this._router = new Router(this._mediator,
    appSettings.defaultController, appSettings.defaultAction);
    this._dispatcher = new Dispatcher(this._mediator,
    this._controllers);
    this._onErrorHandler = appSettings.onErrorHandler;
  }

  public initialize() {
    this._router.initialize();
    this._dispatcher.initialize();
    this._mediator.subscribe(new AppEvent("app.error", null, (e:
    any, data? : any) => {
      this._onErrorHandler(data);
    }));
    this._mediator.publish(new AppEvent("app.initialize", null,
    null));
  }
}
export { App };
```

上面的代码声明了一个名为 App 的类，并接受一个实现了 IAppSettings 的参数作为构造函数的唯一参数。构造函数初始化了这个类的各种属性（调度器、中介器、路由、controller 和全局错误处理函数）。

当我们新建一个程序时，它将自动创建一个新的中介器，并且将它传入路由和调度器。这意味着唯一的中介器实例被程序所有的组件共享。换句话说，这个中介器是一个单例，它将在程序的生命周期内一直存在内存中。

在创建了 App 类实例后，必须调用 initialize 方法开始执行这个程序，随后将看到路由是什么时候被初始化的，它会使用中介器订阅 app.initialize 事件。

initialize 方法调用了程序组件内的一些 initialize 方法（路由和调度器）。随后它设置了全局的错误事件处理并发布了 app.initialize 事件。

随后中介器调用了路由的 app.initialize 事件的处理函数，这就解释了程序的执行

流是如何从 application 类到 router 类的。

路由表

为了理解 router 类的实现，我们需要先了解它的一些依赖的实现。第一个依赖就是 Route 类。

Route 类实现了 IRoute 接口。这个接口在本章的早些时候介绍过了，我们不再赘述它的细节：

```
interface IRoute {
  controllerName : string;
  actionName : string;
  args : Object[];
  serialize() : string;
}
```

将之前实现过的 Route 类也包含进来，但 serialize 方法在之前故意被忽略。serialize 方法将一个 Route 类实例转化成一个 URL。

```
/// <reference path="./interfaces"/>

class Route implements IRoute {
  public controllerName : string;
  public actionName : string;
  public args : Object[];

  constructor(controllerName : string, actionName : string, args :
  Object[]){
    this.controllerName = controllerName;
    this.actionName = actionName;
    this.args = args;
  }

  public serialize() : string {
    var s, sargs;
    sargs = this.args.map(a => a.toString()).join("/");
```

```
      s = `${this.controllerName}/${this.actionName}/${sargs}`;
      return s;
    }
  }
  export { Route };
```

事件发射

Router 类还依赖 EventEmitter 类。这个类非常重要，因为整个框架中除了程序组件的每一个组件都将继承这个类。

我们已经知道，所有的组件都使用中介器与其他组件进行通信。中介器是一个单例，这意味着程序中的每一个组件都需要能访问中介器实例。

EventEmitter 类就是为了减少代码重复而实现的，并提供一些方便事件发布和订阅的帮助：

```
interface IEventEmitter {
  triggerEvent(event : IAppEvent);
  subscribeToEvents(events : Array<IAppEvent>);
  unsubscribeToEvents(events : Array<IAppEvent>);
}
```

现在，我们在 framework 目录下新建一个名为 event_emitter.ts 的文件并将下面的代码复制进去：

```
/// <reference path="./interfaces"/>

import { AppEvent } from "./app_event";

class EventEmitter implements IEventEmitter{
  protected _metiator : IMediator;
  protected _events : Array<IAppEvent>;

  constructor(metiator : IMediator) {
    this._metiator = metiator;
  }

  public triggerEvent(event : IAppEvent){
```

```
    this._metiator.publish(event);
  }

  public subscribeToEvents(events : Array<IAppEvent>) {
    this._events = events;
    for(var i = 0; i < this._events.length; i++) {
      this._metiator.subscribe(this._events[i]);
    }
  }

  public unsubscribeToEvents() {
    for(var i = 0; i < this._events.length; i++) {
      this._metiator.unsubscribe(this._events[i]);
    }
  }
}
export { EventEmitter };
```

当 subscribeToEvents 方法被调用时，_events 属性被用来存储组件订阅的事件。

当一个组件决定使用 unsubscribeToEvents 方法移除事件处理时，不需要传入整个事件列表，因为事件发射器使用_events 属性记住它们了。

路由

路由检测 URL 的变更并生成 Route 类的实例，随后使用程序事件将它传入调度器。Router 类实现了 IRouter 接口：

```
interface IRouter extends IEventEmitter {
  initialize() : void;
}
```

让我们看一下 Router 类的内部实现：

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";
import { Route } from "./route";
```

```
class Router extends EventEmitter implements IRouter {
  private _defaultController: string;
  private _defaultAction: string;

  constructor(metiator: IMediator, defaultController: string,
  defaultAction: string) {
    super(metiator);
    this._defaultController = defaultController || "home";
    this._defaultAction = defaultAction || "index";
  }

  public initialize() {

    // 监测URL被用户改变
    $(window).on('hashchange', () => {
      var r = this.getRoute();
      this.onRouteChange(r);
    });

    // 拥有改变URL的能力
    this.subscribeToEvents([

      // 用于在应用启动时触发路由
      new AppEvent("app.initialize", null, (e: any, data?: any) => {
        this.onRouteChange(this.getRoute());
      }),

      // 用于从其他组件改变URL
      new AppEvent("app.route", null, (e: any, data?: any) => {
      this.setRoute(data); }),
    ]);
  }

  // 读取URL
  private getRoute() {
    var h = window.location.hash;
```

```
        return this.parseRoute(h);
    }

    // 改变URL
    private setRoute(route: Route) {
        var s = route.serialize();
        window.location.hash = s;
    }

    // 解析URL
    private parseRoute(hash: string) {
        var comp, controller, action, args, i;
        if (hash[hash.length - 1] === "/") {
            hash = hash.substring(0, hash.length - 1);
        }
        comp = hash.replace("#", '').split('/');
        controller = comp[0] || this._defaultController;
        action = comp[1] || this._defaultAction;

        args = [];
        for (i = 2; i < comp.length; i++) {
            args.push(comp[i]);
        }
        return new Route(controller, action, args);
    }

    // 通过中介器将控制权转移给调度器
    private onRouteChange(route: Route) {
        this.triggerEvent(new AppEvent("app.dispatch", route, null));
    }
}
export { Router };
```

我们在本章前面已经见过这个类了，但在这里它有一些显著的区别。这次 Route 类继承了 EventEmitter 类，并接受一个中介器和一个默认 controller 的名字和 controller 上的默认方法名作为构造函数参数。

initialize 方法现在包含了对 subscribeToEvents 方法的调用，用来增加对

app.initialize 事件的处理。这个事件保证了在应用首次启动时路由解析了 URL。路由会监测 URL 的变更，但在程序初始化时并没有 URL 变更，所以程序不会调用任何 controller，路由使用 app.initialize 事件来解决这个问题。

路由还订阅了 app.route 事件，该事件的处理函数使用了一个名为 setRoute 的方法设置浏览器 URL。app.route 程序事件允许其他组件导航到一个路由下。

最后，可以看到名为 parseRoute 的方法，它将 URL 转变成一个 Route 类的实例和 onRouteChange 方法，被用来发布 app.dispatch 程序事件。

调度器

调度器被用来在需要时创建与销毁 controller。销毁 controller 是非常重要的，因为一个 controller 会使用到大量的 model 和 view，它会消耗不少内存。

如果有很多 controller，对内存的占用就会变成性能问题。调度器的一个职责就是避免潜在的这种问题。

调度器实现了 IDispatcher 和 IEventEmitter 接口：

```
interface IDispatcher extends IEventEmitter {
  initialize() : void;
}
```

让我们看一下 Dispatcher 类的具体实现：

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";

class Dispatcher extends EventEmitter implements IDispatcher {
  private _controllersHashMap : Object;
  private _currentController : IController;
  private _currentControllerName : string;

  constructor(metiator : IMediator, controllers :
  IControllerDetails[]) {
    super(metiator);
    this._controllersHashMap = this.getController(controllers);
```

```
    this._currentController = null;
    this._currentControllerName = null;
  }
```

现在应该对中介器如何工作非常熟悉了。每一个组件都继承自 EventEmitter 类，并且都用它在初始化时订阅一些事件。

在本章的后面部分，我们将会观察一些拥有 dispose 方法的类（Controllers、Views 和 Models），它们使用 dispose 取消订阅 initialize 方法中订阅的消息。

```
// 监听app.dispatch事件
public initialize() {
  this.subscribeToEvents([
    new AppEvent("app.dispatch", null, (e: any, data? : any) => {
      this.dispatch(data);
    })
  ]);
}
```

下面方法中的 hash 表被用来尽快寻找 controller，当一个新的路由表需要被调度时，下面这个方法生成一个 hash 表，它使用 controller 的名字作为键，使用 controller 的构造函数作为值：

```
private getController(controllers : IControllerDetails[]) : Object {
  var hashMap, hashMapEntry, name, controller, l;

  hashMap = {};
  l = controllers.length;

  if(l <= 0) {
    this.triggerEvent(new AppEvent(
        "app.error",
        "Cannot create an application without at least one
        contoller.",
        null));
  }

  for(var i = 0; i < l; i++) {
    controller = controllers[i];
```

```
    name = controller.controllerName;
    hashMapEntry = hashMap[name];
  if(hashMapEntry !== null && hashMapEntry !== undefined) {
    this.triggerEvent(new AppEvent(
        "app.error",
        "Two controller cannot use the same name.",
        null));
    }
    hashMap[name] = controller.controller;
  }
  return hashMap;
}
```

下面这个方法负责创建、初始化、销毁一个 controller 的实例，下面的代码能帮助对它的理解：

```
private dispatch(route : IRoute) {
  var Controller = this._controllersHashMap[route.controllerName];

  // 试图发现controller
  if (Controller === null ||    Controller === undefined) {
    this.triggerEvent(new AppEvent(
      "app.error",
      `Controller not found: ${route.controllerName}`,
      null));
    }
    else {
      // 创建一个controller实例
      var controller : IController = new
      Controller(this._metiator);

      // 该行为不可用
      var a = controller[route.actionName];
      if (a === null || a === undefined) {
        this.triggerEvent(new AppEvent(
          "app.error",
          `Action not found in controller: ${route.controllerName}
           -  + ${route.actionName}`,
```

```
            null));
        }
        // 该行为可用
        else {
          if(this._currentController == null) {
            // 初始化controller
            this._currentControllerName = route.controllerName;
            this._currentController = controller;
            this._currentController.initialize();
          }
          else {
            // 若之前的controller不再需要，则销毁
            if(this._currentControllerName !== route.controllerName) {
              this._currentController.dispose();
              this._currentControllerName = route.controllerName;
              this._currentController = controller;
              this._currentController.initialize();
            }
          }
          // 将流从调试器传递至controller
          this.triggerEvent(new AppEvent(
          `app.controller.${this._currentControllerName}.
          ${route.actionName}`,
            route.args,
            null
          ));
        }
      }
    }
  }
  export { Dispatcher };
```

在销毁了上一个 controller（如果需要）并创建了新的 controller 之后，新的 controller 被初始化。当一个 controller 被初始化时，`initialize` 方法被调用，随后开始订阅一些事件。

当调度器发布下面的 controller 已经订阅的程序事件时，执行流会传到 controller 的事件处理中：

```
`app.controller.${this._currentControllerName}.
${route.actionName}`
```

controller

controller 负责初始化和销毁 view 和 model。由于 controller 必须被调度器销毁，所以在 IController 接口中必须实现 dispose 方法：

```
interface IController extends IEventEmitter {
  initialize() : void;
  dispose() : void;
}
```

model 和 view 被设置为继承 Controller 类的类属性。Controller 类本身不提供任何功能，这意味着开发者在开发程序的时候需要自己实现上面的功能。

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";

class Controller extends EventEmitter implements IController {

  constructor(metiator : IMediator) {
    super(metiator);
  }

  public initialize() : void {
    throw new Error('Controller.prototype.initialize() is abstract you
    must implement it!');
  }

  public dispose() : void {
    throw new Error('Controller.prototype.dispose() is abstract you
    must implement it!');
  }
}
export { Controller };
```

即使不是框架的强制限定，还是推荐使用中介器使执行流从 model（不是 view）转移到 controller。

model 和 model settings

model 被用来与网络服务进行通信，并格式化它返回的数据。model 可以让我们读取、格式化、更新或删除从服务器返回的数据。model 实现了 `IModel` 和 `IEventEmitter` 接口：

```
interface IModel extends IEventEmitter {
  initialize() : void;
  dispose() : void;
}
```

需要为一个 model 提供网络服务的 URL。我们将使用一个叫作 `ModelSettings` 的装饰器来设置这个 URL。

可以通过构造函数注入 URL，但注入数据（与行为相反）到构造函数中被认为是一个不好的实践方式。下面的装饰器代码中有一些注释，方便理解：

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";

function ModelSettings(serviceUrl : string) {
  return function(target : any) {
    // 保存原构造函数的引用
    var original = target;

    // 一个用于生成类实例的工具函数
    function construct(constructor, args) {
    var c : any = function () {
        return constructor.apply(this, args);
      }
      c.prototype = constructor.prototype;
      var instance =  new c();
      instance._serviceUrl = serviceUrl;
      return instance;
    }
```

```
    // 新构造函数的行为
    var f : any = function (...args) {
      return construct(original, args);
    }

    // 为了使 instance 操作符继续可用，复制原型
    f.prototype = original.prototype;

    // 返回新的构造函数（将覆盖原来的）
    return f;
  }
}
```

在下一章中，我们会这样使用这个装饰器：

```
@ModelSettings("./data/nasdaq.json")
class NasdaqModel extends Model implements IModel {
//...
```

让我们看一下 Model 类的内部实现：

```
class Model extends EventEmitter implements IModel {

  // _serviceUrl 的值必须使用 ModelSettings 装饰器来设置
  private _serviceUrl : string;

  constructor(metiator : IMediator) {
    super(metiator);
  }

  // 必须由派生类来实现
  public initialize() {
    throw new Error('Model.prototype.initialize() is abstract and
    must implemented.');
  }

  // 必须由派生类来实现
  public dispose() {
```

```
    throw new Error('Model.prototype.dispose() is abstract and
    must implemented.');
  }

  protected requestAsync(method : string, dataType : string, data) {
    return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
      $.ajax({
        method: method,
        url: this._serviceUrl,
        data : data || {},
        dataType: dataType,
        success: (response) => {
          resolve(response);
        },
        error : (...args : any[]) => {
          reject(args);
        }
      });
    });
  }

  protected getAsync(dataType : string, data : any) {
    return this.requestAsync("GET", dataType, data);
  }

  protected postAsync(dataType : string, data : any) {
    return this.requestAsync("POST", dataType, data);
  }

  protected putAsync(dataType : string, data : any) {
    return this.requestAsync("PUT", dataType, data);
  }

  protected deleteAsync(dataType : string, data : any) {
    return this.requestAsync("DELETE", dataType, data);
  }
}
export { Model, ModelSettings };
```

和在 controller 里一样，initialize 和 dispose 方法需要被它的子类实现，所以并没有包含任何的逻辑。

requestAsync 方法被用来从网络服务或静态文件请求数据。可以看到，这个方法使用了 jQuery 的 AJAX API 和 Q 的 promise。

这个类还包含了 getAsync、postAsync、putAsync 和 deleteAsync 方法，分别对应 GET、POST、PUT、DELETE 请求。

即使不是框架的强制限制，还是推荐使用中介器来将执行流从 model 传入 view。

view 和 view settings

view 被用来渲染模板和处理 UI 事件。和程序中剩下的组件一样，View 类继承自 EventEmitter 类：

```
interface IView extends IEventEmitter {
  initialize() : void;
  dispose() : void;
}
```

需要为一个 view 提供一个模板的地址。我们将使用一个名为 ViewSettings 的装饰器来设置这个地址。

如同在 model 中一样，从构造函数中传入这个 URL 是不好的实践方式。下面这个装饰器代码中包含一些注释，方便理解：

```
/// <reference path="./interfaces"/>

import { EventEmitter } from "./event_emitter";
import { AppEvent } from "./app_event";

function ViewSettings(templateUrl : string, container : string) {
  return function(target : any) {
    // 保存原构造函数的引用
    var original = target;

    // 一个用于生成类实例的工具函数
    function construct(constructor, args) {
      var c : any = function () {
```

```
        return constructor.apply(this, args);
      }
      c.prototype = constructor.prototype;
      var instance =  new c();
      instance._container = container;
      instance._templateUrl = templateUrl;
      return instance;
    }

    // 新构造函数的行为
    var f : any = function (...args) {
      return construct(original, args);
    }

    // 为了使instanceof操作符继续可用，复制原型
    f.prototype = original.prototype;

    // 返回新的构造函数（将覆盖原来的）
    return f;
  }
}
```

在下一章中，这个装饰器会被这样使用：

```
@ViewSettings("./source/app/templates/market.hbs", "#outlet")
class MarketView extends View implements IView {
//...
```

让我们看一下 View 类。和 controller 与 model 一样，initialize 和 dispose 需要被它的子类实现，所以并不包含任何逻辑：

```
class View extends EventEmitter implements IView {

  // _container和_templateUrl 的值必须使用ViewSettings装饰器设置
  protected _container : string;
  private _templateUrl : string;

  private _templateDelegate : HandlebarsTemplateDelegate;
```

```
constructor(metiator : IMediator) {
  super(metiator);
}

// 必须由派生类来实现
public initialize() {
  throw new Error('View.prototype.initialize() is abstract and
  must implemented.');
}

// 必须由派生类来实现
public dispose() {
  throw new Error('View.prototype.dispose() is abstract and must
  implemented.');
}
```

View 类包含了两个新方法（bindDomEvents 和 unbindDomEvents），这两个方法也必须被它的子类实现。从它们的名字可以猜到，这两个方法可以设置（bindDomEvents）和解绑（unbindDomEvents）UI 事件：

```
// 必须由派生类来实现
protected bindDomEvents(model : any) {
  throw new Error('View.prototype.bindDomEvents() is abstract
  and must implemented.');
}

// 必须由派生类来实现
protected unbindDomEvents() {
  throw new Error('View.prototype.unbindDomEvents() is abstract
  and must implemented.');
}
```

下面的异步方法使用了 promise 并加载了一个模板（loadTemplateAsync），编译（compileTemplateAsync）、缓存（getTemplateAsync）后渲染（renderAsync）。除了 renderAsync 方法，其他都是私有方法，这意味着 renderAsync 必须被它的子类实现：

```
// 异步加载模板
```

```
private loadTemplateAsync() {
  return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
    $.ajax({
      method: "GET",
      url: this._templateUrl,
      dataType: "text",
      success: (response) => {
        resolve(response);
      },
      error : (...args : any[]) => {
        reject(args);
      }
    });
  });
}

// 异步编译模板
private compileTemplateAsync(source : string) {
  return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
    try {
      var template = Handlebars.compile(source);
      resolve(template);
    }
    catch(e) {
      reject(e);
    }
  });
}
// 若操作仍未完成，则异步加载和编译一个模板
private getTemplateAsync() {
  return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
    if(this._templateDelegate === undefined ||
    this._templateDelegate === null) {
      this.loadTemplateAsync()
          .then((source) => {
            return this.compileTemplateAsync(source);
          })
```

```
            .then((templateDelegate) => {
              this._templateDelegate = templateDelegate;
              resolve(this._templateDelegate);
            })
            .catch((e) => { reject(e); });
        }
        else {
          resolve(this._templateDelegate);
        }
      });
    }

    // 异步渲染一个view
    protected renderAsync(model) {
      return Q.Promise((resolve : (r) => {}, reject : (e) => {}) => {
        this.getTemplateAsync()
            .then((templateDelegate) => {
              // 生成HTML并添加到DOM中
              var html = this._templateDelegate(model);
              $(this._container).html(html);

              // 将 model 作为参数传给 model
              // 让子视图和 DOM 事件初始化
              resolve(model);
            })
            .catch((e) => { reject(e); });
      });
    }
}
export { View, ViewSettings };
```

框架

框架文件被用来从一个文件提供所有框架组件的访问入口。这意味着当我们使用这个框架实现一个应用的时候，不需要从不同的文件导入各个模块：

```
/// <reference path="./interfaces"/>
```

```
import { App } from "./app";
import { Route } from "./route";
import { AppEvent } from "./app_event";
import { Controller } from "./controller";
import { View, ViewSettings } from "./view";
import { Model, ModelSettings } from "./model";

export { App, AppEvent, Controller, View, ViewSettings, Model,
ModelSettings, Route };
```

小结

在本章中，我们学习了什么是单页应用，一个常见的组件是什么样子的，以及这个架构典型的组成是什么。

我们也构建了自己的 MV* 框架，这个实践经验和知识有助于深入理解各种现存的 MV* 框架。

在下一章中，我们尝试将本书中学到的很多概念付诸实践，并使用这一章中创造的框架开发一个完整的 SPA。

10
汇总

在本章中，我们将会把前面章节中提到的主要概念付诸实践。

我们将使用第 9 章中开发的 SPA 框架开发一个小型单页应用。

这个应用能让我们获知 NASDAQ 和 NYSE 股票在特定日期内的走势。这不是一个大型应用，但是足以证明使用 TypeScript 与优秀的程序架构的好处。

我们将会写一些类和函数，部分函数是异步的（将用到第 1 章、第 3 章、第 4 章和第 5 章中的知识），还会使用一些我们之前的 SPA 框架提供的装饰器（将用到第 8 章中的知识）。

为了完成本章内容，我们将建立一个自动化构建环境来方便开发流程（将用到第 2 章中的知识），尝试优化应用的性能（将用到第 6 章中的知识），通过编写一些单元测试和集成测试来确保它能正常工作（将用到第 7 章中的知识）。

本章旨在帮助你提高使用 TypeScript 和 SPA 框架的自信，着眼于 SOLID 原则与解耦，以建立一个可维护、可测试、高扩展性、模块可复用的应用为目标。

准备工作

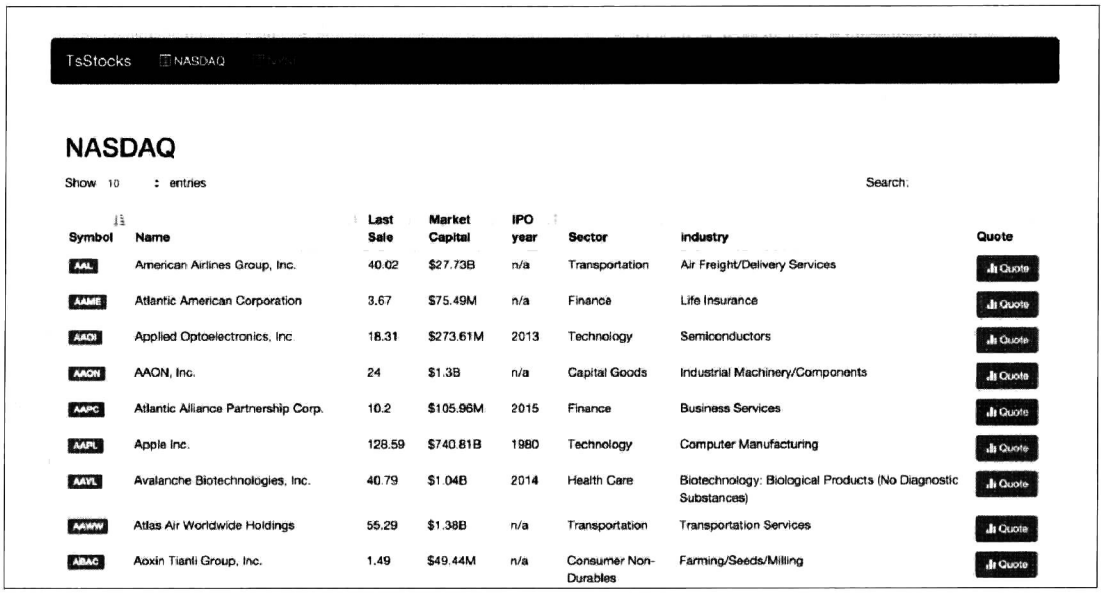
本章将继续使用上一章中介绍的工具和文件目录，你可以使用随书代码中的 `tsd.json` 和 `package.json` 文件安装 npm 包依赖和类型定义文件。在第 9 章中的“从零开始实现一个 MVC 框架”部分可以找到本程序需要的准备工作。

程序依赖

我们将开发一个允许用户查看股票代号列表的小程序，每个股票代号代表了证券交易所的一个公司。

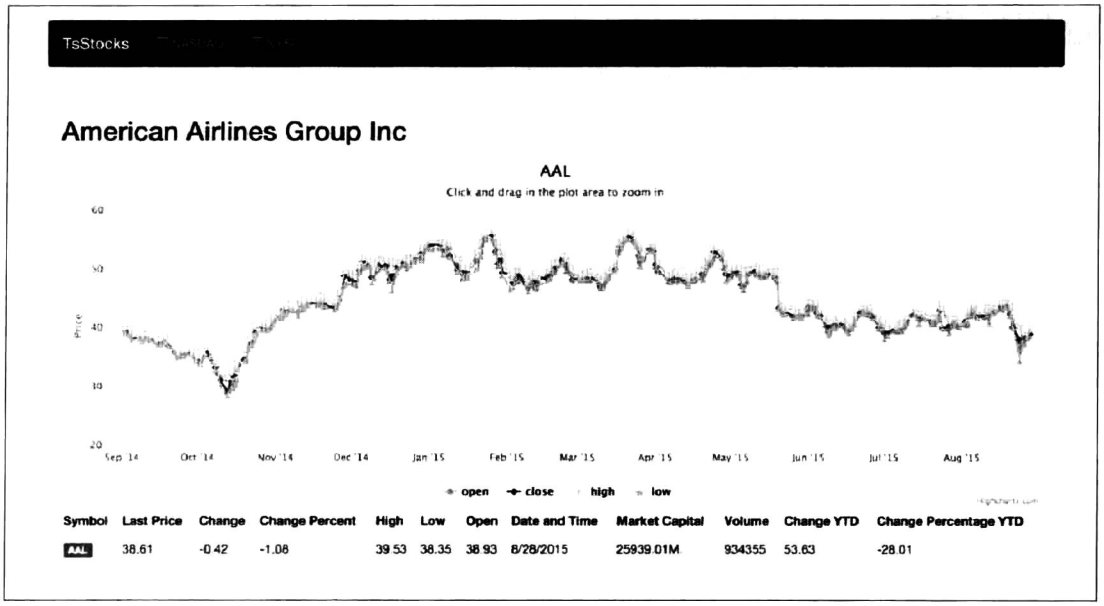
这个程序的主页将显示两个主流的证券交易所的股票代号：NASDAQ（纳斯达克）和NYSE（纽交所）。

从下方截图中可以看到，这个程序需要一个顶部菜单来包含证券交易所的股票代号列表。股票代号列表将在一个表格中显示，它包含了一些基本的股票信息，比如每股价格、剩余股数、股票名、公司名。



表格的最后一列包含了一些可跳至第二屏查看股票报价的按钮。股票的报价是给定时间内的价格细节汇总。

股票报价页面将显示一个给股票经理使用的图表，它可以看到每股的价格（y 轴）随时间（x 轴）的走势。我们可以显示多条线来表示开盘价格、收盘价格、最高价格（一天内）和最低价格（一天内）。



程序中的数据

在上一章解释过，要开发一个 SPA，需要一个后端的程序来让我们从浏览器使用 AJAX 请求数据。这意味着我们需要一个 HTTP API。

我们将使用一个免费的公共 HTTP API 来获取真实的股票报价数据。而对于可用的股票代码列表，我们使用一个静态的 JSON 文件。这个 JSON 文件通过 NASDAQ 官网提供的 CSV 文件生成。外部的 HTTP API 也提供了图表数据。

总计使用以下三组数据。

- **市场数据**：这个数据将存储在静态 JSON 文件中。这个文件从 NASDAQ 官网的 CSV 文件转换生成，可以在随书的例子中找到。
- **股价数据**：这些数据由外部的服务提供。本例中使用的是名为 Markit 公司提供的数据，他们专职提供金融信息服务。我们将使用他们的市场数据 API（v2），这是一个免费服务，并提供了很好的说明文档 `http://dev.markitondemand.com/`。
- **图表数据**：这些数据也由 Markit 的服务提供。

程序架构

我们将使用自己的框架开发一个 SPA。从上一章可以看到，我们的框架可以将一个 URL 映射到一个 controller 中的行为上。

我们的程序包含三个页面，每一个页面使用不同的 URL，如下所示。

- `#market/nasdaq` 显示 NASDAQ 交易所的股票。
- `#market/nyse` 显示 NYSE 交易所的股票。
- `#symbol/quote/{symbol}`显示选定的股票代号的股票报价。

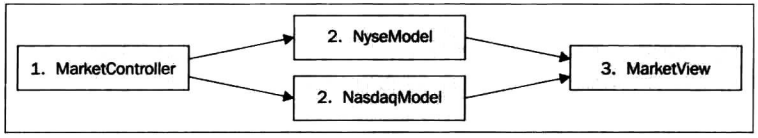
上面提到的 URL 都将会被实现为程序中 controller 的行为。在上一章中，我们看到了 URL 遵循这样的命名习惯：`#controllerName/actionName/arg1/arg2/argN`。

如果按照命名约定来推断上面列表中提及的 URL，可以知道程序将会有两个 controller：`MarketController` 和 `SymbolController`。

`MarketController` 将会被两个 model 和一个 view 实现。

- **NasdaqModel**：它将从一个静态 JSON 文件中加载一个 NASDAQ 股票列表。
- **NyseModel**：它将从一个静态 JSON 文件中加载一个 NYSE 股票列表。
- **MarketView**：它将渲染 NASDAQ 或 NYSE 股票列表。

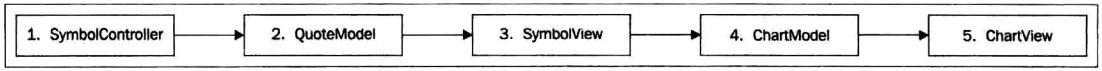
每个模块之间使用程序事件和中介与其他模块进行通信。market 页面的执行顺序如下所示：



`SymbolController` 将会被两个 model 和两个 view 实现：

- **QuoteModel**：它会加载选定股票的股票报价。
- **ChartModel**：它会加载选定股票的价格走势数据点。
- **ChartView**：它将股票价格走势显示在一个可交互图表中。
- **SymbolView**：它显示选定的股票的最近价格变化。

每个组件使用程序事件和中介器与其他组件进行通信，这一页面的执行顺序如下所示：



程序文件结构

这一节将会列出我们将要开发的程序的文件结构。在根目录中，你会找到程序的入口（index.html）和其他自动化工具的配置文件（gulpfile.js、karma.conf.js、package.json 等）。还可以观察到 typings 文件夹，里面包含了一些类型定义。

同上一章，程序源码位于 source 文件夹，单元测试和集成测试位于 test 文件夹。下面是整个程序的文件结构：

```
├── LICENSE
├── README.md
├── css
│   └── site.css
├── data
│   ├── nasdaq.json
│   └── nyse.json
├── gulpfile.js
├── index.html
├── karma.conf.js
├── node_modules
├── package.json
├── source
│   ├── app
│   │   ├── controllers
│   │   │   ├── market_controller.ts
│   │   │   └── symbol_controller.ts
│   │   ├── main.ts
│   │   ├── models
│   │   │   ├── chart_model.ts
│   │   │   ├── nasdaq_model.ts
│   │   │   ├── nyse_model.ts
│   │   │   └── quote_model.ts
│   │   ├── templates
│   │   │   ├── market.hbs
│   │   │   └── symbol.hbs
│   │   └── views
│   │       ├── chart_view.ts
│   │       ├── market_view.ts
```

```
|   |       └── symbol_view.ts
|   └── framework
|       └── framework.ts (Chapter 9)
├── test
|   ├── app
|   └── framework
├── tsd.json
└── typings
```

在 source 文件夹下，你可以观察到两个目录，app 和 framework。我们在上一章中已经创建了 framework 中的文件。这次，将关注 application 文件夹，这意味着我们大多数时间将工作在 app 目录下。

在 app 目录下，可以看到一些名为 controllers、models、templates 和 views 的文件夹。可以猜到，这些文件夹将用来存放 controller、model、view 和模板。

还可以在 app 目录下找到 main.ts 文件。这个文件是程序的入口，但因为我们使用的是 ES6 模块，所以不能在浏览器中使用<script/>加载它。

配置自动构建流程

如同在第 2 章中看到的，我们将新建一个配置文件设置 Gulp 任务。新建 gulpfile.js 文件并导入依赖的插件：

```
var gulp        = require("gulp"),
    browserify  = require("browserify"),
    source      = require("vinyl-source-stream"),
    buffer      = require("vinyl-buffer"),
    tslint      = require("gulp-tslint"),
    tsc         = require("gulp-typescript"),
    karma       = require("karma").server,
    coveralls   = require('gulp-coveralls'),
    uglify      = require("gulp-uglify"),
    runSequence = require("run-sequence"),
    header      = require("gulp-header"),
    browserSync = require("browser-sync"),
    reload      = browserSync.reload,
    pkg         = require(__dirname + "/package.json");
```

切记，在引入依赖前，应该先通过 npm 安装它们。

引入依赖后，就可以开始写我们的第一个任务了，它用来检查一些基本的命名是否符合规则，并且避免一些错误实践（检查 source 和 test 目录下的 TypeScript 文件）：

```
gulp.task("lint", function() {
  return gulp.src([
              "source/**/**.ts",
              "test/**/**.test.ts"
             ])
            .pipe(tslint())
            .pipe(tslint.report("verbose"));
});
```

还需要一个任务将 TypeScript 代码编译为 JavaScript 代码。考虑到我们使用了装饰器特性，需要保证使用 TypeScript 1.5 或以上的版本且设置了 experimentalDecorators 及编译目标，如下面的代码所示：

```
var tsProject = tsc.createProject({
  target : "es5",
  module : "commonjs",
  experimentalDecorators: true,
  typescript: typescript
});
```

设置完编译选项后，我们可以继续写一些编译任务。第一个任务将会编译程序代码：

```
gulp.task("build", function() {
  return gulp.src("src/**/**.ts")
             .pipe(tsc(tsProject))
             .js.pipe(gulp.dest("build/source/"));
});
```

第二个任务将会编译单元测试和集成测试代码。我们将使用一个新的项目对象来避免潜在可能导致的运行时错误：

```
var tsTestProject = tsc.createProject({
  target : "es5",
```

```
    module : "commonjs",
    experimentalDecorators: true,
    typescript: typescript
  });

  gulp.task("build-test", function() {
    return gulp.src("test/**/*.test.ts")
              .pipe(tsc(tsTestProject))
              .js.pipe(gulp.dest("/build/test/"));
  });
```

上面两个任务已足够生成需要的 JavaScript 代码了，但由于我们使用的是 CommonJS 模块，所以还需要编写一个任务将 CommonJS 模块打包成浏览器可加载并执行的文件。如在第 2 章中看到的，我们将新建一些 Gulp 任务并且用 Browserify 来达到这个目的：

```
gulp.task("bundle-source", function () {
  var b = browserify({
    standalone : 'TsStock',
    entries: "build/source/app/main.js",
    debug: true
  });

  return b.bundle()
    .pipe(source("bundle.js"))
    .pipe(buffer())
    .pipe(gulp.dest("bundled/source/"));
});
```

还需要打包单元测试：

```
gulp.task("bundle-unit-test", function () {
  var b = browserify({
    standalone : 'test',
    entries: "build/test/bdd.test.js",
    debug: true
  });
  return b.bundle()
    .pipe(source("bdd.test.js"))
```

```
    .pipe(buffer())
    .pipe(gulp.dest("bundled/test/"));
});
```

最后打包集成测试：

```
gulp.task("bundle-e2e-test", function () {
  var b = browserify({
    standalone : 'test',
    entries: "build/test/e2e.test.js",
    debug: true
  });
  return b.bundle()
    .pipe(source("e2e.test.js"))
    .pipe(buffer())
    .pipe(gulp.dest("bundled/e2e-test/"));
});
```

我们将在本章的后面部分再次用到这个 gulpfile.js 配置文件，增加一些任务，来运行应用程序和测试及进行部分优化。

到目前为止，我们已经在解决一个自动化工作流的配置了。从现在开始，我们将集中于程序的组件。一个组件由 4 个核心元素构成：模板、样式、服务和组件逻辑。你可以在随书的例子中找到对应的模板和样式，我们在这里只关注 TypeScript 文件（服务和组件逻辑）。

程序布局

在程序根目录下新建一个名为 index.html 的文件。下面的代码是真实 index.html 的简略版本，真实的版本包含在随书的代码中：

```
<ul class="nav navbar-nav">
  <li>
    <a href="#market/nasdaq">NASDAQ</a> </li>
  <li>
    <a href="#market/nyse">NYSE</a>
  </li>
```

```
</ul>
<div id="outlet">
  <!-- HTML GENERATED BY VIEWS GOES HERE -->
</div>
```

在上面的 HTML 中，包含了两个重要的元素。第一个重要的元素是两个链接中的 URL。这些链接包含了 hash 字符（#），它们将被程序的路由处理。

第二个重要的元素是使用了 `outlet` 为 ID 的元素，这个节点被框架用作每个新页面动态生成的 DOM 的容器。

实现根组件

在上一章中我们已经看到，MVC 框架的根组件是 App 组件。所以我们在 `source/app` 目录下新建一个名为 `main.ts` 的文件。

可以通过像下面这样增加 `source/interface.ts` 的引用来访问框架的所有接口：

```
/// <reference path="../framework/interfaces"/>
```

可以通过引用 `framework/framework.ts` 来访问框架的所有组件：

```
import { App, View } from "../framework/framework";
```

本程序中有两个 controller，虽然目前这些文件还不存在，但依然可以增加两个导入语句：

```
import { MarketController } from
"./controllers/market_controller";
import { SymbolController } from
"./controllers/symbol_controller";
```

这时候，需要创建一个字面量对象实现 `IAppSetting` 接口。这个对象允许我们进行一些基础的设置，比如设置默认的 controller 名和行为。然而这个对象中最重要的字段是 `controller` 字段，它必须是一个 `IControllerDetails` 类型的数组。如果你忘了 `IControllerDetails` 的细节，可以查看上一章中的介绍。

```
var appSettings : IAppSettings = {
  isDebug : true,
  defaultController : "market",
```

```
    defaultAction : "nasdaq",
    controllers : [
      { controllerName : "market", controller : MarketController },
      { controllerName : "symbol", controller : SymbolController }
    ],
    onErrorHandler : function(e : Object) {
      alert("Sorry! there has been an error please check out the console for
more info!");
      console.log(e.toString());
    }
  };
```

我们可以创建一个 App 的实例，并调用 intialize 方法初始化它：

```
var myApp = new App(appSettings);
myApp.initialize();
```

此时，代码尚未被编译，因为还没有定义 MarketController 和 SymbolController，下面来定义第一个 controller。

实现 market controller

在 app/controllers 目录下新建一个名为 market_controller.ts 的文件。需要导入 Controller、AppEvent 和一些其他还未实现的模块（NyseModel、NasdaqModel 和 MarketView）。

```
/// <reference path="../../framework/interfaces"/>

import { Controller, AppEvent } from "../../framework/framework";
import { MarketView } from "../views/market_view";
import { NasdaqModel } from "../models/nasdaq_model";
import { NyseModel } from "../models/nyse_model";
```

在使用我们框架的程序中，controller 必须继承 Controller 类并实现 IController 接口：

```
class MarketController extends Controller implements IController {
```

不强制规定将 view 和 model 声明为 controller 的属性，但推荐这么做：

```
private _marketView : IView;
private _nasdaqModel : IModel;
private _nyseModel : IModel;
```

同时推荐在 controller 的构造函数内设置所有依赖的值：

```
constructor(metiator : IMediator) {
  super(metiator);
  this._marketView = new MarketView(metiator);
  this._nasdaqModel = new NasdaqModel(metiator);
  this._nyseModel = new NyseModel(metiator);
}
```

比在构造函数内初始化依赖的值更好的做法是，使用一个 IoC 容器通过构造函数自动注入依赖。但是，实现一个 IoC 容器并不是一件简单的事情，它超出了本书的讨论范围。

我们还必须实现 initialize 方法。controller 中的 initialize 需要做以下事情：

- 订阅 controller 中所有可用方法的事件。在本例中，controller 有两个方法（nasdaq 和 nyse 方法）。
- 调用 View.initialize() 方法初始化 view，在本例中，只有一个 view（marketView）。
- 调用 Model.initialize() 方法初始化 model。在本例中，有两个 model（nasdaqModel 和 nyseModel）。

```
public initialize() : void {
  // 订阅controller事件
  this.subscribeToEvents([
    new AppEvent("app.controller.market.nasdaq", null, (e, args
    : string[]) => {
    this.nasdaq(args); }),
    new AppEvent("app.controller.market.nyse", null, (e, args :
    string[]) => { this.nyse(args); })
  ]);
```

```
    // 初始化view和model事件
    this._marketView.initialize();
    this._nasdaqModel.initialize();
    this._nyseModel.initialize();
  }
```

dispose 方法与 initialize 方法相反。如果在 initialize 方法中创建一个事件处理函数，那它就应该在 dispose 方法中被销毁。unsubscribeToEvents 工具方法将取消订阅所有通过 subscribeToEvents 方法订阅的事件。

```
  // 销毁 view/model并停止监听controller中的变化
  public dispose() : void {

    // 销毁 controller事件
    this.unsubscribeToEvents();

    // 销毁view和model事件
    this._marketView.dispose();
    this._nasdaqModel.dispose();
    this._nyseModel.dispose();
  }
```

在上一章中提到过，调度器会使用 controller 的 initialize 和 dispose 方法释放一些内存，一旦忘记使用 dispose 方法销毁一个类，这个 view 可能会永远存在内存中。

controller 中的方法不能有任何操作数据的行为（这是 model 的职责）或用户界面事件处理（这是 view 的职责）。理想状态下，controller 中的方法应该只是发布一个或多个事件，让代码运行流从 controller 到一个或多个 model 中。

以 nasdaq 方法为例，controller 发布了 NasdaqModel 的 initialize 中订阅的方法：

```
  // 显示NASDAQ股票走势
  public nasdaq(args : string[]) {
    this._metiator.publish(new AppEvent("app.model.nasdaq.change",
    null, null));
  }
```

以 nyse 方法为例，controller 发布了 NyseModel 的 initialize 中订阅的方法：

```
  // 显示NYSE股票走势
```

```
public nyse(args : string[]) {
  this._metiator.publish(new AppEvent("app.model.nyse.change", null,
  null));
}
```

实现 NASDAQ model

在 app/models 目录下新建一个名为 nasdaq_model.ts 的文件，随后从框架中导入 Model、AppEvent 和 ModelSettings，然后声明一个名为 NasdaqModel 的类。这个类必须继承 Model 类并实现 IModel 接口。

我们也会使用 ModelSettings 装饰器指出网络服务的路径或静态文件路径。在这个例子中，我们将使用静态数据文件，它在随书的例子中可以找到：

```
/// <reference path="../../framework/interfaces"/>

import { Model, AppEvent, ModelSettings } from "../../framework/framework;
@ModelSettings("./data/nasdaq.json")
class NasdaqModel extends Model implements IModel {

  constructor(metiator : IMediator) {
    super(metiator);
  }
```

这个 model 在 initialize 方法被调用时会订阅 app.model.nasdaq.change 事件。这正是 controller 的方法中发布的将运行流从 controller 切换到 model 的事件：

```
// 监听model事件
public initialize() {
  this.subscribeToEvents([
    new AppEvent("app.model.nasdaq.change", null, (e, args) =>
    { this.onChange(args); })
  ]);
}
```

在之前的 controller 中可以看到，所有通过 subscribeToEvents 工具方法订阅的事件，会通过 unsubscribeToEvents 工具方法取消订阅：

```
// 销毁model事件
public dispose() {
  this.unsubscribeToEvents();
}
```

下面是一个 app.model.nasdaq.change 事件的处理函数。这个事件处理函数使用 getAsync 方法从之前在 ModelSettings 装饰器中定义的服务 URL 中获取数据。getAsync 方法是从 Model 类上继承的，它是在上一章实现的。

getAsync 方法返回一个 promise，如果这个 promise fulfilled，数据将会被格式化并传给 view：

```
private onChange(args) : void {
    this.getAsync("json", args)
        .then((data) => {

          // 格式化数据
          var stocks = { items : data, market : "NASDAQ" };

          // 将程序控制权转给 market view
          this.triggerEvent(new AppEvent("app.view.market.render",
          stocks, null));
        })
        .catch((e) => {
          // 将程序控制权转给全局错误处理方法
          this.triggerEvent(new AppEvent("app.error", e, null));
        });
  }
}
export { NasdaqModel };
```

实现 NYSE model

让我们在 app/models 目录下新建一个名为 nyse_model.ts 的文件。NyseModel 和 NasdaqModel 并没有太多区别，所以我们不会深究其中的细节：

```
@ModelSettings("./data/nyse.json")
class NyseModel extends Model implements IModel {
```

```
    // ...
}
export { NyseModel };
```

我们需要做的只是将 nasdaq_model.ts 文件中的内容复制到 nyse_mode.ts 文件，并将 nasdaq 替换为 nyse。

这种类型的代码重复叫作代码异味，代码异味指那些我们需要重构（提升性能）的代码。可以使用范型来避免大多数的代码异味，然而这里没有使用范型，因为我们认为展示装饰器的用法对本书的读者更有意义。

实现 market view

在 app/views 目录下新建一个 market_view.ts 文件，随后从框架中导入 AppEvent、ViewSettings 和 Route 组件并声明 MarketView 类。这个类必须继承 View 基类并实现 IView 接口。

我们也将使用 ViewSettings 装饰器来指明一个路径、Handlebar 模板的位置和一个选择器，来寻找一个 DOM 元素作为这个 view HTML 的父节点：

```
/// <reference path="../../framework/interfaces"/>

import { View, AppEvent,ViewSettings, Route } from "../../framework/
framework";

@ViewSettings("./source/app/templates/market.hbs", "#outlet")
class MarketView extends View implements IView {
  constructor(metiator : IMediator) {
    super(metiator);
  }
```

这个 view 订阅了 app.view.market.render 事件并在 renderAsync 方法中处理它，这个方法继承自 View 基类。这个方法返回一个 promise 对象，当 ViewSettings 传入的模板被载入并编译完成后它会被 fulfilled。

为了 fulfilled 这个 promise，view 必须被成功渲染，且将模板插入 ViewSettings 装饰

器中定义的选择器所匹配的 DOM 元素中：

```
initialize() : void {
  this.subscribeToEvents([
    new AppEvent("app.view.market.render", null, (e, args : any) => {
      this.renderAsync(args)
        .then((model) => {
          // 设置DOM事件
          this.bindDomEvents(model);
        })
        .catch((e) => {
          // 将程序控制权转移到全局错误处理方法
          this.triggerEvent(new AppEvent("app.error", e,
          null)); });
    }),
  ]);
}
```

如上面提到过的 controller 和 model，unsubscribeToEvents 工具方法会取消所有 subscribeToEvents 工具方法订阅的事件。

```
public dispose() : void {
  this.unbindDomEvents();
  this.unsubscribeToEvents();
}
```

view 承担处理用户事件的责任。我们框架内的组件使用 initialize 方法订阅事件，使用 dispose 方法取消事件订阅。在用户事件的例子中，我们将使用 bindDomEvents 方法设置用户事件，用 unbindDomEvents 方法销毁它：

```
// 初始化DOM事件
protected bindDomEvents(model : any) {
  var scope = $(this._container);
  // 处理quote按钮上的单击事件
  $(".getQuote").on('click', scope, (e) => {
    var symbol = $(e.currentTarget).data('symbol');
    this.getStockQuote(symbol);
  });
```

```
    // 让表格变得可排序和可搜索
    $(scope).find('table').DataTable();
  }

  // 销毁DOM事件监听
  protected unbindDomEvents() {
    var scope = this._container;
    $(".getQuote").off('click', scope);
    var table = $(scope).find('table').DataTable();
    table.destroy();
  }
```

其中一个用户事件观察 quote 按钮的单击事件。当这个事件被触发时，下面的事件处理方法会被调用：

```
private getStockQuote(symbol : string) {
  // 使用route事件跳转路由
  this.triggerEvent(new AppEvent(
  "app.route",
  new Route("symbol", "quote", [symbol]), null));
}
```

可以看到，这个事件处理方法新建了一个路由，并发布了一个 app.route 事件。这将导致路由切换到 SymbolController: export { MarketView }中的 quote 方法。

实现 market 模板

MarketView 中载入并编译的模板如下：

```
<div class="panel panel-default fadeInUp animated">
  <div class="panel-body">
    <h2>{{market}}</h2>
    <table class="table table-responsible table-condensed">
      <thead>
        <tr>
          <th>Symbol</th>
          <th>Name</th>
```

```
          <th>Last Sale</th>
          <th>Market Capital</th>
          <th>IPO year</th>
          <th>Sector</th>
          <th>industry</th>
          <th>Quote</th>
        </tr>
      </thead>
      <tbody>
        {{#each items}}
          <tr>
            <td><span class="label label-default">{{Symbol}}</span></td>
            <td>{{{Name}}}</td>
            <td>{{LastSale}}</td>
            <td>{{MarketCap}}</td>
            <td>{{IPOyear}}</td>
            <td>{{Sector}}</td>
            <td>{{industry}}</td>
            <td>
              <button class="btn btn-primary btn-sm getQuote"
             data-symbol="{{Symbol}}">
                <span class="glyphicon glyphicon-stats" aria-
                hidden="true"></span>
                Quote
              </button>
            </td>
          </tr>
        {{/each}}
      </tbody>
    </table>
  </div>
</div>
```

实现 symbol controller

在 app/controllers 目录下新建名为 symbol_controller.ts 的文件，这个文件将包含新的名为 SymbolController 的 controller。这个 controller 的实现与早些时候的 MarketController 非常类似，所以我们不会深究太多的细节。

这个 controller 与之前的那个主要的区别在于，它使用了两个新的 model（QuoteModel 和 ChartModel）和两个新的 view（SymbolView 和 ChartView）：

```
/// <reference path="../../framework/interfaces"/>

import { Controller, AppEvent } from "../../framework/framework";
import { QuoteModel } from "../models/quote_model";
import { ChartModel } from "../models/chart_model";
import { SymbolView } from "../views/symbol_view";
import { ChartView } from "../views/chart_view";

class SymbolController extends Controller implements IController {
  private _quoteModel : IModel;
  private _chartModel : IModel;
  private _symbolView : IView;
  private _chartView : IView;

  constructor(metiator : IMediator) {
    super(metiator);
    this._quoteModel = new QuoteModel(metiator);
    this._chartModel = new ChartModel(metiator);
    this._symbolView = new SymbolView(metiator);
    this._chartView = new ChartView(metiator);
  }

  // 初始化view/model并开始监听controller中的变化
  public initialize() : void {

    // 订阅controller中的变化
    this.subscribeToEvents([
      new AppEvent("app.controller.symbol.quote", null, (e, symbol
      : string) => { this.quote(symbol); })
```

```
    ]);

    // 初始化view和model中的事件监听
    this._quoteModel.initialize();
    this._chartModel.initialize();
    this._symbolView.initialize();
    this._chartView.initialize();
  }

  // 销毁view/model并停止监听controller中的变化
  public dispose() : void {

    // 销毁controller中的事件监听
    this.unsubscribeToEvents();

    // 销毁view/model中的事件监听
    this._symbolView.dispose();
    this._quoteModel.dispose();
    this._chartView.dispose();
    this._chartModel.dispose();
  }
```

值得一提的是，quote 方法将程序控制权转移到 QuoteModel：

```
  public quote(symbol : string) {
    this.triggerEvent(new AppEvent("app.model.quote.change",
    symbol, null));
  }
}
export { SymbolController };
```

实现 quote model

在 app/models 目录下新建一个名为 quote_model.ts 的文件。这是到目前为止我们创建的第三个 model，这意味着你应该对基础设置非常熟悉了，但在这个 model 中有一些非必需的附加代码。首先就是网络服务不再是静态文件：

```
/// <reference path="../../framework/interfaces"/>
```

```
import { Model, AppEvent, ModelSettings } from "../../framework/
framework";

@ModelSettings("http://dev.markitondemand.com/Api/v2/Quote/jsonp")
class QuoteModel extends Model implements IModel {

  constructor(metiator : IMediator) {
    super(metiator);
  }

  // 监听model中的事件
  public initialize() {
    this.subscribeToEvents([
    new AppEvent("app.model.quote.change", null, (e, args) => {
    this.onChange(args); })
    ]);
  }

  // 销毁model中的事件
  public dispose() {
    this.unsubscribeToEvents();
  }
```

第二件事是，onChange 方法在 getAsync 返回的 promise fulfilled 后调用了一个新的函数（formatModel）：

```
private onChange(args) : void {
    // 格式化参数
    var s = { symbol : args };
    this.getAsync("jsonp", s)
        .then((data) => {

          // 格式化数据
          var quote = this.formatModel(data);

          // 将程序控制权转移给market view
          this.triggerEvent(new AppEvent("app.view.symbol.render",
```

```
            quote, null));
        })
        .catch((e) => {
          // 将程序控制权转移到全局错误处理方法
          this.triggerEvent(new AppEvent("app.error", e, null));
        });
    }
```

这个新的方法仅仅是格式化网络服务返回的数据，使之能显示得更加友好。我们将在 promise 的 fulfilled 回调中进行格式化，使用一个分离的方法可以让代码变得更加清晰：

```
    private formatModel (data) {
      data.Change = data.Change.toFixed(2);
      data.ChangePercent = data.ChangePercent.toFixed(2);
      data.Timestamp = new
      Date(data.Timestamp).toLocaleDateString();
      data.MarketCap = (data.MarketCap / 1000000).toFixed(2) + "M.";
      data.ChangePercentYTD = data.ChangePercentYTD.toFixed(2);
      return { quote : data };
    }
  }
  export { QuoteModel };
```

实现 symbol view

在 app/views 目录下新建一个名为 symbol_view 的文件。SymbolView 通过 app.view.symbol.render 事件接收 QuoteModel 格式化后的股票数据：

```
/// <reference path="../../framework/interfaces"/>

import { View, AppEvent,ViewSettings } from "../../framework/
framework";

@ViewSettings("./source/app/templates/symbol.hbs", "#outlet")
class SymbolView extends View implements IView {
  constructor(metiator : IMediator) {
```

```
    super(metiator);
  }
```

这个 view 和 MarketView 非常类似，它使用 initialize 方法订阅一些事件，使用 dispose 方法销毁它们。SymbolView 也可以使用 bindDomEvents 和 unbindDomEvents 方法初始化和销毁用户事件订阅。

然而，SymbolView 和 MarketView 有一个显著的差别。在 rederAsync 方法返回的 promise fulfilled 和用户事件初始化完成之后，程序的执行流通过 app.model.chart.change 事件移交到了 model。此时，股票报价页面已经显示出来了，但是还缺少表格部分：

```
initialize() : void {
  this.subscribeToEvents([
    new AppEvent("app.view.symbol.render", null, (e, model :
    any) => {
      this.renderAsync(model)
          .then((model) => {
            // 设置DOM事件
            this.bindDomEvents(model);

            // 将程序控制权转移到chart View
            this.triggerEvent(new
             AppEvent("app.model.chart.change", model.quote.Symbol,
             null));
          })
          .catch((e) => {
            this.triggerEvent(new AppEvent("app.error", e,
            null));
          });
    }),
  ]);
}

public dispose() : void {
  this.unbindDomEvents();
  this.unsubscribeToEvents();
}
```

```
    // 初始化DOM事件监听
    protected bindDomEvents(model : any) {
      var scope = $(this._container);
      // 设置DOM事件监听
    }

    // 销毁DOM事件监听
    protected unbindDomEvents() {
      var scope = this._container;
      // 释放DOM事件
    }
}
export { SymbolView };
```

实现 chart model

在 app/models 目录下新建一个名为 chart_model.ts 的文件：

```
/// <reference path="../../framework/interfaces"/>

import { Model, AppEvent, ModelSettings } from "../../framework/framework";

@ModelSettings("http://dev.markitondemand.com/Api/v2/InteractiveChart/
jsonp")
class ChartModel extends Model implements IModel {

  constructor(metiator : IMediator) {
    super(metiator);
  }

  // 监听model事件
  public initialize() {
    this.subscribeToEvents([
      new AppEvent("app.model.chart.change", null, (e, args) =>
      { this.onChange(args); })
    ]);
  }
```

```
// dispose model events
public dispose() {
  this.unsubscribeToEvents();
}
```

接下来我们需要格式化请求和响应部分。先将请求的参数进行编码，因为网络服务无法接受 URL 中未编码的请求参数。

onChange 方法使用浏览器的 JSON.stringify 方法把需要的请求参数（一个 JSON 对象）转化成字符串。这个字符串被浏览器的 encodeURIComponent 方法编码，这样它就可以作为 URL 的参数被发送到网络服务。

请求的响应部分使用 formatModel 方法进行格式化：

```
private onChange(args) : void {
  // 格式化参数（更多信息请访问 http://dev.markitondemand.com/）
  var p = {
    Normalized : false,
    NumberOfDays : 365,
    DataPeriod : "Day",
    Elements :[
      { Symbol : args , Type : "price", Params :["ohlc"] }
    ]
  };
  var queryString = "parameters=" +
  encodeURIComponent(JSON.stringify(p));

  this.getAsync("jsonp", queryString)
    .then((data) => {

      // 格式化数据
      var chartData = this.formatModel(args, data);

      // 将程序控制权转移给 market view
      this.triggerEvent(new AppEvent("app.view.chart.render",
      chartData, null));
      })
      .catch((e) => {
```

```
        // 将程序控制权转移到全局错误处理方法
        this.triggerEvent(new AppEvent("app.error", e, null));
    });
}
```

下面这个方法被用来格式化 dev.markitondemand.com 返回的数据，让它可以很简单地被 Highcharts 使用。Highcharts 是一个可以在客户端显示图表的库：

```
private formatModel(symbol, data) {
    // 更多信息参见http://dev.markitondemand.com/
    // 和http://www.highcharts.com/demo/line-time-series
    var chartData = {
      title : symbol,
      series : []
    };

    var series = [
      { name : "open", data :
      data.Elements[0].DataSeries.open.values },
      { name : "close", data :
      data.Elements[0].DataSeries.close.values },
      { name : "high", data :
      data.Elements[0].DataSeries.high.values },
      { name : "low", data :
      data.Elements[0].DataSeries.low.values }
    ];

    for(var i = 0; i < series.length; i++) {
      var serie = {
        name: series[i].name,
        data: []
      }

      for(var j = 0; j < series[i].data.length; j++){
        var val = series[i].data[j];
        var d = new Date(data.Dates[j]).getTime();
        serie.data.push([d, val]);
      }
```

```
        chartData.series.push(serie);
      }
      return chartData;
    }
}
export { ChartModel };
```

实现 chart view

在 app/views 中新建一个名为 chart_view.ts 的文件，这将是我们实现的最后一个 view。这个 view 和前面的那些几乎相同，但有一个显著的差别。图表是被 Highcharts 渲染而不是被 Handlebars 渲染，所以不要在 ViewSettings 装饰器中传入模板 URL：

```
/// <reference path="../../framework/interfaces"/>

import { View, AppEvent,ViewSettings } from "../../framework/
framework";

@ViewSettings(null, "#chart_container")
class ChartView extends View implements IView {

  constructor(metiator : IMediator) {
    super(metiator);
  }
```

ChartView 订阅了 app.view.chart.render 事件。这个事件处理函数在 ChartModel 加载并格式化后被调用，考虑到我们不需要渲染 Handlebars 模板，所以不需要在这里调用 renderAsync 方法（在前面所有的 view 中我们都这么做了），使用一个 renderChart 方法取而代之：

```
initialize() : void {
  this.subscribeToEvents([
    new AppEvent("app.view.chart.render", null, (e, model : any) =>
{
      this.renderChart(model);
```

```
        this.bindDomEvents(model);
      }),
    ]);
  }

  public dispose() : void {
    this.unbindDomEvents();
    this.unsubscribeToEvents();
  }

  // 初始化DOM事件
  protected bindDomEvents(model : any) {
    var scope = $(this._container);
    // 设置DOM事件
  }

  // 销毁DOM事件
  protected unbindDomEvents() {
    var scope = this._container;
    // 释放DOM事件
  }
```

renderChart 方法使用了 Highcharts API（http://api.highcharts.com/highcharts）将 ChartModel 返回的数据转变成一个好看的交互式图表：

```
  private renderChart(model) {
    $(this._container).highcharts({
      chart: {
        zoomType: 'x'
      },
      title: {
        text: model.title
      },
      subtitle: {
        text : 'Click and drag in the plot area to zoom in'
      },
      xAxis: {
```

```
        type: 'datetime'
      },
      yAxis: {
        title: {
          text: 'Price'
        }
      },
      legend: {
        enabled: true
      },
      tooltip: {
        shared: true,
        crosshairs: true
      },
      plotOptions: {
        area: {
        marker: {
          radius: 0
        },
          lineWidth: 0.1,
          threshold: null
        }
      },
      series: model.series
    });
  }
}
export { ChartView };
```

测试应用

可以使用与前几章一样的工具测试这个应用。你知道，我们需要先创建一个这样的 Gulp 任务：

```
gulp.task("run-unit-test", function(cb) {
  karma.start({
    configFile : "karma.conf.js",
    singleRun: true
  }, cb);
});
```

我们使用了 Karma，然后需要使用 karma.conf.js 设置它。这里的 karma.conf.js 与第 7 章中的几乎一样，为了保持书本内容简洁，不再赘述。

还需要一个任务运行端对端测试：

```
gulp.task('run-e2e-test', function() {
  return gulp.src('')
    .pipe(nightwatch({
      configFile: 'nightwatch.json'
    }));
});
```

nightwatch.json 也和第 7 章中的几乎一样，也不在这里列举了。

可以在随书代码中找到 nightwatch.json 和 karma.conf.js 的内容。

准备发布程序

现在，程序已经被实现并进行了测试，可以准备把它发布到生产环境中了。

在这一节中，我们会实现两个 Gulp 任务。第一个用来压缩编译出来的 JavaScript 代码。代码压缩可以提高加载速度和执行效率：

```
gulp.task("compress", function() {
  return gulp.src("bundled/source/bundle.js")
             .pipe(uglify({ preserveComments : false }))
             .pipe(gulp.dest("dist/"))
});
```

第二个任务用来为代码的头部增加版权信息。这个任务使用了 package.json 的一些字段来生成字符串，它包含了版权信息。这些信息随后被加到上一个任务生成的压缩后的

代码的顶部：

```
gulp.task("header", function() {

  var pkg = require("package.json");

  var banner = ["/**",
    " * <%= pkg.name %> v.<%= pkg.version %> - <%= pkg.description
    %>",
    " * Copyright (c) 2015 <%= pkg.author %>",
    " * <%= pkg.license %>",
    " * <%= pkg.homepage %>",
    " */",
    ""].join("\n");

  return gulp.src("dist/bundle.js")
             .pipe(header(banner, { pkg : pkg } ))
             .pipe(gulp.dest("dist/"));
});
```

也可以在以后创建一些其他的 Gulp 任务提升应用的性能。比如，可以新建一个任务生成缓存清单（一个简单的文本文件，列出浏览器在离线时可访问的资源目录）来实现离线缓存。

小结

在本章中，我们创建了一个 MVC 应用，它能让我们获知 NASDAQ 和 NYSE 股票在特定日期的走势。这个单页应用的架构使得组件很容易扩展、复用、维护和测试。

这个应用展示了很多在前几章中介绍的概念。我们创建了一个自动化构建工具，使用了很多函数、类、模块和其他语言特性，还使用了一些异步函数和装饰器。自动化构建工具让我们能够在优化程序性能的同时保证它能正常工作。

这虽并非大型应用，但是足以表现出 TypeScript 是如何帮助开发具有扩展性的复杂大型应用的。

我希望大家能享受阅读此书的过程，并急切地想要了解更多 TypeScript 的特性。

如果想挑战并增强你的 TypeScript 技能，可以尝试以下几点：

- 尝试将前两章中开发的应用测试覆盖率提高到 100%。
- 改良我们的框架，增加一些新的特性，例如 IoC 容器或使用单项数据流。

也可以访问 TodoMVC 网站（`http://todomvc.com/`）来找一些 TypeScript 和流行的 MV*框架结合的例子，例如 Ember.js 或 Backbone.js，了解如何使用成熟的 SPA 框架。