SaaS软件工程：
云计算时代的敏捷开发
第1版

Armando Fox　　David Patterson

2014年3月28日

Copyright 2013 Strawberry Canyon LLC. All rights reserved.

No part of this book or its related materials may be reproduced in any form without the written consent of the copyright holder.

未经出版者书面许可，不得以任何方式复制本书中的任何部分或其相关材料。

本书版本： 1.0

本书的背景图片是西班牙的**塞戈维亚渡槽**。我们选择它作为漂亮而持久的设计的一个例子。渡槽全长大约20英里（32公里），由罗马人在公元1世纪或2世纪建造的。这张照片是使用没有上过灰浆的花岗岩石块建造的一段半英里（0.8公里）长，地上92英尺（28米）高的渡槽。罗马设计师遵循了Marcus Vitruvius Pollio公元前15年写的《**建筑十书**》（De Architectura）里的建筑性原则。它一直保持着当年的风貌，一直到16世纪，国王Ferdinand和女王Isabella才第一次重建了这些拱门。渡槽直到最近还在使用并输送水源。

印刷书籍和电子书都使用了LaTeX，`tex4ht`，以及Ruby脚本，后者在文本中调用了Nokogiri，通过改动XHTML输出和HTTParty来自动保持最新的Pastebin和视频URI的。必要的Makefile，样式文件和大多数的脚本都可以在`http://github.com/armandofox/latex2ebook`下以BSD许可的方式获得。

Arthur Klepchukov设计了所有版本的封面和图片。

译者序

在上一次清华园里绿树成荫的时候，我们拿到了这本书的原稿，开始着手将这部优秀的作品介绍给广大读者。在这段时间里，园子里的银杏叶黄了又落了，在不知不觉间枝头上已擎起饱满的嫩芽，只待东风唤得春回大地的时候重新将绚烂的生命力绽放。而我们的书稿也终于得以付梓印刷，也算是对这一学期以来在文字间的辛苦耕作有了一个交待。

本书封面图中的塞戈维亚大渡槽历经千年风雨，到如今依然屹立不倒，甚至还发挥着它原本的作用。而在更早的战国时期由李冰设计建造在成都的都江堰，至今仍为沃野千里的天府之国带来福祉。软件开发在经历了简陋而效率低下的发展阶段后，进入了软件工程阶段，同样也需要像这些建筑工程一样的漂亮而持久的设计。

尤其是进入21世纪以来，软件即服务和敏捷开发成为了软件工程的两个新的发展方向。然而，对软件工程的从业者来说，想要找到合适的学习资料却似乎并不容易。软件即服务方面的书籍太过繁多，读者要从中分辨并找到适合自己的一本更为不易。而对软件工程以及敏捷开发来说，情况可能更糟糕：我们很难找到令大家满意的书籍，有的太过冗长，有的局限于定性分析，有的偏于学究而没有告诉读者该如何做。如果将范围缩小到中文书籍，情况只能是更加艰难。

本书的原作者结合自己的研究教学和实践经历，将软件即服务和敏捷开发的内容结合到具体的示例中去，最终形成了这本易于学习而又内容丰富的书。书中包含了大量的真实可运行代码、阐述，还有视频帮助大家学习。初次接触到这本书的时候，书中的内容让我耳目一新。结合本人从事软件工程相关工作的经验，本书讲述的内容会给有志于开发SaaS软件的人们带来巨大帮助。我很高兴这么快就有了机会亲自来组织翻译这本书。

我要首先感谢本书的原作者，也是我的博士导师，Armando Fox和David Patterson教授。两位作者不仅在计算机科学研究领域建树颇丰，而且一生都致力于提升计算机科学的教学水平。他们在在本书翻译过程中不断地进行更新，使本书包含了到出版为止最新的内容。本书的英文原版不仅内容丰富深刻，而且语言生动优美。这也使我们在翻译的时候不得不反复忖度，生怕失之偏颇。好在最后的结果也还差强人意。

本书的翻译是集体劳动的成果。本书中文版翻译过程中，得到了许多人的支持与付出，我们感谢所有为本书的编写和翻译作出辛苦努力的人：

我的博士生曹锐创负责了整个翻译团队的组织工作，并对本书翻译结果进行了全面校对。感谢交叉信息研究院的李浒同学、李古月同学、陈天佳同学，他们利用课余时间共同完成了本书的大部分翻译工作。

感谢清华大学交叉信息研究院的宫澄光同学、王康宁同学，以及清华大学软件学院的马洪天同学、张桂菊同学、江梦琪同学、陈欣洁同学、蔡历明同学、卢悦婷同学。他们利用自己的课余时间，协助完成了本书的主要校对工作。

感谢清华大学交叉信息研究院、教务处、在线教育办公室，支持我用本书和本书配套的在线课程进行《云计算与软件工程》课程混合式教学的试点。感谢本门课程的助教贾永正、魏达、刘佳倩。同时也要感谢选修此门课程的同学，他们在课程上使用本书，并协助指出了翻译中的不少错误。祝他们通过这门课程和这本书学到更多有用的东西。

祝愿国内所有的读者都能从本书中发现价值，获得提高，同时祝本书也能成为一个漂亮而持久的设计示例。由于译者的专业技术及语言水平有限，翻译时间紧，加上翻译过程中发生的疏漏，书中难免会有我们没有察觉到的不足甚至是错误之处，在此希望广大读者能够谅解。同时，我们热切希望读者在阅读的过程中，指出书中的谬误，对本书提出宝贵的意见。

大规模开放在线课程（MOOC）

学堂在线是清华大学发起的大规模开放在线教育平台。提供了大量免费的国内外优秀课程。

本书的作者还同步制作了两门大规模开放在线课程（MOOC）。目前这两门课程在全球范围内已经有数十万人选课，数以万计的学生获得了课程合格证书。课程的每一讲与本书的一个章节一一对应。

我们在2013年春季学习尝试了在北京几所高效推广这两门在线课程，秋季学习在清华大学开设了基于这两门在线课程的混合式教学试点，都取得了很好的效果。

清华大学发起的在线教育平台“学堂在线”（http://www.xuetangX.com[1]）将于2014年初引进本书配套在线课程的中文版（英文原版视频+中文字幕）。希望届时通过作者原汁原味的精彩讲解，在线提交并且自动批改的编程作业和小测验，能够真实还原美国顶尖大学的课程体验。同时学堂在线平台将提供的本地中文助教和讨论区的支持，加以本书中文版的配合，相信会使读者获得更好的学习效果。

译者简介

徐葳博士现任清华大学交叉信息研究院助理教授，博士生导师，中组部“青年千人计划”特聘专家。博士毕业于美国加州大学伯克利分校，是本书的两位作者David Patterson和Armando Fox联合指导的博士生。他主要从事超大规模分布式系统以及大数据处理方向的跨领域研究工作，相关研究工作发表SOSP，ICML，ICDM等顶级国际会议。教学中他主要负责计算机科学实验班的“软件工程”（使用本书作为教材）和“操作系统”两门全英文授课的课程。徐葳博士具有丰富的大型分布式系统工程经验和项目管理经验。加入清华大学前曾在Google总部工作，负责系统日志和监控系统的研发工作。他在清华大学和美国宾夕法尼亚大学完成了本科学业，并在加州大学伯克利分校获得硕士和博士学位。

曹锐创现为清华大学交叉信息研究院博士研究生，徐葳博士的学生，目前主要从事云计算与大数据方面的研究。2012年毕业于清华大学计算机科学实验班（“姚班”），并免试获得博士生资格。在校期间多次获得获学业和综合优秀奖励。曾两次两次赴香港中文大学访问交流，并曾在在惠普中国研究院实习半年，从事智能打印方面研究。

对 软件工程即服务 课程和本教材的褒奖

很荣幸看到学生发信息强调产生真正有用的软件的重要性。我也赞赏强调在过程的早期获得结果的看法，没有什么比这更能鼓舞学生的士气和活动的了。

——Frederick P. Brooks, Jr.，图灵奖获得者以及 《人月神话》 作者

和其他任何我看过的毕业生相比，我可能会更倾向于选择这个项目的毕业生。

——Brad Green，工程经理，谷歌公司

C3 Energy的很多软件工程师不断地报告称，这本书和它伴随的网络课程让他们能够迅速地在SaaS开发上达到熟练的程度。我向那些希望开发或改善其SaaS编程技巧的人推荐这本独特的书和这门的独特课程。

——Thomas M. Siebel，首席执行官，C3 Energy以及Siebel Systems（主要的客户关系管理软件公司）的前首席执行官

广泛而深入地覆盖了你在SaaS入门中需要的所有内容。

——Vicente Cuellar，首席执行官，Wave Crafters，Inc。

这本书填补了我在云计算上的知识缺口，它的课程也很容易学习。也许最令人兴奋的部分是编写一个云应用程序，上传并部署到Heroku。

——Peter Englmaier，苏黎世大学，瑞士

一个优秀的学习Ruby、Rails和测试驱动方法的入门。此书完美介绍了现代WEB开发，深入涉猎探索了其基本原理。这是现代Web开发的介绍，它应该是一名新的工程师必备的书籍。

——Stuart Corbishley，Clue Technologies/CloudSeed，南非

一本令人振奋的书，让你在短短数天内就可以逐步构建运行SaaS应用。本书的视频和代码实例部分是无价的。这是一门非常实用的敏捷软件开发方法。你不会知道它，但是你甚至可以在不知道正在做什么的过程中学到软件工程的技术。

——Rakhi Saxena，助理教授，德里大学，印度

作者完成了一个对任何现代高级软件工程课程的开端来说，都非常受欢迎的理论与实践的融合。一方面，它们囊括了关键的软件工程基础，包括开发流程、需求工程、软件测试、软件体系结构、配置管理、实现以及部署。另一方面，它们传达了以Ruby/Rails为中心建立一个“真实世界”的方法，包括其丰富的敏捷生态系统、测试驱动和行为驱动开发工具与技术，并指明了在云端部署运行优质软件的康庄大道。我已经在本科生高级软件工程课程里成功地使用了这本书的Beta版本，它几乎完美地充实了我的课程和项目。

——Ingolf Krueger，教授，加州大学圣地亚哥分校

一本真正好的介绍实用敏捷开发的书。所有你需要的东西都汇聚在这本书里，其中有很多实用的例子。

——Dmitrij Savicev，Sungard Front Arena，瑞士

关于作者

Armando Fox是加州大学伯克利分校计算机科学专业的一名住校教授（Professor-in-Residence），同时也是该校在线教育的教务主任。他在斯坦福大学期间获得了由斯坦福大学学生会、女性工程师协会以及国家工程荣誉协会（Tau Beta Pi）颁发的教学与指导奖。他获选2003年的“科学美国人50强”（Scientific American 50），同时还是NSF CAREER奖和美国国家工程院Gilbreth Lectureship奖的获得者。在此之前他参与设计了英特尔奔腾微处理器并创建了一个公司，成功地将他在伯克利进行的移动计算方面的论文研究商业化，这其中就包括世界上第一个运行在移动设备上的图形Web浏览器（Top Gun Wingman on the Palm Pilot）。他在麻省理工学院和伊利诺伊大学分别获得了电气工程和计算机科学学位，并获得ACM杰出科学家称号。他也是一位经过古典音乐训练的音乐家和自由音乐总监，以及一个移民到旧金山的同时掌握两种语言/文化（古巴裔美国人）的纽约人。

David Patterson是加州大学伯克利分校计算机科学方向的Pardee讲席教授，目前是并行计算实验室（ParLab）负责人。过去他曾担任伯克利计算机部门的主席、CRA的主席以及美国计算机协会（ACM）主席。他最著名的研究项目是精简指令集计算机（RISC）、磁盘冗余阵列（RAID）以及网络工作站（NOW）。这些研究产生了多篇论文、6本书以及超过35项荣誉，包括入选美国工程院和科学院院士，以及硅谷工程名人堂，还有计算机历史博物馆、ACM、IEEE、美国艺术与科学学会（AAAS）、美国科学促进会（AAAS）的会士。他的教学奖项包括卓越教学奖（加州大学伯克利分校）、Karlstrom杰出教育家奖（ACM）、穆里根教育奖章（IEEE），以及本科教学奖（IEEE）。他在加州大学洛杉矶分校获得了所有的学位，学校授予了他杰出工程学术校友奖。他在加州长大，在业余生活中，他和两个成年的儿子共同参加体育赛事，包括每周的足球游戏、年度慈善自行车行以及短跑三项全能运动，还偶尔参加举重比赛。

致谢

Armando Fox将这本书献给他的妻子，也是他最好的朋友Tonia。在他写作的时候，她的支持让这本书与众不同。还有Pogo，这本书的大部分是在它的细心监督下写出来的，而它充满生气的精神也将永远留在我们的家里，留在我们的心中。

David Patterson把这本书献给他的父母和他们所有的后代：
——给我的父亲David，从他身上我继承了创造性、崇尚运动还有为正义而战的勇气；
——给我的母亲Lucie，从她身上我继承了才智、乐观和我的性情；
——给我们的儿子David和Michael，他们是我的朋友，运动伙伴，以及让我成为一个好男人的鼓励；
——给我们的儿媳妇Heather和Zackary，她们是我们的孙子孙女们聪明、有趣且有爱心的母亲；
——给我们的孙子孙女Andrew，Grace和Owyn，他们给了我们延续生命的机会（并且帮助营销这本书）；
——给我的弟弟妹妹Linda、Don和Sue，他们给了我第一次教课的机会；
——给他们的后代，他们让Patterson家族变得更大，并友好快乐相处；
——给我的美丽和善解人意的妻子Linda，她是我最好的朋友，是我生命的挚爱。

简 要 目 录

第1章　SaaS与敏捷开发介绍 0

第I部分　软件即服务

第2章　SaaS应用的架构 34
第3章　SaaS架构：针对Java编程者的Ruby简介 62
第4章　SaaS架构：Rails简介 92
第5章　SaaS框架：Rails高级功能 126
第6章　SaaS的客户端框架：JavaScript简介 154

第II部分　软件开发：敏捷开发与计划文档模式

第7章　需求：行为驱动设计用户故事 200
第8章　软件测试：测试驱动开发 238
第9章　软件维护：使用重构和敏捷方法改进遗留软件 276
第10章　项目管理：Scrum、结对编程、及版本控制系统 306
第11章　SaaS类的设计模式 334
第12章　SaaS非功能性需求：性能、发行、可靠性和实用安全 366

编后记 402

附录 A　使用本书配套软件 410

目　录

第1章　SaaS与敏捷开发介绍 **0**
1.1　简介 2
1.2　SaaS，软件即服务 2
1.3　面向服务的架构（SOA） 4
1.4　云计算 8
1.5　漂亮的代码与遗留代码 10
1.6　软件质量保证：测试 11
1.7　生产力：简练、合成、代码重用和工具 12
1.8　软件开发过程：计划和文档过程 15
1.9　软件开发过程：敏捷宣言 20
1.10　本书导览 23
1.11　不应如此读这本书 25
1.12　谬论和陷阱 27
1.13　结束语：软件工程不只是单纯的编程 27
1.14　深入学习 29
1.15　建议的项目 31

第I部分　软件即服务 **33**

第2章　SaaS应用的架构 **34**
2.1　100000英尺：客户端——服务器架构 36
2.2　50000英尺：通信——HTTP和URI 38
2.3　10000英尺：表示方法——HTML和CSS 41
2.4　5000英尺：3层体系结构和横向扩展 44
2.5　1000 英尺：模型——视图——控制器架构 47
2.6　500英尺：模型的活动记录 48
2.7　500英尺：路由、控制器与REST 51
2.8　500英尺：模板视图 54
2.9　谬论和陷阱 56
2.10　结束语：模式、架构和长寿命的API 57
2.11　深入学习 58
2.12　推荐作业 58

第3章 SaaS架构：针对Java编程者的Ruby简介 62
3.1 概述以及Ruby的三大支柱 64
3.2 一切都是对象 68
3.3 每一个操作都是一个方法调用 69
3.4 类、方法和继承 71
3.5 所有的编程都是元编程 75
3.6 块：迭代器、函数式习语以及闭包 77
3.7 mix-in和鸭子类型 81
3.8 用Yield构造自己的迭代器 82
3.9 谬论和陷阱 85
3.10 结束语：惯用语言使用 86
3.11 深入学习 87
3.12 建议的项目 87

第4章 SaaS架构：Rails简介 92
4.1 Rails基础：从0到CRUD 94
4.2 数据库和迁移 98
4.3 模型：Active Record基础 100
4.4 控制器和视图 104
4.5 调试：当出现错误时 110
4.6 表单提交：New和Create 112
4.7 重定向和Flash 114
4.8 完成CRUD：编辑/更新和销毁 117
4.9 谬论和陷阱 121
4.10 结束语：SOA的设计 122
4.11 深入学习 122
4.12 建议的项目 123

第5章 SaaS框架：Rails高级功能 126
5.1 去除MVC中的重复：Partials、验证和过滤器 128
5.2 单点登录和第三方认证 134
5.3 关联和外键 138
5.4 间接关联 142
5.5 关联的RESTful路由 145
5.6 组合可重用作用域的查询 148
5.7 谬论和陷阱 150
5.8 结束语：语言、生产力和美观 151
5.9 深入学习 151
5.10 建议的项目 152

第6章 SaaS的客户端框架：JavaScript简介 154
6.1 JavaScript：大图景 156
6.2 面向Ruby程序员的客户端JavaScript介绍 158
6.3 函数和构造器 163
6.4 文档对象模型和jQuery 166
6.5 事件与回调 169
6.6 AJAX：异步的JavaScript和XML 174
6.7 测试和调试JavaScript与AJAX 178

6.8 单页面应用和Json API . . . 186
6.9 谬论和陷阱 . . . 190
6.10 结束语：JavaScript的昨天、今天以及明天 . . . 193
6.11 深入学习 . . . 195
6.12 建议的项目 . . . 197

第II部分 软件开发：敏捷开发与计划文档模式 199

第7章 需求：行为驱动设计用户故事 200
7.1 行为驱动设计与用户故事简介 . . . 202
7.2 SMART用户故事 . . . 204
7.3 任务点、速率和Pivotal Tracker . . . 206
7.4 敏捷开发的成本估计 . . . 208
7.5 Cucumber和Capybara介绍 . . . 209
7.6 运行Cucumber和Capybara . . . 211
7.7 低精度用户界面模型和故事板 . . . 213
7.8 优化Rotten Potatoes . . . 216
7.9 显式的和隐式的、命令式的和声明式的情景 . . . 220
7.10 计划文档的视角 . . . 223
7.11 谬论和陷阱 . . . 229
7.12 结束语：BDD的利与弊 . . . 230
7.13 深入学习 . . . 232
7.14 建议的项目 . . . 234

第8章 软件测试：测试驱动开发 238
8.1 背景：RESTful应用程序接口和Ruby工具包 . . . 240
8.2 FIRST、TDD、以及RSpec入门 . . . 241
8.3 TDD周期：红色——绿色——重构 . . . 244
8.4 更多的控制规则和重构 . . . 248
8.5 夹具和工厂 . . . 252
8.6 面向模型的TDD . . . 254
8.7 为Internet建立桩 . . . 258
8.8 覆盖率概念、单元测试与集成测试 . . . 263
8.9 其它测试方法和术语 . . . 265
8.10 计划文档的视角 . . . 267
8.11 谬论和陷阱 . . . 270
8.12 结束语：TDD与传统的调试方法 . . . 271
8.13 深入学习 . . . 272
8.14 建议的项目 . . . 273

第9章 软件维护：使用重构和敏捷方法改进遗留软件 276
9.1 什么是代码“遗留”，敏捷开发对其有什么作用? . . . 278
9.2 探索遗留代码库 . . . 280
9.3 使用鉴定测试来创建基准事实 . . . 283
9.4 注释 . . . 286
9.5 度量、代码味道及SOFA . . . 288
9.6 方法级别的重构：使用嫁接替代依赖 . . . 291

9.7 计划文档的视角 296
9.8 谬论和陷阱 300
9.9 结束语：持续的重构 301
9.10 深入学习 302
9.11 建议的项目 304

第10章 项目管理：Scrum、结对编程、及版本控制系统 306
10.1 团队规模：两个匹萨大小的团队以及Scrum 308
10.2 结对编程 309
10.3 敏捷设计和代码评审 311
10.4 两个匹萨大小团队的版本控制：合并冲突 311
10.5 有效地使用分支 314
10.6 报告和修复代码：五个R 318
10.7 计划文档的视角 320
10.8 谬论和陷阱 327
10.9 结束语：团队、合作、以及四十年的版本控制 328
10.10深入学习 329
10.11建议的项目 331

第11章 SaaS类的设计模式 334
11.1 模式、反模式及SOLID类架构 336
11.2 正好够用的UML 340
11.3 单一责任原则 342
11.4 开闭原则 345
11.5 里氏代换原则 348
11.6 依赖注入原则 350
11.7 迪米特法则 354
11.8 计划文档的视角 359
11.9 谬论和陷阱 360
11.10结束语：框架包含了设计模式 361
11.11深入学习 361
11.12建议的项目 363

第12章 SaaS非功能性需求：性能、发行、可靠性和实用安全 366
12.1 从开发到部署 368
12.2 量化响应性 370
12.3 持续集成和持续部署 372
12.4 发行和特征标记 373
12.5 量化可访问性 376
12.6 监控和查找瓶颈 378
12.7 用缓存来提高渲染和数据库性能 379
12.8 避免滥用数据库查询 383
12.9 安全：在应用程序中保护用户数据 386
12.10计划文档的视角 390
12.11谬论和陷阱 392
12.12结束语：性能、可靠性、安全和有漏洞的抽象层 394
12.13深入学习 395
12.14建议的项目 398

第13章　后记 **402**
13.1 SaaS、SOA、Ruby和Rails的视角 402
13.2 回顾过往 402
13.3 展望未来 406
13.4 最后的话 407
13.5 深入学习 407

附录 A　使用本书配套软件 **410**
A.1 Beta版本的指导 412
A.2 本书配套软件概览 412
A.3 在VirtualBox中使用Bookware的虚拟机 412
A.4 在Amazon的EC2上使用Bookware的虚拟机 414
A.5 编辑代码：编辑器和Unix生存技巧 415
A.6 开始使用Git进行版本控制 415
A.7 开始学习GitHub或ProjectLocker 417
A.8 使用Heroku部署到云端 420
A.9 谬论和陷阱 421
A.10 深入学习 422

前言

> 如果你想造一条船，你不能只是叫人来捡木头，分派工作，然后不停地下命令。
> 相反的，你应该教他们去向往那广阔无尽的海洋。
>
> ——Antoine de Saint-Exupéry，**Citadelle**，1948

欢迎！

2000年以来的两个重大技术进步组成了本书的两个部分。第一部分讲解了**软件即服务（SaaS，Software as a Service）**，SaaS正在给软件工业带来一场革命。SaaS在云端运行单一的软件副本，并满足百万级的用户的不同的需求。与之对应，传统的软件要求客户在自己的计算机上安装这一软件的百万个副本之一。

SaaS的用户和开发人员的激情带来了新的、高效的SaaS软件开发框架。我们在本书中使用 **Ruby on Rails（RoR）** 这一架构。RoR这一架构被广泛认为是开发SaaS的最好工具。当然也有许多用其他语言的SaaS框架，例如Python/Django，JavaScript/Jasmine，以及Java/Enterprise Java Beans。

现在的问题在于，什么样的软件开发方法最适合SaaS。因为SaaS软件只有一个副本，并且运行在一个严格受控的环境中，我们可以很容易地以增量的方式增加功能，因此SaaS软件进化的速度要远快于传统的软件。因此，我们需要一个能够将代码改动作为常态，而不是异常的软件开发过程。

工业界经常抱怨软件工程教育的薄弱。我们同很多顶尖软件公司的代表沟通过，例如亚马逊、eBay、脸谱、谷歌和微软。我们非常震惊地发现，这些公司一致认为，第一要务是让学生了解如何改善缺乏文档的遗留代码。这些公司的其他要求，按照重要性排列，分别为：让测试成为软件中的一等公民；与非技术背景的客户一起工作；团队合作。与非技术背景的客户合作，以及团队合作等社交技能显然是对软件开发者的职业发展是大有帮助的，但是问题是如何在本书中介绍它们。因此，我们需要一个软件工程方法，这个方法能够帮助处理遗留代码，强调测试，能够整合非技术用户，并且能更好地支持团队合作而非单干。

碰巧的是，与SaaS出现几乎相同的时间，一些开发者在2001年提出了**敏捷宣言（The Agile Manifesto）**。敏捷宣言是传统软件工程方法根本上的变革。敏捷的一个基本信条是："响应改变重于遵循计划"，因此它与"计划文档"方法，例如瀑布模型，螺旋模型和RUP模型相比，更加适合快速演化的SaaS软件开发。敏捷的另一个基本信条是："与客户合作重于合约的谈判"。这条的直接结果是每周与非技术背景的客户交流。敏捷开发的两个基础是**行为驱动开发（behavior-driven development）**和**测试驱动开发（test-driven development）**。这两条意味着，测试是在代码**之前**完

成的，因此在敏捷开发中测试真正成为了软件中的一等公民。敏捷开发中的 **结对编程** 和 **Scrum** 等理念，强调了与他人合作的重要性。就像我们将要看到的，敏捷开发的技术对于演进遗留代码也有重要的作用。

因此，本书后半部分通过开发并部署一个用RoR的SaaS应用来讲述敏捷开发。另外，每章也介绍了计划文档方法的不同视角，例如需求、测试、管理和维护。这种对比让读者可以自己决定哪种方法更适合他们的SaaS或者是非SaaS的应用。

从授课教师的角度看，这种对比的方式让这本书适宜作为软件工程课程的教材。例如，我们确保了这些材料能够满足2013年ACM/IEEE软件工程课程大纲的标准。事实上，章节后的习题中大约有45个直接来自大纲标准中的学习目标（我们作了一个特殊的页边标志）。用另一种指标来看，大约40%的大纲要求可以直接从习题中获得，另外40%直接对应了章节中的内容。两者合计远超过了大纲中要求的，包括45%的知识点的要求。在配套的教师指南中，我们更深入地探讨了这一问题

很多教师（包括我们自己）的经验表明，学生很喜欢在项目中学习和使用敏捷开发。敏捷开发是基于迭代的、计划周期很短，因此对于日程繁忙的本科生和一个快节奏的课程来说，是绝好的搭配。繁忙的学生会很自然地拖延项目的进展，而只是在期末快要交项目的时候连续熬几夜把项目对付掉。敏捷开发不仅能够击败学生这种对付课程项目的策略（因为我们在每次迭代后都会对学生进行评估），而且我们的经验表明学生会采取一种更负责任的态度，能够做出实在的进步。

学生能够遵照敏捷开发方法的一个重要原因，是本书中介绍的RoR工具使得敏捷开发变得更容易。应用这一方法能够真正让学生更容易地做好他们的项目。我们的学生学到的敏捷开发方法也能在需要时应用在非敏捷的项目中。

我们甚至展示了如何将敏捷方法用在非敏捷方法开发的一流代码中，并且说明了敏捷方法不一定只对写新代码有效。

为什么我们要写一本新书?

作者如果对一些已有的书很满意，就不会去写一本新书。我们对一些已有书籍的不满之处，各有不同。

对于第一部分来说，问题并不是关于SaaS的好书太少，而是因为太多了！图1和图2列出了我们参考的50本书中的24本。仅仅这24本书就有超过10000页！这么多的书可能会吓走初学者。因此，我们写作这本新书的原因之一就是将SaaS的相关内容，用一个相对较短的篇幅和相对便宜的价格，给出一个一致明了的介绍和综述。本书测试版的一个评论者说，这本书的第一部分没有新意。当然，如果您愿意花时间和金钱来阅读那么多其他的书，我们认为这个评论是合理的。

对于第二部分，目前只有少量软件工程的教材的选择，并且没有既简短又便宜的选择！我们看过的SaaS的书的书评都是很好的——在亚马逊上的书评一般都是4星以上——但是软件工程教材的书评可就不是这样了。两本最常用的软件工程教材的书评都2~3星之间，并且有许多很差的评语。

原因之一在于这些软件工程教材大都是对那些文献的冗长而定性的综述——在每个内容上仅仅列出了基于研究论文的一些方法的选择——但是没有告诉读者任何具体的方法如何在其中进行挑选。另一个原因可能是因为这些教材的初版创作于SaaS和敏捷开发出现之前很久，导致现在很难将这些截然不同的新理念融入到过去的材料中去。

这种情况似曾相识，本书作者之一在25年前对于计算机体系结构的教材有着同样的感觉：这些书都是很长，都是对于当时研究论文定性的调研和综述，并且没有告诉读者任何关于如何选择这些实现方法的信息。更糟糕的是，那时计算机体系结构领域已经有了巨大的、并且有争议的变革，这些变革并没有被包含在这些教材

图 1: 图中12本图书包括了超过5000页的内容，本书的作者在写作本书的过程中阅读了超过50本图书，这些书中的大部分都在适当的章节中“深入学习”部分列出过

图 2: 另外12本包括了超过5000页内容的图书，这些书中的大部分都在适当的章节中“深入学习”部分列出过

中。这种对当时教材的不满，导致了作者之一和一个朋友共同写了一本全新的计算机体系结构的教材。

重复历史的成功经验，本书的第二部分与传统的软件工程教材截然不同。我们把敏捷过程当做一等公民来介绍，并且包含了具体的、动手实践的代码和工具的实例。这些实例可以让读者遵循敏捷过程，更容易地实现满足客户需求的产品。如上面提到的，第二部分的每章中也讨论了计划文档过程的观点，以帮助读者更好地认识敏捷开发的优劣，并认识到什么时候应该选择哪种过程。

我们在每部分的目标是，将一组广泛的内容用 **单一的陈述** 表达出来，并且给出具体的例子来帮助读者理解。我们可以想象有已经熟悉敏捷开发的读者略过第二部分而只读第一部分来学习SaaS，或者反之。如果你刚开始接触这些内容——或者你学习的是SaaS和敏捷开发之前的内容——那么你可以一箭双雕地学到SaaS和敏捷开发这两个新的、令人振奋的理念。这一极有针对性的策略使本书能够用传统图书的一般章节和页码，以及四分之一的价格，讲解SaaS和敏捷开发两个内容。

大规模开放式在线课程（MOOC）

我们决定开始写书之后，应邀在加州大学伯克利分校讲授的课程的第一部分做成了一个大规模开放式在线课程（MOOC）。这一课程目前可以在saas-class.org[2]看到。MOOC的开发影响了这本书的写作，反之亦然。MOOC中的视频片段几乎与本书的章节一一对应。事实上，正像MOOC倡导的碎片化教学那样，本书中的每一节后边都包括了两道很短的“自测”题，并包含了答案。

我们在2012年2月开始的MOOC中，多达25000学生观看了视频，其中10000多学生尝试提交了编程作业，3500个心理承受能力极强的学生最终完成了整个课程。由于这一MOOC很受欢迎，我们在EdX平台上再次开设了这门课，又有几千名学生参加（CS169.1和CS169.2两门课分别对应伯克利课程的五周）。2012年有超过10000个学生取得了课程证书。因此，本书获得了比我们预期多得多的测试读者。

因此，你应该考虑注册这一免费的在线课程。你可以在saas-class.org[3] 了解更多关于这门课程的内容。

MOOC中的作业和考试都是和本书对应的。如果只是想自己测试一下你是否学懂了本书的内容，你可以在任何时间注册MOOC并提交作业和考试。这些作业和考试将会像正式参加MOOC课程的学生一样获得自动评分。如果你希望获得一个证书，你必须在特定的开课时间内注册并按时完成课程要求，且考试和作业成绩合格。然而，如果你不想要证书，你可以在任何时候观看视频或者进行作业和考试。

为什么用了那么多引述?

我们认为引述可以让本书读起来更有趣，而且引述是老人将他们的智慧传给下一代的方式，并且能够为软件工程设置出好的文化上的标准。我们也希望读者能了解一些这一领域的历史。这就是为什么我们在每章的开头和章节中，引用了很多图灵奖获得者的妙语。

勘误和辅助内容

从作者的角度看，电子书的好处是，我们可以在读者指出书中错误后，可以随时更新电子版本。我们会收集读者的勘误表，并且每年更新几次电子版图书。

如果你在读一个纸版的书，请按照传统的方式，在saasbook.info[4]上查找纠错表。

教师参考材料

网站saas-class.org[5]包含了本书的更多相关资源，包括MOOC的小规模私有版本、可以自动批改的作业题、课堂幻灯片、习题集、教师手册以及更多内容。

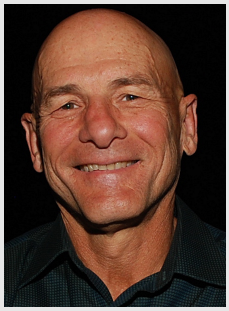
Armando Fox

本书的历史

本书是加州大学伯克利分校的一个研究项目[6]的副产品。在那个研究项目中，我们开发了一个能让下一代互联网服务开发变得更容易的工具。我们认为，年轻人更有可能开发出有意思的互联网服务。因此我们在2007年开设了一门关于SaaS的本科生课程，并在课上使用了敏捷方法。这门课的范围和流行程度每年都在增长，并且我们使用了很多新的Rails提供的工具。在2007～2012年间，这门课的注册人数以摩尔定律的速度增加：35，50，75，115，直到165人。

一个同事建议说，这些内容可以用于改进伯克利已经讲了无数年的软件工程课程，因此我们中的一个（Fox）就这样做了。我们获得了非常好的效果，因此另一个作者（Patterson）建议写一本教材来使其他大学也能从这一课程中受益。

David Patterson



我们坚信两点：电子书将会蓬勃发展，还有避免传统出版商的高额费用和漫长的出版时间的可能性。所以，在2011年3月，我们开始一起协作这本书。使这些内容更为广泛地被人们所接触，以及重新思考电子书应该是什么样子，一样令我们感到非常激动。因为电子书最先是纸质书的PDF文件。

我们与很多人讨论了本书的内容。我们参加了SIGCSE（计算机教育年会），软件工程教育和培训会议，以及计算机联合研究会议等会议，并且在会议上向同行们进行了调研。

基于教育家和工业界人士的观点，我们写出了能够阐明这些问题的提纲，并与当年6月开始协作。由于Fox在这些问题上的更深入的了解，我们最初计划由Fox写大约三分之二的章节，Patterson写其他章节。我们在全书的组织结构上紧密合作，并且互相检查了对方的章节。最终我们都写了对方章节中的一部分，因此最终成书中章节的作者比预想得更为混杂。Fox撰写了以下章节：2、3、4、5、6、8、9、11、12、附录A，以及10.4～10.6节。Patterson撰写了以下几章：1、7、10、序言、结语、教师手册，以及以下节中的计划文档视角部分：7.10、8.10、9.7、10.7、11.8、12.10。Fox编写了LaTeX 的排版流水线，使我们能够生成多种不同格式的电子和纸版图书。这一排版流水线的大部分目前已经开放在github.com/armandofox/latex2ebook[7]，可供其他需要的作者使用。

我们在2012年春季学期为115个伯克利学生和上千名MOOC学生提供了本书的Alpha版本。写作被教学任务延误了一些，在2012年6月，我们获得了学生的反馈后，我们继续加紧写作。Beta版图书在2012年秋季学期和2013年春季学期被其他大学采用。基于Beta版读者和对于2013年版ACM/IEEE计算机协会的软件工程课程大纲的学习，我们在2013年2月推出了Beta版第二版。我们将在2013年秋季学习在伯克利和MOOC学生中再次进行一次Beta测试，并且推出最终经过严格试用的版本。

请到这一网址Errata web page[8] 查看勘误表，并给出您对本书的建议。

我们提前为您在这一版本书中可能发现的错误致歉，并且希望听到你对如何能进一步改进本书提出的意见和建议。

Armando Fox和David Patterson
2013年5月
加州伯克利

免责声明

在本书中，我们使用了网站、工具、产品或者商标名称来作为真实的案例。除非特别说明，作者和这些网站、工具、产品并没有正式的关系，这些例子只是用于介绍的目的，而非商业上的认可。任何商标都属于其所有者，这里的使用只是作为介绍的目的。在任何可能的情况下，我们都专注于免费的和开源的软件和服务，使得学生可以动手实践这些实例，而无需花费更多的钱。

作者的观点是他们自己的，而不是他们的雇主的。

鸣谢

我们感谢我们在工业界的同事们对于本书及我们的课程提出的建议和反馈，特别是以下做出了杰出贡献的个人，以公司的字母顺序排列：Peter Vosshall，Amazon Web Services；Tony Ng，eBay；Tracy Bialik，Brad Green和Russ Rufer，Google Inc；Peter Van Hardenberg，Heroku；Jim Larus，Microsoft Research；Brian Cunnie，Edward Hieatt，Matthew Kocher，Jacob Maine，Ken Mayer和Rob Mee，Pivotal Labs；Jason Huggins，SauceLabs；还有Raffi Krikorian，Twitter。

我们感谢我们学术界的同仁，感谢他们对本书f的方法和思想的反馈意见，特别是：Fred Brooks，北卡罗来纳大学教堂山分校；Marti Hearst和Paul Hilfinger，加州大学伯克利分校；Timothy Lethbridge，渥太华大学；John Ousterhout，斯坦福大学；还有Mary Shaw，卡内基梅隆大学。

我们由衷感谢以下审阅了特定章节的专家们：Danny Burkes，Pivotal Labs；Timothy Chou，斯坦福大学；Daniel Jackson，麻省理工学院；Jacob Maine，Pivotal Labs；John Ousterhout，斯坦福大学；还有Ellen Spertus，密尔斯学院。

感谢悉尼大学的Alan Fekete及时指出了我们应该参考2013年ACM/IEEE计算机学会软件工程教学大纲，使我们能在本书中包含这些内容。

我们特别感谢在他们的课堂中试用本书的测试者们：Daniel Jackson，麻省理工学院；Richard Ilson，北卡罗来纳大学教堂山分校；Samuel Joseph，夏威夷太平洋大学；Almudena Konrad，密尔斯学院；Ingolf Krueger，加州大学圣迭戈分校；Kristen Walcott-Justice，科罗拉多大学科罗拉多斯普林斯校区；还有Rose Williams，宾汉姆顿大学。

本书配套软件的一部分是一组优秀的支持SaaS的第三方网站。他们帮助我们找到了合适的产品和服务，并且将这些产品和服务免费提供给学生在课堂上使用，他们还提供了有益的关于如何在教学中使用这些工具的建议，我们感谢：John Dunham，SauceLabs；Maggie Johnson和Arjun Satyapal，Google Inc；Dana Le，Salesforce；James Lindenbaum，Heroku；Kami Lott和Chris Wanstrath，GitHub；Rob Mee，Pivotal Labs；Ann Merrihew，Kurt Messersmith，Marvin Theimer，Jinesh Varia和Matt Wood，Amazon Web Services；还有Juan Vargas和Jennifer Perret，Microsoft。

我们感谢我们的研究生助教Kristal Curtis和Shoaib Kamil 帮助我们在变革校内教学中做的努力，最终促成了本书的创作。我们感谢研究生助教Michael Driscoll和Richard Xia帮我们设置了可以扩展的自动评分系统，使得上千的学生可以在线提交作业。最后但是很重要的，我们感谢我们本科生实验室助教，他们协助我们完成了课程中的4次迭代更新：Alex Bain、Aaron Beitch、Allen Chen、James Eady、David Eliahu、Max Feldman、Amber Feng、Karl He、Arthur

Klepchukov、Jonathan Ko、Brandon Liu、Robert Marks、Jimmy Nguyen、Sunil Pedapudi、Omer Spillinger、Hubert Wong、Tim Yung、和Richard Zhao。

我们也感谢Andrew Patterson，Grace Patterson和Owyn Patterson 帮助我们推广本书，以及他们的经理Heather Patterson、Michael Patterson、David Patterson以及Zackary Patterson。

最后，我们感谢甘做小白鼠的上百名伯克利学生和上万名MOOC学生！

1 SaaS与敏捷开发介绍

当我有一次往返于EDSAC机房和楼梯转角处的纸带打孔机时，我强烈地意识到，我后半生的大部分将要花在寻找我自己程序的错误上。

——Maurice Wilkes，计算机先驱回忆录，1985

1.1 简介 2
1.2 SaaS，软件即服务 2
1.3 面向服务的架构（SOA） 4
1.4 云计算 8
1.5 漂亮的代码与遗留代码 10
1.6 软件质量保证：测试 11
1.7 生产力：简练、合成、代码重用和工具 12
1.8 软件开发过程：计划和文档过程 15
1.9 软件开发过程：敏捷宣言 20
1.10 本书导览 23
1.11 不应如此读这本书 25
1.12 谬论和陷阱 27
1.13 结束语：软件工程不只是单纯的编程 27
1.14 深入学习 29
1.15 建议的项目 31

Sir Maurice Wilkes（1913——2010）他在1949年设计和制造EDSAC，它是首次实现存储程序的计算机，这为他带来了生平的最高荣誉——图灵奖（1967）。图灵奖[9]是计算机领域中的最高奖项，计算机械协会（ACM）自1966年以来每年授予计算机界杰出的贡献者，以计算机之父Alan Turing的名字命名，被称作计算机界的“诺贝尔奖”。

（本书在侧边栏会补充一些对于本文内容上的有意思的想法，或者简短的计算机先驱的简历，希望你们会乐在其中）。

重要概念

每章开始用一页内容来概括这一章的重要概念。本章主要介绍了以下几个概念，

- **软件即服务** （ **SaaS**，**Software as a Service** ）。厂商将软件统一部署在自己的服务器上，并可以供数以万计的用户访问使用，为用户和开发商提供了非常大的好处。用户从服务商获得软件单一副本的模式，以及SaaS产品的竞争环境，导致Saas软件服务与成品包装的软件相比，拥有更快的 **软件演化** 速度。
- **面向服务的架构** （ **SOA** ）。用相互协作的组件来构建应用，从而更容易的让他们搭建新的系统。
- **云计算** 通过可能多达100，000台服务器的 **仓库规模的计算机** 为SaaS提供可靠且可扩展的计算服务以及存储服务。规模经济效应使云计算服务可以作为一种公共设施，SaaS供应商只需要支付 **真实** 使用的资源而不是 **可能** 需要的资源。
- **软件品质** 是衡量给客户和开发人员带来的商业价值的标准。软件的 **品质保证** （ **QA** ）来自于多层次的测试： **单元** 、 **模块** 、 **集成** 、 **系统** 和 **验收** 。
- **简明清晰** 、 **综合** 、 **复用** 和 **工具自动化** 是提高 **软件生产力** 的四种主要途径。根据满足以上要求的Ruby on Rails来编写框架，可以让SaaS程序员的工作效率得到进一步提高。
- **无重复（DRY，Don't Repeat Yourself）** 敬告大家不要重复代码来达到代码再利用的目的。一个规则实现一次，这是面向对象编程中的基本原则，旨在软件开发中，减少重复的信息。
- **计划和文档** 软件开发过程或 **生命周期** 依赖于预先的规划，广泛详尽的文档与精心的管理使得软件开发前景更加清晰。著名的案例像 **Waterfall model（瀑布模型）** 、 **螺旋模型（Spiral model）** 还有 **统一软件过程** （ **RUP** ）这样的流程。
- 与此相反， **敏捷（Agile）软件开发过程** 是需要大量客户持续的回馈，并以此进行 **迭代** ，然后在原型上不断进行增量开发的过程，每一次这样的流程迭代需要一至两周时间。

由于改变是敏捷开发的常态，因此敏捷过程对SaaS来说是个很好地过程，因此敏捷过程也是本书关注的重点。

1.1 简介

> 对我而言，编程不仅仅是一门实用的艺术，更是一种对于基础知识的伟大的实践。
>
> ——Grace Murray Hopper

Grace Murray Hopper（1906——1992）是世界上最早的程序员之一，她开发了世界上第一个编译器并被称为"伟大的Grace"，她当上了美国海军少将，并且在1997年甚至有一艘军舰以她的名字命名：the USS Hopper。

我们非常荣幸向大家介绍软件在开发中的巨大进展。这一段不是单单向大家介绍软件开发的几个重要案例，成功的软件项目可以创造出一个让数以万计的用户每天离不开的服务，像 **亚马逊** 、 **facebook** 和 **谷歌** 这样响当当的公司。谷歌现在都成为了一个动词，有事没事"google"一下成为了搜索的代名词。每个人都与这样的服务器紧密相关，这让我们感到无比的自豪。

别笑得太早，软件项目也可能带来灾难性的失败并让你背上永久的骂名。资深程序员都听说过 **Ariane5号火箭爆炸案** 、 **Therac-25** 致命核辐射过量事故、 **火星气候探测者号** 失去联络以及FBI **虚拟宗卷** 项目耗资巨大到最后却虎头蛇尾。没有软件工程师希望在自己的简历上有这样的项目。

这本书 **不仅仅** 是一本传统意义上的告诉读者在软件开发的每个阶段该干什么和不该干什么的书。我们的宗旨是教大家怎样把概念具体化以及示范如何设计并实施和部署在云端的应用程序。与这本书相关的虚拟机映像需要读者安装相关软件。在仔细研读本书之外，你还可以在后面的章节中的27段视频中学习到更多。你也可以在我们的官网中找到电子版的相关教材。这本书与免费的 **大规模开放式在线课程** （**MOOC**）相配套，网址为https://www.edx.org/[11]。CS169.1x 与CS169.2x提供8～10分钟的视频片段与这本书的每一章节一一对应（包括这节内容）。这些MOOC里面的编程作业以及小测验，在你答完之后会及时给你反馈，并且你可以在我们的网上论坛与读友进行互动。[1]。

Ariane5号火箭501。 1996年6月4日，一枚火箭升空37秒后，由于将一个浮点数转换成了一个短整数，结果导致制导系统发生溢出，结果十分惨烈[10]。这个异常在比较慢的Ariane4号火箭上根本不会发生，而只是发生在较快的Ariane5号上。它导致了价值3.7亿美元的损失。所以说，仅仅对成功的组件进行重用而非进行彻底的系统测试，这个代价是非常昂贵的。

最后，我想隆重介绍两个主要创新技术，它们彻底改变了软件在过去的十年中的发展。这两项创新技术也是我写这本书的灵感，那么就让我们开始我们的旅程，从软件分配上的一个重大突破开始介绍。

1.2 SaaS，软件即服务

软件即服务（SaaS） 通过互联网提供软件和数据服务，通常是由一个精简的程序运行的，例如我们可以直接在本地客户端的浏览器上运行，而不需要非常麻烦的在自己的计算机上安装程序然后再双击运行。再例如，我们日常生活都会浏览网页，社交网络和看视频。软件即服务核心优势被客户与软件开发者广泛吹捧：

(1) 由于客户不需要安装应用程序，他们不必担心自己的计算机硬件不够强大，也不需要考虑其操作系统是否安装正确。

(2) 自己计算机的数据是与服务器同步的。有了服务器的备份，用户不需再担心当本地硬件故障或者整个设备丢失而丢失数据的问题了。

(3) 当一组用户对于相同的数据需要进行交互的时候，SaaS生来就是为这种情况量身定做的。

(4) 当数据非常巨大或者需要经常更新，也许集中管理这些数据，并提供远程访问支持，是更合理的做法。

[1]译者注：本书配套的中文版课程即将由清华大学的在线教育平台学堂在线引进，请关注http://www.xuetangX.com[12]

(5) 服务器上的软件只有一份单一副本在统一、严格控制的硬件和由开发者选择的操作系统环境上运行。这些特性避免了分散的计算机和操作系统与操作系统之间的兼容性造成的麻烦。此外，开发人员可以给小部分用户测试新版本的应用程序而不会影响到绝大多数的用户（我们将在第2章详细介绍，如果SaaS的客户端在浏览器上运行，其兼容性依然极具挑战）。

SaaS公司可以不断推出新的功能以确保，他们的客户不会因为其他公司有了更好地服务而抛弃他们。

(6) 正因为只有开发商的服务器上有软件，只要不破坏外部应用程序接口（API）他们就可以在自己的服务器上对软件以及硬件进行更新。因此，开发人员不需要不断打扰用户向用户申请升级应用程序的许可。

SaaS: 创新还是走向死亡? 为了消除你对不断重复改进一个已经完善软件工程完全是多此一举的偏见，我举个反例：比如曾今热门的搜索引擎Alta Vista还有曾今流行的社交网站MySpace如今都已经日薄西山了。

这些核心优势向客户与开发商说明了为什么SaaS能迅速成长，以及为什么越来越多的传统意义上的软件产品也转而提供SaaS版本。比如微软Office 365，它把我们正在使用的Word、Excel和PowerPiont转化成一个远程服务，通过网上支付使用，而不像以前那样需要预先购买软件并安装在本地计算机上。还有类似的Turbo Tax在线，也提供了与本地软件相同的功能。

SaaS编程框架	编程语言
Active Server Pages (ASP.NET)	Common Language Runtime (CLR)
Django	Python
Enterprise Java Beans (EJB)	Java
JavaServer Pages (JSP)	Java
Rails	Ruby
Sinatra	Ruby
Spring	Java
Zend	PHP

图 1.1: SaaS的编程框架以及编程语言

不出所料，SaaS的普及程度将非同凡响。上图向我们说明了SaaS所用的编程框架。在这本书中，我们将使用Ruby on Rails（简称为“Rails”），与其他语言基本编程思想是相同的。Rails的优势在于它与敏捷软件开发流程相得益彰，因为它已经接受了敏捷开发的生命周期。

Ruby是一种典型的现代脚本语言，包括自动内存管理和动态类型以及以往的编程语言的优势，Ruby像Perl一样，更先进的一点是它支持多种编程范式，例如面向对象和函数式编程。

Ruby有许多有用的功能，比如通过代码再利用像 **mix-ins** 提升编程效率，通过收集类似的行为可以很容易把他们分配到不同的类中。再例如 **元编程**，可以让Ruby代码在运行时生成代码。

Ruby通过 **块** 和 **yield** 语句实现对 **闭包（Closure）** 的支持能够使复用得到增强。第3章是针对那些已经足够了解Java的读者的Ruby简介，第4章介绍了Rails框架。

在我们看来，除了敏捷开发和SaaS方面Rails是一种很好的技术外，Ruby和Rails的使用范围也很广。比如，Ruby已经进入最广泛使用的10种编程语言。Twitter就是一个与Rails相关的SaaS应用。Twitter从2006年起就成为了一个Rails应用，日均发帖数从2007年的20000条增长到了2011年的200000000条。在这段时间里，Twitter的很多部分都被其他的框架替代了。

如果你对Ruby或Rails还不熟悉，我们提供了一个锻炼软件工程技能的重要机会：在工作中使用正确的工具，哪怕这些新的工具或者新的语言对你来说完全是陌生的。事实上，Rails社区非常吸引人的一点就是人们会经常性地编写新的工具，把原来需要手写完成的任务变成自动的并且以此提升效率。

小结：
软件即服务（SaaS） 对于客户和供应商都十分有吸引力，因为通用的客户端（Web浏览器）使得它更容易为客户提供服务和软件应用。而软件的单一副本安装在一个集中的网站使得它更容易提供完善的售后服务。考虑到SaaS频繁升级的能力与需求，因此敏捷开发有许多框架支持SaaS。这本书使用Ruby on Rails框架。

自我检查1.2.1. *Google SaaS*应用例如搜索、地图查找、新闻、*Gmail*、日历、*YouTube*和*Google Docs*——中的哪一个，能够和本章所给的*6*个与*SaaS*相关的特征之一最好地匹配。这些特征重新列举如下。

◇ 当然你可以不同意以下某些映射关系，但是下面为我们的理解（注意！我们其实把一些应用程序放在多个类别中）。

(1) 无须安装：Google Docs。

(2) 不允许数据丢失：Gmail、日历。

(3) 用户之间进行合作：Google Docs。

(4) 需要大型/变化的数据集：搜索、地图搜索、新闻还有YouTube。

(5) 软件在单一环境中运行：搜索。

(6) 在应用程序升级时文件不进行改动：Google Docs。

自我检查1.2.2. 对或错：如果你正在使用敏捷开发过程来开发*SaaS*程序，你可以用*Python*加上*Django*框架或者其他任何基于微软*CLR/ASP.NET*编程语言来代替*Ruby*和*Rails*。

◇ 正确。敏捷与SaaS的编程框架包括了Django和ASP.NET等。

1.3 面向服务的架构（SOA）

> SOA曾长期受缺乏前景与大方向的折磨。SOA本应已经死了，不是因为SOA缺乏内容或者潜力，而是它看起来有着永不休止的错误信息和混乱。
>
> ——Thomas Erl，**关于SOA宣言**，2010

SOA面临的困惑 在写这本书的时期，维基百科对于 **面向服务架构**、**面向服务的体系结构基础** 还有 **面向服务的结构类型** 专门编写了独立词条。词条编写者甚至在大小写跟连字符上都在争论不休。

SaaS更像一种普通软件的特殊情况，它的特殊在于所有设计都是为了服务而存在的。可惜的是，SOA曾经是一个界定模糊，被使用过度与炒作的模型。有一些人甚至认为它只是一个空头支票什么都没兑现，类似于 **模块化编程（Modular programming）**。SOA是一种互操作服务的应用程序组件，并且可以独立使用或者与其他应用程序重新组合。与SOA相对的是“筒仓式软件”（software silo），几乎没有API可以访问组件的内部。

如果你错误地估计了客户需求，在SOA环境下所花费的成本远比在“Silo软件”环境下花费的要小得多，那么SOA会修复错误并尝试别的方法，产生一个相似但不相同的变种软件从而使某一部分客户满意。

例如，亚马逊在1995年开始在他的零售网上使用筒仓式软件。根据前亚马逊员工Steve Yegg[13]的描述，在2002年，亚马逊的创始人兼首席执行官强制了改变公司软件架构，这个改变就是我们今天所说的SOA。他向他的所有员工发出一封邮件，内容如下：

(1) **今后所有的团队都必须公开他们的数据并打开功能服务接口。**

(2) **团队之间必须通过这些接口来进行沟通。**

(3) **将不再有其他形式的进程间通信：没有直接连接，团队与团队之间不再会有直接读取数据存储，不再使用共享内存模型，没有任何形式的后门。唯一通信方式是直接通过网络上服务接口调用。**

(4) **不论他们用什么样的技术，HTTP、Corba、Pub/Sub、自定义协议——都不重要。Jeff Bezos[亚马逊首席执行官]一点也不关心。**

(5) **所有的服务接口必须从底层设计来支持外部化，也就是说，团队必须在暴露在外界的接口上规划和设计自己的内容。没有例外。**

(6) **不照我说的去做的员工立即离职。**

(7) **感谢！祝好！**

在2007年，在 **Facebook平台** 上线三年后，Facebook公司发生了类似的软件革命。Facebook平台依托SOA架构，它允许第三方开发者创建应用程序并允许别人修改Facebook的核心功能。比如人们喜欢什么，他们的朋友是谁，是谁标记了他们的照片等等。纽约时报是早期的Facebook平台开发者之一。Facebook用户在2007年5月24日在线阅读纽约时报时突然发现，他们能看到朋友们正在阅读哪些文章以及他们喜欢哪些文章。反例是另一个社交网站Google+使用了筒仓式软件架构，2011年6月28日之后的三个月Google+都没有推出任何API，直到后来才有一个重量级的API：追踪某个用户的所有更新。

为了使这些概念具体化，我们先后以Silo和SOA实现一个书店服务。两者都会包含相同的三个子系统：评论、用户信息和购买书籍。

图1.2向我们展现了筒仓式版本。筒仓式可在内部子系统与子系统之间共享数据。例如，在“评论”中可以访问用户个人资料信息这样的“用户”子系统。然而，所有的子系统属于一个单一的外部API（“书店”）。

图1.3为一个书店服务的SOA版本，所有的子系统是独立分开的。尽管所的子系统都在书店数据中心的“边界”内，它们好像是以独立的数据中心的形式相互交流。如果负责评论的子系统希望得到用户的信息，那么它不能直接访问用户相关的数据库。它需要去跟用户 **服务系统** 申请通过用户服务系统提供的API访问，类似的这种限制适用于购买。

“书店软件”只是众多的软件的一个特定组合，其他软件仅需要重组“书店软件”就能得到。比如，“我最喜欢的书”这一软件可能会结合一个社交网络的用户服务和评论服务。所以你在看到了网友对这本书的评价之后再决定你是否阅读它（参见图1.2）。

SOA的关键区别是没有任何服务有权利去寻找/访问其他服务的数据，而只能通过外部的API进行数据请求，如果申请到的API没有权限访问你想要的数据就会很糟

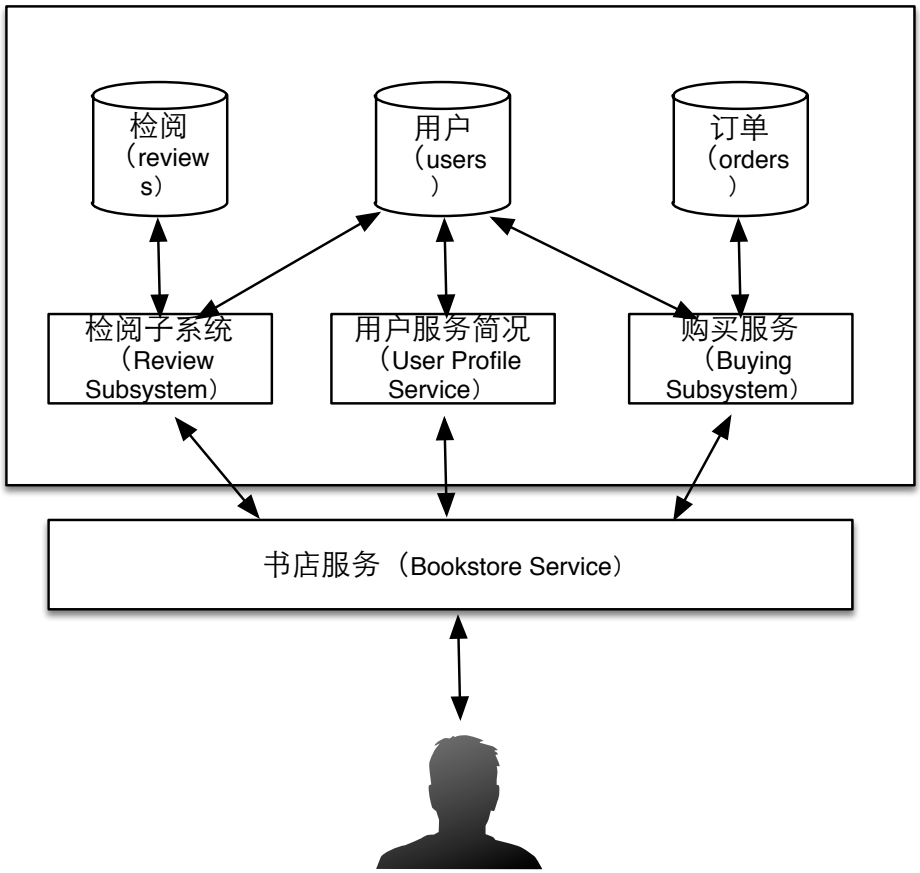


图 1.2: 图示为一个虚拟的书店服务的简仓式版本，每一个API都有各自的子系统

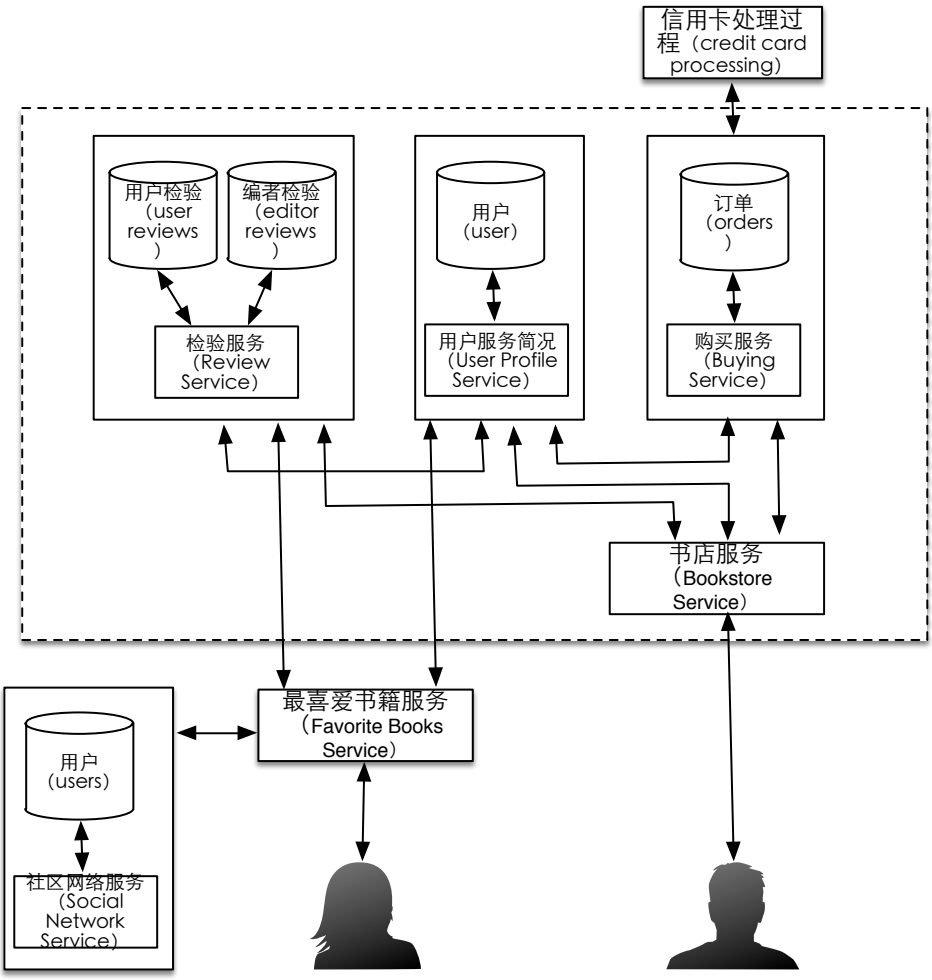


图 1.3: 图示为一个虚拟的书店服务的SOA版本，图中的三个子系统是独立的而且可以通过API建立连接

糕。请注意，SOA与传统的软件分层模型并没有对应关系。在筒仓式软件中每一个更高的层都是在低一层基元上建起来的。然而SOA是纵向切分功能，跨了很多层，之后将这些切分连接起来形成服务。建立一个SOA意味着会比筒仓式架构多一点点的工作量，回报是SOA会有巨大的重用性。SOA另一个优势是明确的API会使得测试更加简单。

SOA有两个大家广为熟知的缺陷。第一，每一个调用服务需要艰难地跃过更深的软件堆栈，所以这会导致一个SOA性能问题。第二，一个独立的系统会经常因为故障完全失败，而SOA的软件工程师不得不面对局部故障的棘手情况，所以说，SOA的可靠性规划带来挑战。

> 小结：
> 尽管这一概念几乎迷失在混沌之中，**面向服务的架构**（**SOA**）只意味着一个软件开发方法使得所有的子系统只对其外部服务有效，这说明其他子系统可以通过不同方式进行重组。接下来的书中所给出的工具和指引会帮助你SaaS应用更适合SOA的设计。

John McCarthy（1927——2011）在1971年获得图灵奖，他是Lisp语言的发明者。作为分时操作大型计算机的先驱者，早在1961年他就设想一个“生态系统（ecosystem）”，预示了当今的软件即服务中的大型计算机服务商给大量客户提供商定好价格的持续服务。集群的商品硬件，以及快速的网络传播使这一设想变为现实。

自我检查1.3.1. *SOA*的另一个优势在于它提高了程序员的工作效率。效率机制使得程序员用更少的时间编写出更多的程序，*SOA*中哪种机制凸现其高效性、简洁的界面、合成、代码再利用或者自动化工具?

◇ 当然是代码再利用！通过代码再利用使得内部API可见，程序员就可以站在别人的肩膀上了！

理解了SaaS和它对于面向服务的架构的依赖，我们可以开始讨论底层的硬件如何使得SaaS成为可能。

1.4 云计算

> 如果我所主张的计算机，能成为未来计算机那么计算功能有一天就像电话系统那样成为公共基础设施，公共计算服务可能会成为一个新的重要的基础行业。
>
> ——John McCarthy，1961年于麻省理工百年校庆

SaaS对我们的信息技术（IT）基础设施的三大要求：

(1) 通信基础，允许任何客户与服务进行交互。

(2) 随时间变化的需求，例如每日的变化或者季节性的变化，必须同时支持一些用户数快速增长的服务的需求。

(3) 可靠性，服务和通信必须每时每刻都能运转：七天二十四个小时（“24×7”）.

黄金标准：美国的公共电话99.999%是可用的（被称为“五个九”可用时间），这意味着需要保证每年停机时间不超过5分钟。

家庭互联网与宽带很容易满足SaaS对通信的需求。虽然一些早期的Web服务部署在十分昂贵的大型计算机上——这样的计算机更可靠，因为同时操作少量大型计算机更为便利相反的——趋势很快就占领了整个行业。

计算机通过以太网交换机连接小规模的计算机，被称为 **集群** 。与“大型机”硬件相比，集群有以下的优势：

- 因为它们依赖于以太网交换机互连，集群的服务器比传统服务器更具扩展性。早期的集群提供了1000台计算机，如今的数据中心包含100000台或更多。

- 仅仅少数操作员就可以维护数以千计的服务器，通过运营商精心挑选出的一系列放置在数据中心的硬件以及对软件的准确控制就能办到。尤其是一些数据中心依靠 **虚拟机** 来简化操作。虚拟机管理程序是一个模拟硬件的软件系统，以至于你甚至可以在它提供的虚拟机的抽象上正确地运行一个操作系统（Rosenblum and Garfinkel 2005）。我们的目标就是以低开销来模拟，并且一个受欢迎的使用就是在集群内简化软件分布。

- 两名谷歌公司的高级架构师发现，集群中数目相当的处理器、内存和存储的成本要比“大型机”要少得多，甚至有可能是大型机的1/20（Barroso and Hoelzle 2009）。

- 尽管集群组件比传统的服务器和存储系统的可靠性要差一些，但是集群软件基础设施通过广泛利用冗余可以让整个系统变得可靠。低成本的硬件使得这些冗余可支撑起软件层面。现代服务提供商也使用分布在不同位置的多数据中心，使得自然灾害来临的时候服务不会瘫痪。

随着互联网数据中心的增多，一些服务提供商意识到在集群的数据中心运行数据所花费的成本远远小于使用自己的小型数据中心，这主要是因为规模经济效应，比如当一次购买和操作10万台计算机。这些服务提供商也能从更高的使用率上获利，因为这些庞大的，通常叫做 **仓库规模的计算机** （Barroso and Hoelzle 2009）的数据中心可以被很多公司共享。相比而言，小的数据中心通常只有10%∼20%的利用率。因此，这些服务商意识到他们可以把其数据中心硬件租赁用给别人并以此收费盈利。

其结果就带来了 **公共云服务** 或称为 **公用计算服务** 。用户仅需每小时几美分就能得到服务器所提供的计算、存储和通信服务（详见文献Armbrust et al. 2010）。此外，扩展规模不增加额外的费用：1小时同时使用1000台计算机，费用不超过使用1台计算机1000小时。“无限可扩展性”和随用随付计算的主要例子是亚马逊网络服务、谷歌的App Engine和微软Azure。公共云服务意味着今天任何人，只要有张信用卡并且有个好主意，都可以开一个SaaS公司，并有机会发展到数百万用户，而无须先建造并运营一个数据中心。

我们相信云计算和SaaS正在改变计算机行业，这场革命将会在接下来的十年影响全世界。事实上，正是接下来的这场革命启发我写这本书，因为我相信SaaS工程和云计算从根本上是与原来的PC机和服务器上的封装好的软件是有本质区别的。

FarmVille崛起
在FarmVille[a]出现之前，世界上用户数目最多的网络游戏约有5百万用户。FarmVille 运营的四天之内就有1百万注册用户，之后的两个月注册用户数就达到了1千万，到了第9个月注册用户就已经达到7千5百万，并且有2千8百万用户每天都使用FarmVille。幸运的是FarmVille使用的是亚马逊所提供的弹性计算云（EC2）服务，因此能够仅仅通过多付一些钱，就很好地处理如此之快的用户数量增长。

[a]译者注：FarmVille是Facebook上类似于开心农场的偷菜游戏

小结：

- 互联网为SaaS提供通信服务。
- **云计算** 为SaaS提供一个可扩展和可靠的硬件计算和存储。
- 云计算包括 **集群计算** 这样的商业服务器并通过局域网交换机连接。软件层提供足够的冗余，使得高性价比的硬件更加可靠。
- 这些大型集群或 **仓库规模的计算机** 提供一定规模的经济效益。
- 利用规模经济，一些云计算供应商提供他的硬件基础设施就能获利，低成本的 **实用计算** 使得任何人都可以使用它，一旦付完款就可以立刻获取资源并且用户使用完资源就可以立刻释放空间等待下一个客户使用。

自我检查1.4.1. 对或错：如果用户接受*SOA*并且配备同一类型的硬件，内部数据中心可以跟仓库规模的计算机一样节约成本。

◇ 错误。模仿仓库规模的计算机的运营模式虽然可以降低一部分成本，但是大型数据中心的核心优势是通过规模经济来实现的。规模经济在当前至少是指10万台服务器，这是内部数据中心无法比拟的规模。

1.5 漂亮的代码与遗留代码

与硬件不同，人们预期软件将会随时间不断发展。硬件需要设计完成很久才可以正式的生产/出货，然而软件设计可以很早就推出新产品，然后在接下来的时间再慢慢修复BUG并且进行升级。另外，硬件升级的开销基本上都是天文数字而软件升级的花费却平民得多。

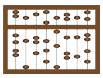
因此，软件是高科技中的不朽的产品，有可能随着时间的推移变得更好，然而计算机硬件很快就会被淘汰。**软件进化**不仅仅是修改错误，根据客户需求增加新的功能并适应新的环境才是重中之重。客户十分乐意看到软件升级和安装改进版本，在软件的整个生命周期中，客户会不断提交错误报告以帮助开发人员修复他们的代码。他们会为这一系列服务而支付软件每年的维护费用！

正如小说家天真地希望他们的创作作品能够贴上经典的标签———对书籍而言是100年———软件工程师也希望自己的作品是持久的。当然软件相比于书籍的优势在于随着时间能够进行改善。事实上，软件生命周期长往往意味着其他人保持和增强了它，让源代码的创作者脱离这个过程。

现存的最古老的项目可能是MOCAS[14]（“机械化的合同管理服务”），它最初于1958年由美国国防部购入的源代码并且到了2005年依然在使用。

这给我们带来了一些将在这本书里使用的术语。术语**遗留的代码（legacy code）**一词是指软件尽管很早就投入使用，但因为它满足客户的需求，所以直到如今我们依然在用它。60%的软件维护成本是在遗留的软件里增加新的功能，只有17%花费在修复错误上，所以遗留的软件是成功的软件。

术语“遗留的（legacy）”有负面的含义，因为它表明代码很难发展，由于它的设计问题或使用不优雅的技术。与遗留的代码相比，我们使用术语**漂亮的代码（beautiful code）**来表述很容易发展的持久代码。然而最坏的情况不是遗留的代码，而是**出人意料的短命代码（unexpectedly short-lived code）**，因为它不符合客户的需求很快就会被丢弃。我们在文中用一个蒙娜丽莎的图标来强调可以帮助写出漂亮的代码的部分。类似的，我们用算盘的图标来强调介绍如何处理遗留代码例子，因为算盘显然是一个持久而很少改变的计算设备。



令人惊讶的是，尽管大家都普遍觉得增强现有软件是非常重要的，但是传统的大学课程和教科书却忽略了这个话题。在这本书中，我们重点着笔于这样的软件有两个原因。第一，你可以通过现有的代码重用省时省力地编写一个程序。第二，它有利于帮助我们学习如何构建代码，这使得继任者更容易增强原程序，因为它增加了软件保持长寿命的机会。在下面的章节中，我们将展示“漂亮的代码”和“遗留代码”的例子，我们希望这两个例子能够给你带来灵感，并且让你的设计更精简。

小结：
成功的软件能存活几十年，并将会发展和提高，而不像计算机硬件，在制造的时候就定型了并且在仅仅几年之内就会过时。这本书的目标之一是教你如何成功的编写一套漂亮的代码，并使你的代码度过一个漫长而且有用的生命周期。

接下来，我们定义了软件质量并将探讨如何进行测试以增加编写漂亮的代码的机会。

1.6 软件质量保证：测试

用户带着嘲笑和讽刺叫嚣道：
“这的确是我们提出的需求，但不是我们想要的。”

——匿名

我们以质量的定义来开始这个话题。对任何产品来说 **质量** 的标准定义就是“适合使用”，即它必须为客户和制造商提供商业价值（Juran and Gryna 1998）。对于软件，质量意味着同时满足客户的需求——使用方便，运行结果正确，不崩溃等等—— **并且** 还要便于开发人员进行调试和改进。 **质量保证** （**QA**）这一词也来自于制造业，指的是有助于生产高质量产品的过程和标准，也指对于制造过程的改进。软件的QA则意味着既要保证正在开发的产品具有高品质，又要保证创造出一系列流程和标准从而生产高质量的软件。正如我们将要看到的，一些传统的软件过程甚至使用一个单独的QA团队测试软件质量（8.10节）。

决定软件质量好坏涉及两个术语，它们通常可以互换但是有细微的区别（Boehm 1979）。

- **验证（Verification）**：你所编写的东西是 **对的** 么？（你有没有满足规范要求?）
- **确认（Validation）**：你编写的代码是我们要的 **东西** 么？（这是客户所需要什么？也就是说，规范本身是否正确吗？）

软件原型通常有助于确认而不是验证，因为客户一旦开始看到产品工作就会改变他们对想要的东西的想法。

验证和确认的主要方法是 **测试**；我们将介绍一种替代测试，称为 **形式方法**，在第8章会详细描述。测试的动机是开发商越早发现错误，修复它们就越便宜。鉴于巨大的维修成本，有大量的研究者和实践者工作在“测试”这一研究领域，他们写了大量的书并探讨了很多课题。在这里我们只是强调了其中一些主要的问题。

考虑到大量不同组合的输入，测试往往不能面面俱到。减少测试空间的一种方法是在软件开发的不同阶段来执行不同的测试。从最下层开始，**单元测试** 是为了确保单个过程或者方法是否符合预期。下一层是 **模块测试**，也就是跨单元测试。例如，单元测试在单个类中进行，而模块测试则跨类进行。更高一层是 **集成测试**，确保单元之间的接口有一致的假设和正确的沟通。在这一层不测试单元的功能。最高一层叫做 **系统测试** 或者 **验收测试**，就是测试综合程序是否符合要求的规格。

全面测试的不可行性
假设仅用1纳秒测试一个程序，如果我们希望彻底测试一个64位输入的程序（显然，大多数程序需要更长的时间来运行以及有更多的输入），这一简单的例子就需要2^{64}纳秒，也就是500年的时间！

另一个今天不太常见的角度，是区分完全基于软件外部规范的 **黑盒测试**，与 **白盒测试**（也被叫做 **玻璃盒测试**），它的设计反映了外部规范没有体现出的软件实现的知识。例如，一个哈希表外部规范只能表明当存储键/值对并读了关键字之后，我们应该能找回了存储的值。黑盒测试通过指定一组随机的键/值对来测试这个行为，而白盒测试则利用哈希函数具体实现的知识来构建最坏情况下的测试，例如用会导致很多的哈希冲突的数据来进行测试。

本书给了方法和多层次的测试，从单元测试到验收测试，它们需要指出一部分之前已经被测试过的可能路径。如果我们用图代表路径，那么 **测试覆盖率** 则是指出哪些路径被测试覆盖了。注意，即使100%的测试覆盖率也没有保证设计的可靠性，因为它没有反映关于运行测试 **质量** 的任何事情。

考虑到软件的长寿命，另一个问题就是设计后期的改变是否会导致以前通过的测试的失败。为防止这样的倒退现象，软件工程师使用 **回归测试** 自动重新运行旧的测试，以确保我们的代码至少可以像过去一样工作。一个相关的术语是 **持续集成** 或称 **CI** 测试，这意味着每次新的代码提交时整个程序都会被测试一下，而不是在完成

开发之后有一个单独的测试阶段。第8.9节讨论通过调整源代码测试你的测试代码（**变异测试**），以及通过向程序掷出随机数据来观测在哪部分停止工作来测试端口（**模糊测试**）。

小结：
测试可以减少设计中出现错误的风险。

- 各种形式的测试有助于 **验证** 软件是满足规范的以及 **确认** 设计是否满足了客户需求。
- 为了应对全面测试的不可行性，我们采取分治法，专注于 **单元测试** 、 **模块测试** 、 **集成测试** 以及最高一层的 **系统测试** 或者 **验收测试** 。每一个高级别的测试给低级别指派更多的测试。
- **黑盒** vs. **白盒** （或者 **玻璃盒测试** ）指的是测试是否依靠外部规范或一个模块的实现。
- 通过把设计映射到图像然后记录所有遍历过的节点和弧， **测试覆盖率** 让我们可以很清楚地知道哪些我们测试过，哪些部分没有测试过。
- **回归测试** 重新应用旧的测试来减少新的版本破坏之前已经工作的设计的可能性。

自我检查1.6.1. 以下的所有方式都可以帮助验证，那么哪一种形式的测试最可能帮助确认：单元测试、模块测试、集成测试、系统测试?

◇ 确认是关心客户真正需要的，而不是代码是否符合规范，因此系统测试是最有可能指出做这件事是正确的和做正确的事情之间的区别的。

讨论完质量保证之后，让我们来看看如何提升开发人员的效率。

1.7 生产力：简练、合成、代码重用和工具

> 如今大部分的软件很像数以百万计的砖块堆放在彼此上面建成的埃及金字塔，缺乏结构完整性，只能通过蛮力和成千上万的奴隶来堆积起来。
>
> ——Alan Kay， **ACM Queue杂志** ，2005

摩尔定律意味着硬件资源规模每18个月翻一番。这些有更大内存的速度更快的计算机可以运行更大的程序。为了构建可以利用更强大的计算机的更大的应用程序，软件工程师需要提高他们的生产率。

工程师们提出了四个基本的机制，以提高他们的效率：

(1) 简洁清晰

(2) 合成

(3) 代码重用

(4) 通过工具实现自动化

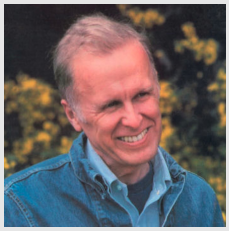
一个提高程序员的效率的关键假设是，如果程序更容易理解，那么它们将会有更少的错误并且更容易发展。一个密切相关的推论是，如果程序更小，那么它通常更容易理解。我们的座右铭“精炼造就清晰”恰恰阐释了这个概念。

编程语言以两种方式来做到这点。第一个就是简单的提供一种语法，让程序员能够自然地并且以更少的字符来表达他们的想法。例如，下面是两种表达一个简单的判定的方式：

```
assert_greater_than_or_equal_to(a, 7)
a.should be >= 7
```

毫无疑问，第二个版本（这恰好是合法的Ruby）更短且更容易阅读和理解，并且可能更容易维护。我们很容易想象第一个版本中除了需要阅读两倍的字符量所带来的更高的认知负荷之外，参数的顺序也会引起的不少混淆。（详见第3章）。

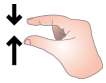
John Backus
（1924——2007）因为对“实用高级编程系统设计的深刻、影响深远以及持久的贡献，特别是通过他在Fortran上的工作”，于1977年获得图灵奖，Fortran是第一个被广泛使用的高级语言。



第二种提高代码清晰度的方法是提高抽象层次。

最初这意味着更高级编程语言如Fortran和COBOL的发明。这一步把开发软件从针对一台特定计算机的汇编语言，提升到一个更高级的语言，高级语言可以简单地通过改变编译器，运行在多台计算机上。

随着计算机硬件性能的不断提高，越来越多的程序员愿意将之前需要他们自己完成的任务委托给编译器和运行时系统。例如，Java和类似的语言从早期的C和C++语言手中接管了内存管理。Python和Ruby等脚本语言都提出了更高的抽象水平。例如允许程序自我检查的 **反射** 和允许程序在运行时修改它们自己的结构和行为的 **元编程** 。为了突出通过简洁性提高效率的例子，我们在文中将会使用这个“简明”图标（见边栏图标）。



第二种生产率的机制是合成。也就是说，实现是自动生成的而不是手动创建的。逻辑合成对硬件工程师来说意味着，他们可以以布尔函数描述硬件，并且通过软件自动获得高度优化的晶体管图来，实现各种功能。经典的软件合成的例子是 **位块传输** 。这个图形基元结合了掩码控制下的两个点阵图。最直接的方法是在最内层的循环里包括一个条件语句来选择掩码的类型，但是它很慢。解决方法是写一个程序，它可以合成相应的特殊用途的代码 **而不用** 在循环里包含一个条件语句。我们将会在边栏中用“CodeGen”齿轮图标突出显示这些通过生成代码来提高效率的例子。

第三种生产率机制是重用过去设计的部分程序而不是从头开始写代码。因为在软件中做少许的改变比在硬件里要更简单，软件甚至比硬件更可能重用一个基本上可用但不是完全适合的一个组件。我们用边栏中“重用”这个回收图标来突出显示这些通过重用来提高效率的例子。

程序和函数是在软件的最早期的时间里发明的，目的在于使程序的不同部分可以使用不同的参数值来重用相同的代码。输入/输出和数学函数等的标准库紧随其后，使得程序员可以重用别人开发的代码。类似的，硬件工程师也有类似的过程和设计库。

程序库让你可以重用单个任务的实现。但更常见的是程序员希望重用和管理任务的 **聚合等抽象数据类型** 。因此软件重用的下一步是 **面向对象编程** ，在那里通过使用像C++和Java语言里的继承，你可以用不同的对象重用相同的任务。

继承支持实现的重用，另一个重用的机会是即使实现有差别，仍可以奏效的一般性策略。受到建造民用建筑的启发产生的 **设计模式** （Alexander et al. 1977）被用来解决这个问题。语言支持的设计模式的重用方面包括利于构造抽象模型的 **动态类型** 和 **mix-in** ，后者提供了从多个方法收集功能的方式，而没有一些在面向对象编程里的多重继承所带来的弊病。Python和Ruby中有许多能够帮助重用设计模式编程语言功能。

请注意，重用并 **不** 意味着简单的复制和粘贴代码。复制和粘贴会导致在很多地方出现非常相似的代码，其带来的问题是，当修复问题或添加一个功能时，你可能

并没改变所有的副本。这里有一个软件工程的指导原则，无重复：

> 任何一部分的知识在系统内必须有一个单一的、明确的、权威的表示。
>
> ——Andy Hunt和Dave Thomas，1999

这条指导原则在座右铭和缩略词里已经讲过了：**无重复原则（DRY，Don't Repeat Yourself）**。我们会下面的章节来给出DRY的例子，我们在以后章节的边栏中用"干毛巾"图标来表示。

计算机工程师的核心价值在于寻找一种工具来代替原先繁琐的手动操作，这样可以节省时间，提高精度，或两者兼之。计算机辅助设计（CAD）软件开发工具是编译器和解释器，可以提高抽象层次并且生成上面提到的代码，但是还有更微妙的提高生产率的工具，像Makefile和版本控制系统（参见10.4节）就能使沉闷的任务自动化。我们在以后章节的边栏中用"锤子"图标来突出表示工具的例子。

学习一个新的系统 国王詹姆斯圣经箴言14:4部分中通过抽出时间学习和使用工具来讨论提高生产率：**没有牛的地方，食槽就是干净的；但是高产量的农作物往往靠的是牛的力量。**

我们始终需要权衡学习一个新的工具所用的时间与应用它所节省的时间。其他的一些问题就是工具的可靠性，用户体验的质量以及如果有很多选择的话，决定挑选哪一个。尽管这样，软件工程的一个原则仍旧认为新的工具可以使我们的生活更美好。

你们的作者接受了自动化和工具的价值。这就是为什么我们在这本书里展示多个工具来让你的开发更有效率。好消息是任何我们展示给你的工具都经过我们的审查，以保证它的可靠性，以及学习它的时间将会在减少的开发时间和提高最终结果的质量上得到成倍的补偿。例如，第7章讲述 **Cucumber软件** 可以自动把用户故事转化为综合测试。我们还介绍了 **Pivotal Tracker** 工具是如何自动测量 **速度（Velocity）** 的，Velocity是一个衡量向应用程序中添加功能的速度的概念。第8 章介绍了自动处理单元测试的 **RSpec** 。坏消息是你想要得到这些便利条件都需要学习新的工具。然而，我们认为能够快速学习和应用新的工具的能力在软件工程项目的成功是不可或缺的，因此我们有必要培养这一技能。

因此，第四种提高生产率的方法是通过工具实现的自动化。我们在边栏使用机器人图标来强调使用自动化的例子，尽管这些例子通常也与工具有关。

小结：
摩尔定律激发了软件工程师提高效率，通过：

- 凯觎简洁，使用紧凑的语法和使用更高级的语言来提高设计水平。最新进展包括允许程序来观察自己的 **反射** 以及在运行时修改其自身的结构和行为的 **元编程** 。
- 合成实现。
- 遵循 **无重复（DRY）** 原则来重用设计，以及依赖利于重用的创新，例如过程，库，面向对象编程以及设计模式。
- 使用（或者发明）CAD工具来自动化繁琐的任务。

自我检查1.7.1. 从高级编程语言编译器的生产效率方面来说哪一种机制是最站不住脚的因素：通过简洁达到的清晰，合成，重用或者自动化的工具？
◇ 编译器使高级编程语言变得实用，使得编程者可以用高级语言（HLL，High Level Language）写出更加简洁的代码来提高效率。编译器合成基于高级语言的低级代码。编译器肯定是工具。虽然你可以认为高级语言让重用变得更简单，但是重用是这四个里面在解释编译器的效益方面最站不住脚的。

考虑到SaaS的快速进化的需要，我们需要找到一个软件开发过程，其中变化是常态而不是异常。让我们从传统的软件开发过程开始讨论，尽管事实上它不太适合SaaS。

1.8 软件开发过程：计划和文档过程

> 如果一个建筑工人像程序员写程序的方式一样建房子，那么第一只出现的啄木鸟就能摧毁人类文明。
>
> ——Gerald Weinberg，**温伯格第二定律**

软件开发的不可预测性以及像之前提到过的软件灾难，都导致了我们需要研究在一个可预测的计划和预算上开发高质量软件的方式。**软件工程** 这个术语是类比其他工程领域中时杜撰出来的（Naur and Randell 1969）。软件工程的目标是发现新的方法来建立一个软件，使得它在质量、成本和时间上，和土木工程中建造桥梁一样可预测。

软件工程的一个推力就是给经常无计划的软件开发带来一个工程性规则。在开始编写代码之前，给这个工程想出一个计划，包括涵盖计划的所有阶段的广泛的、详细的文档。当进展情况在计划之外的话要给出必要措施，对现有项目的修改必须能反映在文档中和可能在计划中。

1970年就有一个早期的版本，关于软件开发过程的"计划文档（Plan-and-Document）"（Royce 1970），它遵循以下顺序：

(1) 需求分析和规范

(2) 架构设计

(3) 实现和整合

(4) 验证

(5) 操作和维护

考虑到你越早发现错误，修复它所花的时间与金钱就越少，这个过程反映的哲理就是在软件开发进行到下一阶段之前就完成上一个阶段，因此可以尽早地消除尽可能多的错误。正确的早期阶段也可以避免不必要的下游工作。因为这一过程可能需要持续数年之久，大量的文档确保如果有人离开这个项目时重要信息不丢失，当新人加入该项目进行交接速度会很快。

因为它都是自上而下的完成的，所以这个过程被称为 **瀑布** 软件开发过程，或瀑布软件开发 **生命周期** 。可以理解的是，由于在瀑布生命周期每个阶段都十分复杂，瀑布软件的产品发布都被当作重大事件那般受重视，工程师们都为之疯狂并且狂热地工作着。

在瀑布生命周期中，软件的长寿命是通过发现错误及时修复错误的维护阶段来保证的。使用瀑布模型的新版本的软件也要经历相同的步骤，通常需要6个月～18个月。

瀑布模型可以很好地配合特定的任务，如NASA的太空飞行，但是当客户改变他们想要什么的想法时，瀑布模型往往运行会陷入困境。图灵奖获得者抓住了这一细节：

Windows 95 是在耗资300万美元的户外派对[15]上宣布的，微软公司为此聘请喜剧演员杰伊·莱诺，使用微软Windows的商标投影在整个帝国大厦照亮了全纽约，以滚石乐队的"Start Me Up"作为庆祝的主题曲。

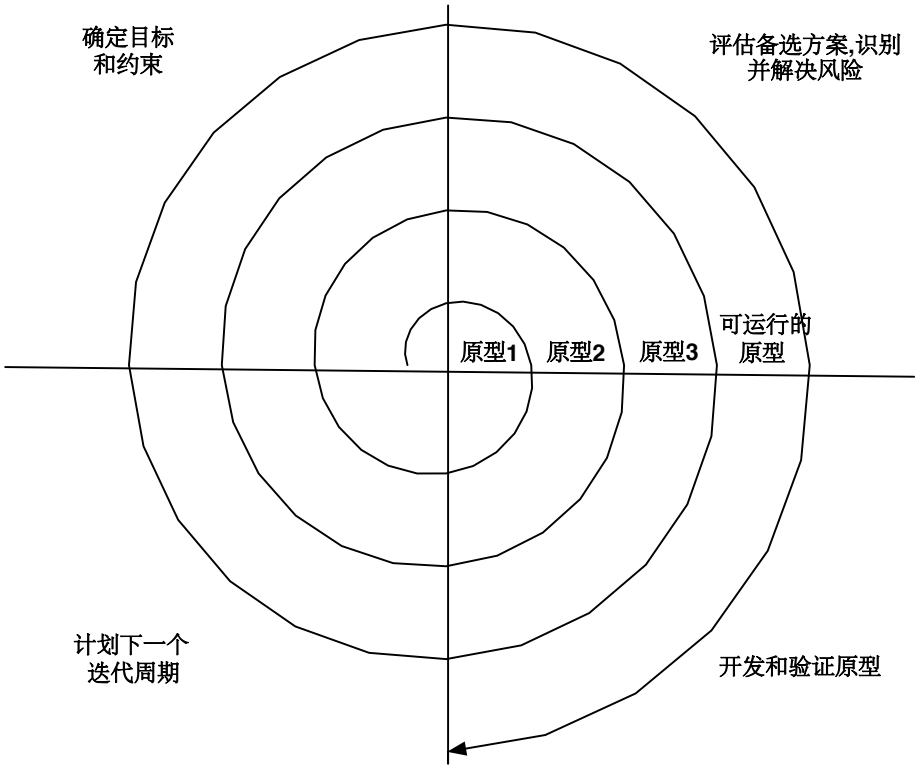


图 1.4: 螺旋生命周期结合瀑布的原型。它从坐标原点开始，每个迭代螺旋经历的四个阶段并且不断的修改原型，直到产品已准备就绪

永远要准备好第二套方案；因为你很有可能会舍弃掉最初的方案。

——Fred Brooks，Jr.

也就是说，客户一旦看到一个原型就更容易理解他们想要什么，而工程师一旦做过了第一次之后就更容易理解如何更好地做一件事。

这个观察导致了20世纪80年代开发的软件开发生命周期，它结合了瀑布模型和原型（Boehm 1986）。主要思路是遍历序列的四个阶段，每一次迭代产生的新模型都是提炼之前的版本。图1.4 说明了这个模型在四个阶段的发展，并且说明了这个生命周期的名字：**螺旋模型**。这些阶段是

(1) 确定本次迭代的目标和约束

(2) 评估各种备选方案，并确认和化解风险

(3) 开发和验证本次迭代原型

(4) 计划下一次迭代

与瀑布模型那不同所有的需求并不需要在一开始就全部提出，螺旋模型的需求文档是在迭代过程中当它们被需要时开发出来的，并且将会随着项目发展。迭代在

产品完成之前就让客户参与进来，这会减少误解的机会。但是，正如最初设想的，这些迭代需要6至24个月，因此一次迭代期间有充足的时间让客户改变他们的想法！因此，螺旋仍然依赖于计划和广泛的文档，但是这个计划预计会在每次迭代中进化。

正因为软件开发的重要性，除了这两个之外人们也提出了“计划与文档化”方法的许多变种。最近一个叫 **Rational Unified Process** （**Rational统一过程，RUP**）（Kruchten 2003），它结合了瀑布和螺旋生命周期的功能，兼有图表和文档化的标准。我们将使用RUP作为计划和文档生命周期的最新思维方式的代表。与瀑布和螺旋不同和技术问题相比，它与商业事务结合得更紧密。

预先进行大方向的设计缩写为 **BDUF**，**Big Design Up Front**，是一些依赖于广泛的计划和文档的软件流程，例如瀑布、螺旋和RUP的软件过程的统称。这些软件过程有时也被叫做**重量级**（**heavyweight**）、**计划驱动**（**plan-driven**）、**纪律性**（**disciplined**）或者**结构性**（**structured**）过程。

与瀑布和螺旋一样，RUP有如下几个阶段：

(1) 构思阶段：为软件做一个商业案例，设置项目的流程跟预算，用来判断进程和确认支出，最后做出关于进度和预算风险初步评估。

(2) 细化阶段：与利益相关者一同协作来确定使用实例、设计软件架构、设置开发计划，并建立一个初始原型。

(3) 构建阶段：编写代码和测试产品，从而推出第一个对外发布的版本。

(4) 移交阶段：将产品从开发转移到实际环境中进行测试，包括客户的验收测试和用户教学。

与瀑布模型不同，每个阶段都会进行迭代。例如，一个项目可能有一个初始阶段迭代，两个详细阐述阶段迭代，四个构建阶段迭代，和两个过渡阶段迭代。就像螺旋，一个项目也可以在四个阶段中反复迭代。

除了项目的动态变化阶段，RUP确定了六个“工程规范”（也称为工作流程），所有参与此项目的人应该共同考虑：

(1) 商业建模

(2) 需求

(3) 分析与设计

(4) 实现

(5) 测试

(6) 部署

这些规则比阶段更为稳定，因为他们贯穿于整个项目的生命周期。然而，有些流程更倾向于在早期阶段更多地使用（如商业建模），一些定期实施（如测试），一些更适合在尾声阶段实施（如部署）。图1.5展示了阶段与规则之间的关系，每一片区域面积表示每一个规则随时间的工作量。

所有的这些计划与文档生命周期的目标是通过广泛的文档来提高软件开发过程中的可预测性，只要目标改变这个文档就一定会变化。下图是一些教科书作者的见解（Lethbridge and Laganiere 2002; Braude 2001）：

> 文档应该写在发展的各个阶段，包括需求、设计、用户手册、测试人员和项目计划的说明。
>
> ——Timothy Lethbridge 和Robert Laganiere，2002

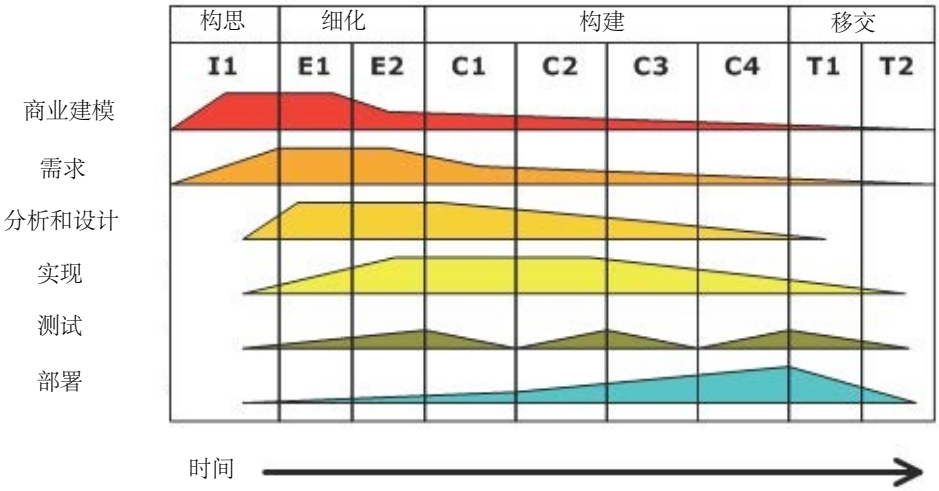


图 1.5: Rational统一过程生命周期允许项目在每一个阶段多次迭代，并指出项目团队在不同的时间所需要的技能。RUP同样也有三个"支持规则"在图中没有显示出来：配置和变更管理、项目管理和环境（插图来自维基百科，作者Dutchgilder）

文档是软件工程的生命线。

——Eric Braude, 2001

这个过程甚至有一个官方的文档标准：IEEE/ANSI 标准830/1993。

这个严格的方法教学中的一个不幸的缺点就是，学生可能会觉得软件开发非常繁琐（Nawrocki et al. 2002; Estler et al. 2012）。但是鉴于可预测的软件开发的重要性，这个理由不足以让我们不教授它；好消息是，我们在探索教学过程中发现一个好的替代过程，也更加适合课堂教学，将在下一节讲解这一方案。

小结：
软件工程 **活动** 基本的软件开发过程 **生命周期** 是相同的，但随着时间的推移，产品发布的交互过程是不同的。瀑布的生命周期，其特点是许多的设计在编码之前就完成了，每个阶段完成才进行到下一个。螺旋生命周期整个生产原型开发阶段都进行迭代，但就像瀑布一样客户只能在迭代所需要的6至24个月修改项目。Rational统一过程与生命周期一样包括阶段，迭代和原型，同时确定项目所需的业务能力。所有以上的三种模型依赖精心的策划和详尽的资料，以及对计划进展进行评估并采取相应措施。

■ 详细阐述：SEI能力成熟模型（Capability Maturity Model，CMM）

卡内基梅隆大学软件工程研究所提出了 **能力成熟模型**（**CMM**）（Paulk et al. 1995）来评估组织基于计划与文档化的软件开发过程，其背后的想法是通过建模软件开发过程，使得一个组织能够对这些过程进行改进。SEI研究发现软件实践分五个层次：

1.初始级或混乱的——无正式文件的/自组织的/不稳定的软件开发过程。
2.可重复级——没有遵循严格的规则，但是某些过程重复时有一致的结果。
3.已定义级——定义和记录随着时间提高的标准过程。
4.已管理级——管理可以使用过程流量来控制软件开发过程，把这个过程成功地应用到不同的项目里。
5.优化级——将主动的优化提高作为管理过程的一部分。

级别越大越好，CMM隐式地地鼓励组织提升其CMM的级别。虽然它不是作为软件开发的方式提出来的，但是许多人还是把它当作一种方式。例如，文献（Nawrocki et al. 2002）对CMM模型中的第二级与敏捷软件方法进行了比较（见下一节）。

自我检查1.8.1. 像瀑布模型、螺旋模型和*RUP*这样的过程之间的主要相似点与区别分别是什么？
◇ 所有的模型都依靠计划文档，但是螺旋模型和RUP使用迭代与原型的方法来提高它们，而不是得到产品的一个单一长路径。

自我检查1.8.2. 这些过程中的各个阶段之间的区别是什么？
◇ 瀑布阶段会把计划（例如需求和架构设计）和实现分开。接下来是在发布之前测试这个产品，紧随其后的是单独的运行阶段。螺旋阶段旨在迭代：设置迭代的目标；探究替代选择；开发和验证迭代的原型；并且计划下一次迭代。RUP阶段和商业目标紧密结合：构思阶段做商业案例并且设置日程安排和预算；细化阶段与用户合作建立一个初始原型；构建阶段建立并测试第一个版本；移交阶段部署产品。

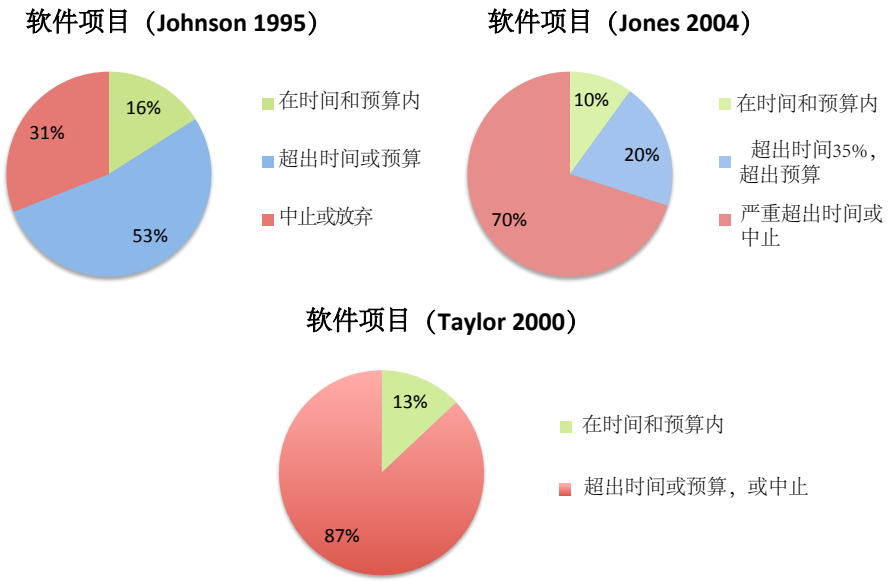


图 1.6: 第一个关于软件项目的研究发现53%的项目都超出预算的2.9倍，并且超过了工期的3.2倍，另外有31%的软件项目在完成前就被取消了（Johnson 1995）。美国每年花费在这样的软件项目的成本估计是一千亿美元。第二个研究调查了250个大型项目，每个都相当于超过一百万行C语言代码，在调查中发现同样令人失望的结果（Jones 2004）。英国计算机协会的成员最终调查发现，1027个项目中只有130项满足了时间表和预算。一半的项目是维护或者数据转移，另一半是新开发的项目，但是成功的项目里有127个是前者，只有3个是后者（Taylor 2000）

1.9 软件开发过程：敏捷宣言

> 如果对于一个问题没有解决办法，它可能就不是一个问题，而是一个事实——我们不是要解决它，而是需要在长时间不断应对。
>
> ——以色列前总理佩雷斯

虽然计划和发展过程给软件开发带来了规则，但是大家还是对此有所抱怨。一篇文章提及了“软件耻辱墙”以及31个高度引人注目的软件开发项目，这些项目要对17B美元的亏损负责，它们中的大多数项目都被遗弃了（Charettte 2005）。图1.6小结了三个软件项目的调查。仅有10%～16%按照计划表严格执行并且在预算之内，更多的项目还没实现就被取消或放弃了。仔细审视第三个研究中13%的成功率会更发人深省，因为只有小于1%的新开发项目满足了他们的计划表和预算。

也许这些有问题的项目没有遵循组织方法，但话又说回来，他们也许真这么做了。事实上，被视为软件灾难的美国联邦调查局虚拟案件文档（1.1节），是以800页的系统要求规则开始的（详见7.10节），这个规则是程序员与联邦调查局合作开发超过6个月才完成的（Goldstein 2005）。

敏捷 有时也被称为 **轻量级的** 或者 **无规则的** 过程。

也许软件工程的“改革的里程碑”是2001年2月的 **敏捷宣言** 。一组软件开发人员一起开发一个轻量级的软件生命周期。以下是 **敏捷联盟** 钉到“计划与文档”殿堂大门上的内容。（译者注：此处作者是在类比马丁路德在天主教教堂门前钉上了宗教改革的宣言，开始了具有开创性意义的新教改革）。

“**我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。**

由此我们建立了如下价值观：

- **个体和互动**高于 **流程和工具**
- **工作的软件**高于 **详尽的文档**
- **客户合作**高于 **合同谈判**
- **响应变化**高于 **遵循计划**

也就是说，尽管右项有其价值，我们更重视左项的价值。”

这个可选的开发模型是将接受变化作为生命中的常态：开发商应不断改善有效的、但是不完整的原型直到客户满意为止，每次迭代时客户都会提供反馈。敏捷强调通过在编写代码 **之前** 编写测试的 **测试驱动开发（Test Driven Design，TDD）** 来减少错误，通过 **用户故事** 来达成协议并确认客户的需求，以及用 **速度** 来衡量项目的进度。将在后面的章节介绍这些内容。

关于软件的生命周期，敏捷软件生命周期非常短，以至于短短一周或两周就有新的版本——有的敏捷软件甚至每天都有新的版本——所以与一个计划文档模型不同，软件发布不再是什么重大事件。

敏捷的变种 敏捷软件开发有许多变种（Fowler 2005）。在这本书中，我们使用的是Kent Beck发明的 **极限编程（Extreme Programming）简称 XP** 。

我们在之前的部分提到，新手会发现计划和文档过程十分繁琐，但是在敏捷的情况下就不会这样。软件工程的教师在早期对敏捷开发的思考中发现：

> 你还记得编程的乐趣吗？这就是你如何首先对计算机感兴趣，之后才对计算机科学感兴趣的吗？这就是为什么有很多选择了这门专业——因为他们喜欢给计算机编程？嗯，那么可能有前途的和受人尊敬的软件开发方法很适合这类人。……[敏捷] 是十分有趣而且高效的，不仅是因为我们没有在堆积如山的文档的过程中陷入瘫痪，而且还因为从很早的开发过程中，开发人员就会与客户面对面的交流，做出可工作的软件。
>
> ——Renee McCauley,“敏捷开发方法准备不安现状”， **SIGCSE Bulletin** ，2001

通过不再强调计划、文档和合约性绑定规则，敏捷宣言与传统的软件工程学者的认识背道而驰，所以它并没有受到普遍欢迎（Cormick 2001）：

> [敏捷宣言]是另一个企图破坏软件工程规则的尝试……在软件工程专业，有工程师也有黑客……在我看来这只不过是试图合法化黑客的行为……对于软件工程行业只有在客户拒绝支付没有达到合约要求的软件时，才会向更好的方向变化……只有鼓励黑客文化转变到基于可预测的软件工程实践的文化，才会帮助把软件工程转化为一个受人尊敬的工程学科。
>
> ——Steven Ratkin，“宣言抒发玩世不恭”， **IEEE Computer** ，2001

一些批评家甚至出版了一本432页的书来质疑敏捷！（Stephens and Rosenberg 2003）

软件工程研究社区继续比较计划文档生命周期与敏捷生命周期，然后发现——令某些愤世嫉俗者吃惊的是——敏捷实际上可以工作得很好，当然这依情况而定。图1.7向我们展示了来自于一本流行的软件工程教材里的10个问题（Sommerville 2010），答案建议了什么时候该使用敏捷什么时候应该使用计划和文档方法。

一个敏捷的接受度的例子是最近的一个研究，这个研究为地理上分布式的编程团队把它和计划文档化作比较，包括外包项目（Estler et al. 2012）。注意，图1.7中的第7个问题建议地理上分布式的团队使用计划文档，所以看起来敏捷似乎处于劣势。研究发现，调查的66个项目中的大部分都使用敏捷，这说明其日益普及甚至是在分

	问题：回答“No”表明它是敏捷，回答“Yes”表明它是计划文档
1	我们需要这个规则么?
2	客户不可用?
3	这个系统非常庞大么?
4	这个系统非常复杂么（例如，实时性要求）?
5	这个产品的寿命长么?
6	它是否在使用粗劣的软件工具?
7	这个项目团队是按照地理分布的么?
8	这个团队是面向文档文明的一部分么?
9	这个团队是否有着粗劣的编程技巧?
10	这个系统是否按照规定建立对象?

图 1.7: 这10个问题帮助我们判定是否用敏捷生命周期（当答案是No）或者用计划文档生命周期（答案是Yes）（Sommerville 2010）。引人注目的是当我们向一个班里的学生提出这10个问题时，几乎所有的答案都指向敏捷。正如这本书证明的那样，开源软件工具都是非常出色的，因此对学生也是可用的（问题6）。我们对行业的调查（见前言）发现毕业的学生确实有良好的编程技能（问题9）。其他8个答案显然针对学生项目是否定的

布式团队里也是一样。第二，尽管敏捷团队小于计划文档团队（和问题3相关的），但是有些敏捷团队有超过60个人。第三，项目管理的问题在计划和文档项目中出现得更频繁，也许部分是因为敏捷项目有更少的通信问题。作者的底线是，保证地理上分布式开发团队使用敏捷做项目结果可以和计划文档的项目的结果一样好。

事实上，上述的1995年的软件项目调查的作者将准时保质的项目数目的提高归功于日益普及的敏捷。它们在2009年从16%提高到32%，但是超支项目的时间和成本从3.0下降到1.7，并且它们的项目所占比例从53%下降至44%（Johnson 2009）。

当需求已经明确并且没有根本的变化时，软件工程知识分子中的传统智慧的精髓就是使用计划和文档生命周期——因为他们扩展到更大的团队，因此有更大的项目，当需求频繁地改变或者当与利益相关者一起创建一个正式的规则十分困难或者不切实际时（Estler et al. 2012）。可以使用敏捷生命周期。

请注意，SaaS的频繁升级与敏捷软件生命周期完美的结合受益于只有一个单一的软件副本。因此，亚马逊、eBay、Facebook、谷歌和其他SaaS供应商都依靠敏捷的生命周期，并且传统的软件公司如微软在他们的产品开发阶段也越来越多地使用敏捷。

因此，我们在本书第二部分的六章软件开发部分集中在敏捷软件开发，但每章也从计划和文件化方法的角度讨论了像需求，测试，项目管理和维护这样的话题。这种对比让读者自己决定每个方法在什么时候是合适的。第一部分介绍SaaS和SaaS的编程环境，包括了Ruby、Rails和Javascript。

小结：
与计划文档生命周期相比，敏捷与客户合作不断地在项目原型里为客户增加新功能，直到客户满意为止，它允许客户在项目开发时改变他们想要的功能。文档主要记录用户事件和测试用例，并且它不测量一个预定义的方案的进度。相反进步是通过记录 **速度** 来衡量的，这实质上是项目完成功能的速度。敏捷过程可以很好地适应SaaS应用快速变化的性质。

自我检查1.9.1. 对或错：瀑布和敏捷开发之间最大的差异是敏捷开发没有使用需求。
◇ 错误。敏捷开发虽然不像瀑布那样开发广泛的需求文档，但是它与客户的互动以创建需求，例如用户故事，我们将在第7章详细介绍。

自我检查1.9.2. 对或错：螺旋与敏捷开发的一个很大的区别是构建原型以及过程中与

客户的互动。

◇ 错误。两者都构建了工作的但是不完整的原型，客户可以帮助评估。不同的是两个模型与客户的沟通频率不一样，敏捷两周一次，而螺旋两年一次。

■ 详细阐述：测试：计划文档生命周期与敏捷生命周期

对瀑布开发过程来说，测试发生在每个阶段完成后，以及在包含了验收测试的最后的验证阶段。对螺旋来说这发生在每次迭代时，而迭代可以持续一年或两年。敏捷的保证来自于测试驱动开发，其中当从头开始编码时测试在代码 **之前** 编写完成。当增强现有代码时，测试驱动设计意味着编写测试代码在编写增强代码之前进行。测试的数量取决于你所针对的是漂亮的代码还是遗留代码，后者需要更多的测试。

■ 详细阐述：生产力：计划文档生命周期与敏捷生命周期

生产力是以人们实现新的功能所花的时间来衡量的。不同的是周期在瀑布和螺旋模型里比敏捷要更长——螺旋周期为6～24个月而敏捷周期为1/2个月——所以更多的工作是在客户看到的各个版本之间完成的，因此对瀑布和螺旋模型来说，工作最终会被客户否定掉的概率会更大。

1.10 本书导览

不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之。

——孔子

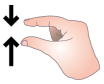
经过介绍之后，我们现在可以解释接下来该做什么以及可能采取什么样的途径。就像孔夫子建议的那样通过做到达到理解，我们从阅读附录A开始。它解释了如何获取和使用本书配套软件。

这本书剩下的部分一分为二。第一部分解释了软件即服务，第二部分介绍现代软件开发并强调了敏捷开发。

第2章以解释SaaS应用架构来开始第一部分，使用一个高度类比，从100000英尺的角度到500英尺的角度来进行。在降落的过程中，你会学到可能已经听说过的许多缩写的定义——API、CSS、IP、REST、TCP、URLs、URI和XML——以及一些广泛使用的流行语：cookie、标记语言、端口号和三层架构。这章还展示了设计模式的重要性，特别是Rails的核心模型-视图-控制器。

我们不是仅仅要告诉你如何建立持久的软件，然后看着你容易这些规则，我们相信你为了理解必须去动手实践。如果有好的工具支持的话，应该尝试好的指南使其变得更加容易。我们相信，如今用Ruby语言写的Rails框架有最好的SaaS工具支持。因此，第3章介绍了Ruby。Ruby的介绍是简短的，因为假设你已经很了解面向对象的编程语言，比如Java。正如上面提到的，我们相信成功的软件工程师将需要在其职业生涯中经常学习新的语言和工具，所以学习Ruby和Rails是很好的练习。

接下来的第4章介绍Rails的基础。我们在第5章介绍了Rails的更多高级功能。我们为了那些想要尽快开始写应用程序的读者，把这部分分割成两章，他们只需要读第4章。然而第5章的内容学习和理解起来更具有挑战性，如果你使用像局部模板，确认、生命周期回调、过滤器、关联和外键这样的概念，那么应用程序可以更加地无重复和简洁。



已经熟悉Ruby和Rails的读者可以自行跳过这些章节内容。

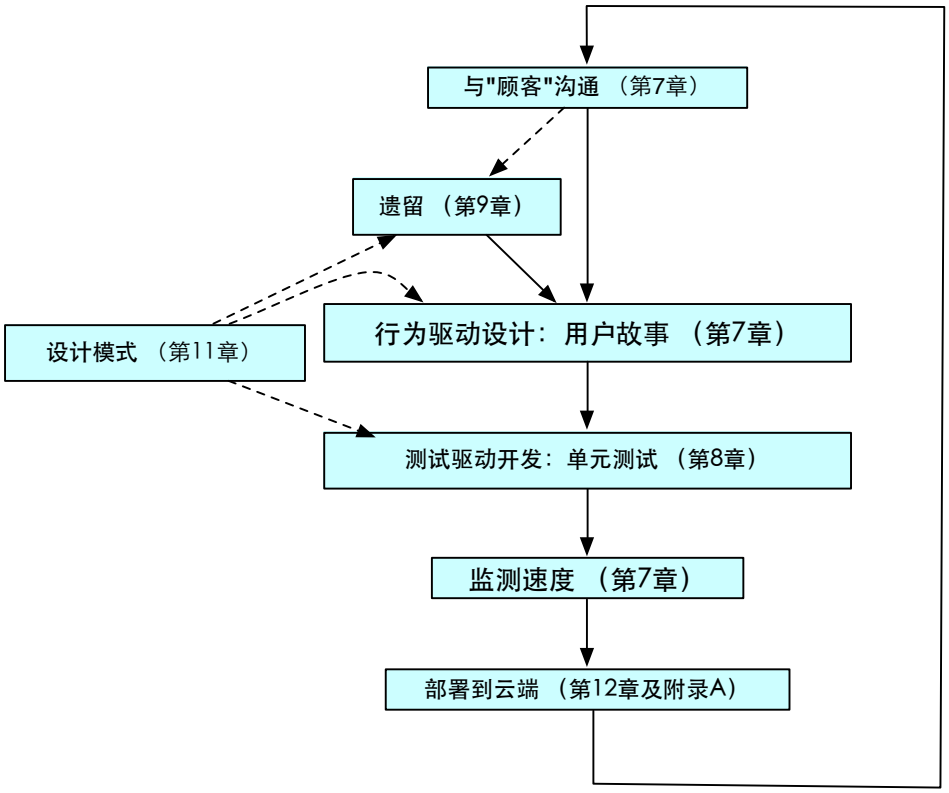


图 **1.8:** 敏捷软件生命周期的迭代和它与这本书的章节之间的关系。虚线箭头表示一个迭代的步骤之间的更为切向的关系，而实线箭头表示典型的流程。请注意，敏捷过程同样适用于现有的遗留应用程序和新的应用程序，虽然客户在遗留的应用程序里可能扮演一个更小的角色

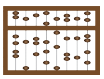
到目前为止本书建立在熟悉了Ruby和Rails的基础之上，第6章介绍编程语言JavaScript，它的高效的jQuery框架以及测试工具Jasmine。

在此背景下，接下来的6章介绍使用Rails的工具来构建和部署一个SaaS应用时，用到的重要软件工程规则。图1.8显示了敏捷生命周期的一个迭代，我们它作为一个框架，本书接下来的章节都围绕它展开。

第7章讨论了如何与客户沟通。**行为驱动设计（BDD）**主张编写没有编程背景的客户可以阅读的测试，称作**用户故事**，而第7章教大家怎样编写用户故事使得它们可以被转化为集成测试和验收测试。它引入了 **Cucumber** 工具帮助我们自动化这一任务。这个测试工具可以被用于任何语言和框架，而不仅仅是Rails。因为SaaS应用通常是面向用户的，所以本章还介绍了如何使用"低保真"原型来构建一个有用的用户界面原型。它也解释了**速度**这个术语以及如何用它来衡量项目的进度，并介绍了基于SaaS的 **Pivotal Tracker** 工具来简化这样的测量。

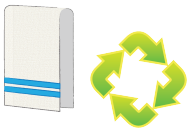
第8章介绍了**测试驱动设计（TDD）**。这章展示了怎样编写好的并且易于测试的代码并且引入 **RSpec** 测试来协助完成单元测试，**Autotest** 工具自动完成测试，而 **SimpleCov** 工具可以测量测试百分比。

第9章介绍了如何处理现有的代码，包括如何增强遗留代码。有益的是，它展示了如何使用BDD和TDD来理解和重构代码，以及如何使用Cucumber和RSpec工具来让这个任务变得更加容易。



第10章给出了建议，告诉我们作为一个有效率的团队的各部分如何组织和工作，而不是所有的东西都是你一个人做。它还介绍了版本控制系统 **Git** 及相应的服务 **GitHub** 及 **ProjectLocker** 如何让团队成员工作在不同的功能上，而不会互相干扰，或者在发布过程中引起混乱。

为了帮助你实践无重复（DRY）的原则，第11章介绍设计模式，这一模式是被证明了的结构性的方案，来解决设计常见的问题中各个类的功能设计，并展示了如何利用Ruby的语言功能来采用并重用这种设计模式。



本章还教大家如何编写好的类。它介绍了刚好够用的UML（统一建模语言）符号来帮助你用符号表示设计模式，以及帮助你做出图表显示类应该如何工作。

请注意第11章的内容是关于软件架构，而第二部分中的第11章之前的章节是关于敏捷开发过程的。我们相信，在大学课程设置中这个顺序会让你更快地开始一个敏捷迭代，并且我们认为迭代的次数越多，就越会更好地理解敏捷生命周期。然而，如图1.8所示，当编写或者重构的代码时，其中的代码是BDD/TDD过程的基础，了解设计模式将会是十分有用的。

第12章提供了实用的建议，关于如何第一次部署然后在云中提高性能和可扩展性，并且简要介绍了一些在部署SaaS时的可靠性和安全性相关的技术。后记回顾了书中的材料，并且计划了接下来可能的继续进行的项目的方向。

1.11 不应如此读这本书

不要跳过视频。读者都希望跳过侧边栏、详细阐述和视频，而只是快速浏览文字，直到你找到你想要的问题的答案。

虽然详细阐述部分通常是给那些有经验的想了解更多关于后台如何运作的读者，而侧边栏都是我们认为你会喜欢的短的旁白，而视频则对于学习本书中的内容是**至关重要的**。虽然我们不建议你跳过文字而只看这些视频。但是我们会说它们是这本书中的一些最重要的部分。它们允许我们表达很多概念，显示它们如何交互，并演示如何完成同样的任务。用很多网页还很难描述的内容可以在一个两到五分钟的视频里活灵活现地展示出来。视频让我们遵循孔子的名言："闻之不若见之。"所以请仔细看它们！

积极动手实践。请你打开计算机运行Ruby编译器并且尝试着去解释文中的例子和截图。我们的配套光盘中包含了可以很容易地复制和粘贴的代码。这种做法遵循孔子的理念："知之不若行之。"边干边学在边栏中用"自行车"图标强调表示。

这里有一些主题你需要了解学习，尤其是在敏捷+ Ruby + Rails + SaaS + 云计算的流行语集约生态系统里。事实上，图13.2在后记中列出了仅仅前3章所用的近120个新术语。为了帮助大家识别这些重要的新名词，许多名词用像**这样**的黑体标注出来，这是指你能在维基百科能找到相应的词条（在电子书中，这样的术语都是维基百科相关词条的链接）。

在本书我们以图标来提醒你常见的主题，图1.9总结了所有的图标方便大家查看。 鉴于各人的技术背景情况不同，有时你可能需要对每个章节熟读多次才能掌握

漂亮的代码	遗留代码
惯例优于配置	无重复(DRY)
简洁清晰	需要动手实践
自动化提高生产力	合成代码提高生产力
通过重用提高生产力	通过工具提高生产力
谬论	陷阱
基于ACM/IEEE软件工程课程标准的练习题(ACM 2013)	

图 1.9: 本书中使用的图标小结

它所阐述的精髓。

为了帮助你抓到重点，每章的一开始都有 **概念** 小结，其中列出了每章的重要概念。每章都以 **谬论和陷阱** 结尾，这部分解释了常见的误解或问题，如果你不警惕会很容易陷进去。每节最后有一个章节关键概念的 **小结** 以及 **自我检测问题** 与答案。每章结尾的 **项目** 部分比自我检测题更加开放。 **侧边栏** 突出显示了我们认为的和材料相关的有趣的旁白。

为了给读者一个视角来了解是谁提出了这些他们正在学习的和信息技术依赖的伟大的想法，我们在侧边栏中介绍了20位图灵奖得主。（因为在IT界没有诺贝尔奖，图灵奖众所周知是计算机领域的最高荣誉，即"计算机界的诺贝尔奖"）。

我们故意选择保持本书的简洁性，因为不同读者需要不同领域的额外细节。我们提供了Ruby和Rails在线文档中内置类或者方法的链接，以及你可能不熟悉的重要概念的定义的链接，还有通常需要进一步阅读相关材料的Web链接。如果你正在使用互动版或者Kindle版，连接到互联网时链接应该是可以直接点击的；在打印版本里，链接的URI出现在每章的结尾处。

1.12 谬论和陷阱

这章节会出现在每章结尾，从另外一个角度来回顾本章内容，给读者一个从其他人的错误中学习的机会。 **谬论** 是一些基于本章的思想看起来似乎是正确的语句（或者实际上是普遍持有的观点），但它们并不是正确的。另一方面， **陷阱** 是和这章的主题有关的常见危险，甚至当你被警告时也很难避免。

谬论：敏捷生命周期对软件开发来说是最好的。

敏捷可以很好地匹配某些类型的软件特别是SaaS，这就是为什么我们在这本书里使用它的原因。然而，敏捷 **并非** 对所有的软件来说都是最好的。例如，敏捷就并不适用于事关安全性的应用。

我们的经验是，一旦你了解敏捷软件开发的经典步骤和使用，拥有相关的使用经验，那么不论使用哪种方法，你都将会在其他的项目里使用这些重要的软件工程规则。

敏捷也不会是最后一种你能见到的软件生命周期。我们相信，在新的机会下会有新的方法发展和流行，所以期待在未来学习新的方法和框架吧。

陷阱：忽视软件设计的成本。

由于没有软件制造成本，所以诱导我们相信几乎不需要成本来改变它，从而使得它可以以客户想要的方式再生。然而，这个观点忽略了设计和测试的成本，它们可以占软件项目总成本的很大一部分。零制造成本也是盗版者经常用来诡辩他们复制软件或其它电子数据的合理性的论据，但事实上，盗版者的错误在于认为不需要支付开发成本而仅仅需要支付制造成本。

1.13 结束语：软件工程不只是单纯的编程

> 但是如果极限编程（Extreme Programming）仅仅是旧药换新瓶，我们究竟极限了什么？Kent告诉了我们答案，达到极限水平是需要明显的常识规则和实践的，例如：
> ——如果短迭代效果不错，那就让迭代越短越好——以小时、分钟或秒为周期，而不是天或周或年。
> ——如果简单是好的，那么总是做最简单可行的事。
> ——如果测试结果是正确的，那就总是测试。在编写代码进行测试之前要写好测试代码。
> ——如果代码检查不错，那就连续地检查代码，通过结对编程，两个编程者使用一台计算机，轮流看对方的代码。
>
> ——Michael Swaine采访Kent Beck，（Swaine 2001）

你可以使用计划和文档的方法开发SaaS应用，用敏捷开发封装应用，或者在任何编程框架下进行编写。虽然SaaS，敏捷还有高生产力的框架例如Rails之间没有没

图 1.10: SaaS是基于（1）SaaS云计算、（2）敏捷开发、（3）高生产力的框架和工具三足鼎立的金三角，他们都是皇冠上的宝石

有固有的依存关系，但是图1.10表明它们之间存在协同关系。敏捷开发意味着当与客户密切合作来不断进步，而云计算上的SaaS可以使客户能够立即使用最新版本，从而规避了反馈环路（详见第7章）。云计算上的SaaS用的是模型——视图——控制器设计模式，高生产力的SaaS框架显示了这一点（详见第2章）。旨在支持敏捷开发的高生产力的框架和工具，为练习敏捷扫除障碍（详见第8章和第10章）。我们相信这三个“皇冠上的宝石”能够形成“良性三角”，生成一个完善的软件即服务，他们构成了这本书的基础。

这种良性三角形有助于解释Rails社区的创新本质，就是经常开发出易用新工具从而进一步提高生产效率。我们完全有理由相信，这本书的未来版本会包括尚未发明的工具，它们是如此地有帮助以至于我们无法想象没有它们我们如何完成我们的工作！

作为教师，因为许多学生觉得计划文档方法枯燥无味，令我们十分开心的是图1.7中的10个问题所给出的答案强烈显示了大家应该使用敏捷去实施学生团队项目。不过，我们相信读者去熟悉计划文档方法是值得的，因为有的任务还是很适合用这种方法并且某些公司所有的任务都是遵循这个方法的，所以在第二部分中所有章的结尾部分都会从计划文档的角度看问题。

作为研究人员，我们相信，未来的软件将越来越多地建立在云服务的基础上并依靠云服务，因此由于和云服务之间的协同作用敏捷方法将更广泛地普及。我们正处在一个技术的幸福点，其中未来的软件开发的学习和教学会更有趣。我们写这本书的主要原因是为了帮助更多的人意识到并利用这个难得的机会。

我们相信，如果熟读这本书的内容并使用本书配套软件，你就可以建立你自己的一个受欢迎的软件服务（简单版的），像FarmVille或者Twitter，只要认真学习并遵循软件工程的步骤。尽管能够模仿目前成功的服务，并在几个月内将其部署在云端会令人印象深刻，但是如果 **你** 能发明一些新的技能就更加令人兴奋。我们期待你的漂亮代码变得持久，并成为这些代码的最热情的粉丝！

1.14 深入学习

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

C. Alexander, S. Ishikawa, and M. Silverstein. **A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental)** . Oxford University Press, 1977. ISBN 0195019199.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. **Communications of the ACM (CACM)** , 53(4):50–58, Apr. 2010.

L. A. Barroso and U. Hoelzle. **The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (Synthesis Lectures on Computer Architecture)** . Morgan and Claypool Publishers, 2009. ISBN 159829556X. URL `http://www.morganclaypool.com/doi/pdf/10.2200/S00193ED1V01Y200905CAC006`.

B. W. Boehm. Software engineering: R & D trends and defense needs. In P. Wegner, editor, **Research Directions in Software Technology** , Cambridge, MA, 1979. MIT Press.

B. W. Boehm. A spiral model of software development and enhancement. In **ACM SIGSOFT Software Engineering Notes** , 1986.

E. Braude. **Software Engineering: An Object-Oriented Perspective** . John Wiley and Sons, 2001. ISBN 0471692085.

R. Charettte. Why software fails. **IEEE Spectrum** , 42(9):42–49, September 2005.

M. Cormick. Programming extremism. **Communications of the ACM** , 44(6):109–110, June 2001.

H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. Agile vs. structured distributed software development: A case study. In **Proceedings of the 7th International Conference on Global Software Engineering (ICGSE'12))** , pages 11–20, 2012.

M. Fowler. The New Methodology. **martinfowler.com** , 2005. URL `http://www.martinfowler.com/articles/newMethodology.html`.

H. Goldstein. Who killed the virtual case file? **IEEE Spectrum** , 42(9), September 2005.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 1995. URL `http://blog.standishgroup.com/`.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 2009. URL `http://blog.standishgroup.com/`.

C. Jones. Software project management practices: Failure versus success. **CrossTalk: The Journal of Defense Software Engineering** , pages 5–9, Oct. 2004. URL `http://cross5talk2.squarespace.com/storage/issue-archives/2004/200410/200410-Jones.pdf`.

J. M. Juran and F. M. Gryna. **Juran's quality control handbook** . New York: McGraw-Hill, 1998.

P. Kruchten. **The Rational Unified Process: An Introduction, Third Edition** . Addison-Wesley Professional, 2003. ISBN 0321197704.

T. Lethbridge and R. Laganiere. **Object-Oriented Software Engineering: Practical Software Development using UML and Java** . McGraw-Hill, 2002. ISBN 0072834951.

P. Naur and B. Randell. **Software engineering** . Scientific Affairs Div., NATO, 1969.

J. R. Nawrocki, B. Walter, and A. Wojciechowski. Comparison of CMM level 2 and extreme programming. In **7th European Conference on Software Quality** , Helsinki, Finland, 2002.

M. Paulk, C. Weber, B. Curtis, and M. B. Chrissis. **The Capability Maturity Model: Guidelines for Improving the Software Process** . Addison-Wesley, 1995. ISBN 0201546647.

M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. **Computer** , 38(5):39–47, 2005.

W. W. Royce. Managing the development of large software systems: concepts and techniques. In **Proceedings of WESCON** , pages 1–9, Los Angeles, California, August 1970.

I. Sommerville. **Software Engineering, Ninth Edition** . Addison-Wesley, 2010. ISBN 0137035152.

M. Stephens and D. Rosenberg. **Extreme Programming Refactored: The Case Against XP** . Apress, 2003.

M. Swaine. Back to the future: Was Bill Gates a good programmer? What does Prolog have to do with the semantic web? And what did Kent Beck have for lunch? **Dr. Dobb's The World of Software Development** , 2001. URL `http://www.drdobbs.com/back-to-the-future/184404733`.

A. Taylor. IT projects sink or swim. **BCS Review** , Jan. 2000. URL `http://archive.bcs.org/bulletin/jan00/article1.htm`.

注释

[1]http://www.xuetangX.com
[2]http://www.saas-class.org
[3]http://www.saas-class.org
[4]http://beta.saasbook.info/
[5]http://www.saas-class.org
[6]http://radlab.cs.berkeley.edu
[7]http://github.com/armandofox/latex2ebook
[8]http://beta.saasbook.info/errata
[9]http://en.wikipedia.org/wiki/Turing_Award
[10]http://www.youtube.com/watch?v=kYUrqdUyEpI
[11]https://www.edx.org/
[12]http://www.xuetangX.com
[13]https://plus.google.com/112678702228711889851/posts/eVeouesvaVX
[14]http://developers.slashdot.org/story/08/05/11/1759213/
[15]http://www.youtube.com/watch?v=DeBi2ZxUZiM

1.15 建议的项目

练习1.1. （讨论）在你看来，要给软件灾难从头到尾排个顺序的话，你会怎么排?

练习1.2. （讨论）说出软件进化的主要事件并且解释这些事件对生命周期的影响。注：我们用图标来标识所有的项目都是来自*ACM/ IEEE2013*计算机科学课程为软件工程标准（*ACM 2013*）。

练习1.3. （讨论）讨论一下在一个变化的系统并且开发环境也在变化时我们面临的挑战。

练习1.4. （讨论）解释一个软件生命周期的概念并提供了一个例子，解释它的各个阶段，包括生产交付。

练习1.5. （讨论）根据图*1.7*比较本章所讲到的模型并且评价它们在软件系统的发展中如下几个方面的价值：信息管理、嵌入式、过程控制、通信和网络应用程序。

练习1.6. （讨论）第一部分所提到最近的事件：硬件错误导致软件灾难*Intel*浮点除法错误[1]你会把这个硬件错误放到软件灾难排名的什么地方？

练习1.7. （讨论）如果从代码行来考量，世界上最大的程序是什么？假设这个程序是一个软件的套件并且是一个单一的产品。

练习1.8. *(*讨论*)*大多数程序员最拥护哪种编程语言？

练习1.9. *(*讨论*)*哪种编程语言每年编写的代码行数量是最多的？哪种拥有最多源代码行累计？

练习1.10. （讨论）你给世界上最伟大的应用程序排个前*10*名吧！它们都是在开发和维护使用四个生命周期吗？列出你的理由。

练习1.11. （讨论）上题你已经选出了你心目中的*10*名最伟大的应用程序，那么你给这章中提到的四个生产力技术也排个序吧。

练习1.12. 程序确认（*program validation*）与程序验证（*program verification*）的区别是什么？

练习1.13. （讨论）上题你已经选出了你心目中的*10*名最伟大的应用程序，有哪些程序不便于测试并且需要依靠正规的方法才行？有哪些程序测试技术异常重要？说出你的观点。

练习1.14. （讨论）*SaaS*跟云计算在发展的过程中五个优势跟五个障碍是什么？

练习1.15. （讨论）软件重用的优点和缺点。

练习1.16. （讨论）描述和区分不同类型不同层次的测试（单元、集成、模块、系统和验收）。

练习1.17. 请描述瀑布模型与使用迭代计划文档模型之间的差异，并且解释其原理。

练习1.18. 解释敏捷以及各种计划文档模型关键部件有何不同？

练习1.19. 请你区分不同阶段的软件开发所用的计划文档模型。

第I部分

软件即服务

2 SaaS应用的架构

Dennis Ritchie（左边，**1941——2011**）和**Ken Thompson**（右边，**1943——**）因为在操作系统设计和Unix的发明方面的重大贡献，共同荣获1983年的图灵奖。

我认为Unix的伟大之处在于其简洁的界面：打开、关闭、读取、写入。

—— **Unix与突破：一次对Ken Thompson的采访**，IEEE 计算机32（5），1999年5月

2.1 100000英尺：客户端——服务器架构 36
2.2 50000英尺：通信——HTTP和URI 38
2.3 10000英尺：表示方法——HTML和CSS 41
2.4 5000英尺：3层体系结构和横向扩展 44
2.5 1000 英尺：模型——视图——控制器架构 47
2.6 500英尺：模型的活动记录 48
2.7 500英尺：路由、控制器与REST 51
2.8 500英尺：模板视图 54
2.9 谬论和陷阱 56
2.10 结束语：模式、架构和长寿命的API 57
2.11 深入学习 58
2.12 推荐作业 58

重要概念

软件架构 介绍了分系统是如何组成软件并连接在一起，以满足应用的功能性和非功能性需求。 **设计模式** 描述了一个解决一系列类似问题的通用架构方案。该方案是根据开发人员以往解决这些问题的经验小结出来的。仔细观察一个SaaS应用，你会发现在所有的细节层次其设计模式都非常清晰：

- SaaS应用遵循 **客户端——服务器** 模式，即客户端发出请求，服务器响应多个客户的请求。
- SaaS服务器遵循 **三层架构** 模式，不同的SaaS服务器组件各司其职，并可以通过 **水平扩展（horizontal scaling）** 容纳数百万的用户。
- SaaS应用代码依附在 **应用层** 上，并遵循 **模型-视图-控制器** 模式，在这种模式下，模型可以处理应用程序的 **资源** ，例如用户邮件或博客文章，视图通过浏览器将信息呈现给用户，而控制器将用户对浏览器的操作映射到应用程序相应的代码。
- 在模型方面，Rails采用了 **Active Record模式** ，因为它非常适合 **关系数据库** 。这是SaaS最流行的一种数据存储方式。在视图方面，Rails使用 **模板视图模式** 创建网页然后发送到浏览器。在控制器方面，Rails遵循 **代表性状态传输** 或者REST原则，即每个控制器操作描述了一个应用程序 **资源** 的自包含的单一操作。

Rails等现代化的SaaS框架汇聚了程序员十年的开发经验，通过对这些SaaS设计模式的封装，使SaaS应用编写者可以轻松地应用它们。

1、Web 客户端（火狐）从Web服务器请求Rotten Potatoes主页（WEBrick）。

2、WEBrick从Rotten Potatoes 应用程序获取内容并且把这些内容传输到火狐客户端中。

3、火狐浏览器显示内容并且关闭HTTP连接。

图 2.1: SaaS客户端——服务器系统的一个100000英尺视图

2.1 100000英尺：客户端——服务器架构

既然动手是理解软件的最好方式，我们现在就开始编代码吧！如果你还没有编写过代码，请翻到附录A，然后在计算机或者云服务器中运行“bookware”代码。准备好后，视频2.1.1会向你演示如何部署和登录到虚拟机，尝试与简单的教学应用程序Rotten Potatoes进行交互。Rotten Potatoes的目标是成为最流行的电影评论网站RottenTomatoes[2]的精简版。

视频2.1.1: 开始使用Rails.
光盘文件：ch_arch/mov/getting_started.mp4
登录到虚拟机，视频将演示如何打开一个终端窗口，`cd`（切换到）目录`Documents/Rotten Potatoes`目录，然后输入`rails server`启动Rotten Potatoes程序，接着打开Firefox浏览器，在地址栏中输入`http://localhost:3000/movies`，按下回车进入Rotten Potatoes首页。

这是怎么回事呢？ 你刚才看到的是最简单的Web应用程序视图：这是 **客户端——服务器架构** 的一个范例。Firefox是客户端的一范例：该程序的功能是要求服务器提供信息，（通常情况下）允许用户通过这一信息进行交互。输入`rails server`即可激活WEBrick。通过`rails server`启动的WEBrick是服务器的一个范例：该程序的功能是等待客户端提出 **请求** ，然后提供 **应答** 。WEBrick等待像Firefox这样的浏览器与之联系，然后将浏览器的请求按路线发送至Rotten Potatoes应用程序。图2.1概述了SaaS应用是如何在10万英尺远的地方运行的。

通过服务器端识别出各种客户端，各种类型的程序可以完成高度专业化的任务：客户端提供一个快速响应和有吸引力的用户界面，而服务器端则专注于同时有效地服务多个客户端。Firefox和其他浏览器（Chrome浏览器、Safari浏览器、IE浏览器）都属于拥有数百万用户的客户端（姑且称之为 **生产客户** ）。从另一方面来说，WEBrick算不上是一个生产型服务器，但可算做一个功能齐全的“微型服务器”，它一次允许一个用户（你，开发人员）与你的Web应用程序进行交互。一个真实的Web站点将使用诸如Apache web服务器[3]或者微软因特网信息服务器[4]的生产型服务器，其中任何一种服务器都可以部署在数百台计算机上，为访问同一站点数百万的用户提供有效的服务。

在1990年提出Web开放标准前，用户需要为各种因特网服务单独安装互不兼容的专属客户端：Eudora（雷鸟前身）可用于阅读邮件，AOL和CompuServe用于访问专属内容的门户网站作用类似今天的MSN和Yahoo!等等。如今，Web浏览器在很大程度上已经取代了专属客户端，名正言顺地成为“通用客户端”。尽管如此，专属客户端和私人服务器仍然可以组建客户端——服务器架构，客户端专门负责代表用户提出问

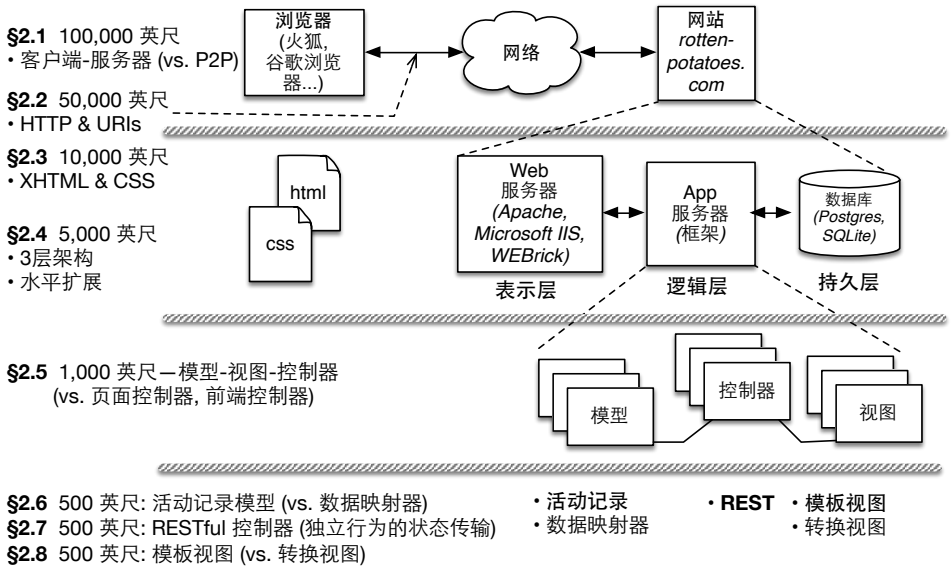


图 2.2: 通过高度单位作为类比，此图说明了SaaS结构各个细节层的重要结构。本章内容将围绕这个整体路线图展开。下述每个章节将每个层次进行逐一讨论

题，服务端专门负责回答客户端的问题。因此，客户端——服务器是我们 **设计模式** 的第一个例子——一个可重用结构，其行为、策略和技术，通过将 **变化的事物和一成不变的事物** 区分开来，对一系列类似问题提供了一个可靠的解决方案。对于客户端——服务器架构，一成不变的是客户端和服务器的关注点相分离。然而客户端和服务器的实现却是经常变化的。由于网络的普及，我们将使用SaaS这个术语代表“使用万维网开放标准操作客户端——服务器系统”。

在过去，客户端——服务器体系意味着服务器是一个比客户端更复杂的程序。在功能强大的笔记本电脑和可以支持动画和3D效果的Web浏览器的今天，一个更好的特征是，客户端和服务器拥有同样的复杂度，只是专门用于不同的途径罢了。在本书中，我们将专注于服务器为中心的应用程序；在第6章中还会涉及JavaScript客户端编程，其环境可支持以服务器为中心的应用程序，而非浏览器中构建Google Docs[5]等复杂的应用程序。

当然，客户端——服务器不是唯一基于互联网服务的架构模式。例如在BitTorrent中使用的 **点对点架构** ，每个参与者既是客户端又是服务器——任何人都可以访问其他人的信息。在这样的系统中，同一程序必须既做客户端又当服务器，这样专门的程序很难做到客户端和服务器两手都抓，两手都硬。

小结：

- SaaS的Web应用程序是一个 **客户端——服务器架构模式** 的例子，客户端软件通常专门与用户交互并且以用户的身份将请求发送到服务器，服务器软件用来处理大量这样的请求。
- 由于Web应用程序使用的是开放的标准，对任何人来说都是免版税的，更早的客户端——服务器应用程序需要专有的标准，所以Web浏览器已经成为了“通用的客户端”。
- 客户端——服务器的一种替代品是点对点的标准（P2P），所有的计算机既可以作为客户端，也可以作为服务器。P2P灵活是相对的，它的这种结构使得它很难为把服务器/客户端中任一角色扮演好。

自我检查2.1.1. *SaaS*的客户端与服务器主要区别是?
◇ SaaS的客户端旨在允许用户与信息进行交互，而SaaS的服务器旨在同时服务于多个客户端。

自我检查2.1.2. 图*2.2*中哪个元素指的是*SaaS*客户端，哪个元素指的是*SaaS*服务器?
◇ 在框图中左上角**浏览器**指一个客户端。**html**和**css**文档图标是指传递给客户端的内容。所有其他元素都是服务器的一部分。

2.2 50000英尺：通信——HTTP和URI

Vinton E. “Vint” Cerf（左边，**1943——**）和**Bob Kahn**（右边，**1938——**）因为在网络架构和协议（包括TCP/IP）方面的开创性工作，共同荣获了2004年图灵奖。

网络协议 是一组代理如何参与网络协议的通信规则。在这种情况下，它的代理是Web客户端（Firefox）和Web服务器（WEBrick或Apache）。浏览器和Web服务器通过 **超文本传输协议**（简称 **HTTP**）进行通信。跟许多互联网应用协议一样，HTTP的基础是 **TCP/IP**，意义深远的 **传输控制协议/因特网协议（Transmission Control Protocol/Internet Protocol）**，它允许一对代理可以进行有序序列的通信。从本质上讲，TCP/IP协议允许一对网络代理进行任意字符串的通信。

在TCP/IP网络中，每台计算机都有一个 **IP地址**。它由四个圆点分隔的字节组成，例如128.32.244.172。我们通常不会直接使用IP地址，而会使用另一个名为 **域名系统**（**DNS**）的网络服务。它在TCP/IP协议上有自己的协议，会自动调用简单易记的 **主机名**，例如`www.eecs.berkeley.edu`，映射到IP地址。浏览器会自动联系DNS服务器来查找所输入地址栏的名称，例如`www.eecs.berkeley.edu`，然后得到实际的IP地址（在这里是128.32.244.172）。TCP/IP兼容机使用了一个约定，如果计算机上运行的一个程序引用了主机（localhost）名称，那么它就会指向运行该程序的那台计算机。因此，在2.1节开头，只要在Firefox的地址栏中输入主机（`localhost`），Firefox就可以与在自身运行的同一台计算机上的WEBrick程序进行通信。

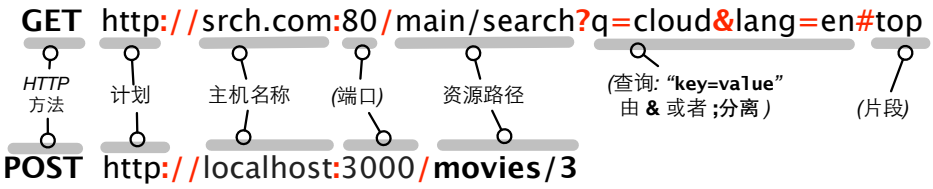


图 2.3: HTTP请求包括一个 HTTP方法 加上一个 URI 。 完整的URI 是以http或者https起始并且包括上述组件。在括号中的是可忽略组件。 局部URI 会忽略最左侧的部分或者全部元素，在这种情况下有些组件是在具体的应用程序的决策下进行填写或者 解决基础的URI 。最佳做法是使用完整的URI

■ **详细阐述：网络：多宿主，IPv6和HTTPS.**

我们从几个方面简化了TCP/IP协议：在技术上，每个 **网络接口设备** 都有一个IP地址，而 **多宿主** 计算机可能有多个网络接口。此外，由于种种原因，比如IP号码的地址空间几近枯竭，当前版本的IP（IPV4）正在逐渐被IPv6淘汰，IPV6使用了不同的地址格式。然而，由于大多数计算机在某一时段只能激活一个网络接口，并且SaaS应用员很少直接与IP地址打交道，所以这些简化并不会从本质上改变我们的解答。只有到第12章我们才会讨论安全HTTP协议（HTTPS）。在HTTP客户端和服务器之间通信时，HTTPS会使用 **公钥加密** 进行加密（编码），因而窃听者只能看到乱码。从程序员的角度来看，HTTPS的行为类似于HTTP，只有在Web服务器配置了支持HTTPS访问特定网页时，它才会起作用。WEBrick等迷你型服务器通常不支持HTTPS。

例子中localhost之后的`:3000`是什么意思呢？网络上的多个代理可以同时在同一个IP地址上运行。事实上，在上述例子中，客户端和服务器都在本机运行。因此TCP/IP使用从1~65535的 **端口号** 来区分位于同一个IP地址的不同的网络代理。在打开一个连接前，所有基于TCP/IP的协议，包括HTTP协议，都必须指定主机和端口号。要求Firefox连接到`localhost:3000/movies`时，默认在本地主机（即"这台计算机"），一个服务器程序正在监听端口3000并等待浏览器链接，如果我们没有明确指定端口号（3000），它会默认通过80`http`或443`https`（安全）获取连接。

IANA。 互联网编号分配机构[6]负责为各种协议分配官方默认的端口号，并管理DNS的顶层或"根"区。

总而言之，当代理通过指定主机名和端口号 **打开至另一个代理的连接时**，HTTP中的通讯会被初始化；HTTP服务器进程必须在该主机名和端口号实施 **连接侦听** 。

在Firefox的地址栏输入的`http://localhost:3000/movies`字符串就是一个 **URI**，或者称为 **统一资源标识符** 。URI以通信 **方案** 名称开头，以便检索该信息，接着是主机名、可选端口号、以及用户希望对该主机上检索的 **资源** 。资源通常是指"任何可以交给浏览器的东西"。一张图片、所有电影的HTML格式列表以及一份创建新电影的提交表格都属于资源。每个SaaS应用都有自己的资源名称解析规则。但我们推荐使用REST，对于不同的SaaS应用中的资源命名，它竭力追求简单性与一致性。

URI还是URL? URI有时指的是URL或者统一资源定位器。尽管两者存在细微的技术差别，对于我们来说，这两个词是可以互换的。我们使用的是URI，因为它的普及度更高，而且大多数库都使用的这个术语。

HTTP是一个 **无状态协议** 。因为每个HTTP请求都与所有先前请求无关。如果一种网页应用程序要一直跟踪"你在哪里"（如你登录了吗？你结账到哪一步了？）。由于HTTP请求记不住有关的信息，该网页程序必须有自己的机制来实现这些功能。HTTP的 **cookies** 会通过服务器中与该用户的 **会话** 对应的信息来关联特定用户的浏览器。确保每个HTTP请求包含正确的cookies是浏览器的责任，而不是HTTP或SaaS应用。因此，无状态协议简化了服务器的设计，但应用程序设计要为

1、Web 客户端（火狐）从Web服务器请求Rotten Potatoes主页（WEBrick）。
a、火狐客户端构造一个HTTP请求使用URI ***http://localhost:3000*** 连接到HTTP 服务器（WEBrick）在同一个计算机的火狐客户端监听端口3000 （**本地服务器**）。
b、WEBrick, 监听端口3000，接收HTTP 请求'/movies'资源（Rotten Potatoes中列出的所有电影）。

2、WEBrick从Rotten Potatoes 应用程序获取内容并且把这些内容传输到火狐客户端中.

3、火狐浏览器显示内容并且关闭HTTP连接

图 2.4: 在50000英尺，我们可以扩展图2.1中的第一步

之付出代价，好在Rails等成功的框架会为你屏蔽掉这些复杂性。

视频2.2.1: Cookies.
光盘文件：ch_arch/mov/cookies.mp4
Cookies用于确立同一用户浏览器的两个独立的请求，因此可认为是会话的一部分。首次访问某个网站时，服务器包含一个带有`Set-Cookie:`HTTP响应报头的长字符串（可达4K）。在后续请求该站点时，浏览器将负责纳入带有`Set-Cookie:`HTTP请求报头。Cookie字符串的目的是对用户不透明，其中包含服务器使用相同用户会话关联该请求的所有信息。 **（注：该Beta版视频中使用的RottenPotatoes版本与其他例子的版本略有不同，但这并不会对我们的视频中的说明造成实质性影响）。**

如图2.4所示，我们现在会用稍微精确的术语来描述在加载Rotten Potatoes主页时发生了什么。我们接下来会更深入地研究内容本身是如何被表示的。

■ **详细阐述：客户端下载与服务器推送。**

Web主要上是一个 **客户端提取** 的客户端——服务器架构，因为客户端启动了所有的交互——HTTP服务器只能等待客户与之联系。这是因为，HTTP的设计符合 **请求-应答协议**：只有客户端才能启动一切。不断演进的标准（包括WebSockets和HTML5）支持服务器将更新内容 **推送** 至客户端。相反，真正的 **服务器推送** 架构，例如在手机上发短信，允许服务器向客户端发起连接，以便在有新信息时“唤醒它”；但HTTP不支持这些功能。人们早期对Web架构的批判是，纯粹的请求-响应协议将无法支持 **基于推送** 的应用。然而，在实践中，高效率专业化的服务器软件可支持那些通过频繁与服务器校验来进行更新的网页。即便没有WebSockets和HTML5提出的特性，用户也会产生推送式应用程序的错觉。

小结：

- Web浏览器和服务器之间使用 **超文本传输协议** 进行通信。HTTP在 **TCP/IP**（传输控制协议/Internet协议）的基础上可靠地交换有序字节序列。
- 每台与TCP/IP网络相连的计算机都有一个 **IP地址**，例如128.32.244.172，而 **域名系统（DNS）** 允许使用人性化的域名来代替IP地址。特定名称的本地主机是指本地计算机，并可以解析成特定的IP地址127.0.0.1。
- 在特定计算机上运行的每个应用程序必须"侦听"一个特定的 **TCP端口**，端口编号为1∼65535（$2^{16}-1$）。端口80用于HTTP（Web）服务器。
- 要在本地运行SaaS应用，你可以激活一个HTTP服务器监听本地主机的端口。Rails的轻量级服务器WEBrick使用的是3000端口。
- **统一资源标识符（URI）** 为互联网上提供的资源命名。对资源名称的解析随着应用程序的不用而变化。
- HTTP是一个无状态协议，每个请求独立于其他请求，甚至独立于同一用户。 **HTTP cookies** 可以关联来自同一用户的HTTP请求。浏览器将负责接收来自HTTP服务器的Cookie，并确保接下来发送到该服务器上的请求包含这个cookie。

自我检查2.2.1. 如果访问*URI* `http://google.com:3000`，会发生什么情况呢？原因何在？
◇ 该连接将最终会"超时"，并且无法连接服务器，因为谷歌（几乎跟所有网站一样）会侦听TCP 80端口（默认值）而非3000端口。

自我检查2.2.2. 如果尝试用`http://localhost:3300`而不用`:3000`访问*Rotten Potatoes*，会发生什么情况？原因何在？
◇ 由于3300端口没有被监听，你的"连接被拒绝"。

2.3 10000英尺：表示方法——HTML和CSS

如果Web浏览器是通用的客户端，超文本标记语言 **HTML** 就是通用语言。 **标记语言** 会把文本和标记结合（有关文本注释）起来，以便在语义上区分两者。请看2.3.1视频中当前语言版本HTML 5中的重点介绍，然后继续往下看。

视频2.3.1: HTML介绍.
光盘文件：ch_arch/mov/html-intro.mp4

HTML包含不同层次的嵌套元素，每层由一个起始标记（例如`<p>`）、内容部分（在某些情况下）以及结束标记（例如`</p>`）组成。大多数起始标记也可以拥有属性，例如`<a href="http://...">`。无内容部分的标签是自我封闭的，例如`<br clear="both"/>`，表示换行符并清除了左右边距。

尖括号标记的使用来自于 **SGML**（标准通用标记语言）。这是IBM通用标记语言的一种编码标准。1960年，为了编写计算机可读的项目文件，IBM开发了这种标记语言。

不幸的是，HTML[7]有很多混淆的术语。HTML 5包括其前身（HTML第一版至第四版）和 **XML** 的子集XHTML（可扩展超文本标记语言）的特性。XML用来表

示数据和描述其他扩展性标记语言。诚然，XML是一种常见的数据表达方式，用于面向服务器的架构 **之间** 的信息交流，当将Rotten Potatoes扩展至从单独的电影数据库服务中检索电影信息时，我们就会在第8章看到这一点。很难直白地解释清楚XHTML和HTML的变体之间的差异，并非所有的浏览器都支持各个版本。从现在起，除非另有说明，我们所说的HTML都是指HTML 5，我们会尽量避免使用未得到广泛支持的特性。

特别有趣的是HTML标签属性`id`和`class`，因为它们使页面的HTML结构和它的视觉形象连接到一起。接下来的视频展示了使用Firefox的Web开发者工具栏来快速识别页面上的ID和HTML元素的类。

视频2.3.2: 检查id和class属性.
光盘文件：ch_arch/mov/html-class-id-attributes.mp4
CSS使用 **选择器号** ，例如`div#` **name** 表示一个id是 **name** 的`div`元素，而`div.` **name** 表示一个类为class的`div`元素。在HTML文档中只有一个元素可以有一个给定的`id`，而许多元素（甚至不同的标签类型）则共享同一个类。一个元素包含三个方面——标签类型、`id`（如果有的话），类属性（如果有的话）——在形象化表示中它们用来唯一确定一个候选元素。

要了解CSS能发挥多大作用的极限例子，请访问CSS Zen Garden[8].

如下一个视频显示，通过使用元素的类和ID， **CSS** （**级联样式表，Cascading Style Sheet** ）标准，可以关联视觉“风格”和HTML元素。视频中只有几个基本的CSS结构，这在图2.5中有概述。本章结尾的资料部分十分详细地列举了描述CSS的网站和书籍，包括如何使用CSS校准网页内容的方法，和设计师手工操作表格使用的一些东西。

视频2.3.3: CSS介绍.
光盘文件：ch_arch/mov/css-intro.mp4
对了和HTML元素的匹配，CSS文件中的选择器有四大机制：标签名称、类别、编号和层次。如果多个选择器同时匹配给定元素，其属性应用规则就会变得很复杂。因此，为了保持CSS的简洁性，大多数设计师会尽量避免这类模棱两可的东西。要看到页面的“骨架”，有一个非常有用的方法，即从FirefoxWeb开发工具栏选择 **CSS>禁用样式>所有样式** 。这会关闭所有CSS格式设置并显示该页面，可以看到CSS可以用来将视觉外观从逻辑结构中分离的程度。

利用这一新信息，根据之前章节中对SaaS工作方法的小结，图2.6对步骤2和3进行了解释。

选择	选了什么
h1	任何h1 元素
div#message	div 的ID 是message
.red	任何带有类red的元素
div.red, h1	带有red类的div，或者任何h1
div#message h1	元素h1 是div#message（内部）的孩子
a.lnk	带有类lnk的元素a
a.lnk:hover	当有hover的时候，元素a带有类lnk
属性	**值**
字体类型	"Times, serif"
字体粗细	bold
字体大小	14pt, 125%, 12px
字体样式	italic
颜色	black
边缘	4px
背景色	red, #c2eed6 (RGB values)
边界	1px solid blue
水平对齐	right
文本修饰	underline
垂直对齐	middle
填充	1cm

图 2.5: 几个CSS结构，包括视频2.3.3中解释的内容。上表显示了一些CSS 选择器，它可以识别特定元素；下表显示了众多属性的一部分，这些名称通常都是不言自明的，它们还可以指定示例值。并非所有属性对所有元素都有效

1.、Web 客户端 （火狐） 从Web服务器请求Rotten Potatoes主页（WEBrick）。
a、 火狐客户端构造一个HTTP请求使用URI ***http://localhost:3000*** 连接到HTTP 服务器（WEBrick）在同一个计算机的火狐客户端监听端口3000 （本地服务器）。
b)、WEBrick, 监听端口3000,接收HTTP 请求'/movies'资源（Rotten Potatoes中列出的所有电影）。

2、WEBrick从Rotten Potatoes 应用程序获取内容并且把这些内容传输到火狐客户端中
a、通过Rack的中间件 （由Ruby编写）, WEBrick在应用层调用Rotten Potatoes代码。这些代码用存储在由当地的SQLite数据库实现的表示层中的电影信息生成页面内容。

3、火狐浏览器显示内容并且关闭HTTP连接
a、火狐浏览器通过重复之前的四个步骤来获取有用的东西（CSS，图片等等）。
b、火狐浏览器根据CSS格式化命令显示页面，包含格式化的信息描述所需的视图属性页面。

图 2.6: 10000英尺的SaaS。相比于图2.4，扩展后的步骤2描述了Web服务器返回的内容，扩展后的步骤3描述了在Web浏览器呈现其内容时CSS所起的作用

小结：

- **HTML**（超文本标记语言）文档包括一系列分层嵌套的元素的集合。每个元素以<尖括号>中的 **标签（tag）** 开头，并且有可选的 **属性** 。某些元素还封装了内容（content）。
- **选择器** 是一个表达式，利用元素名称（例如`body`）、元素`id`（页面上的元素属性必须是唯一的）、元素类（类属性不一定是唯一的）的组合，可以识别文档中的一个或多个HTML元素。
- **级联样式表（Cascading Style Sheet， CSS** ）是一种描述Web页面元素视觉属性的样式语言。将一系列视觉属性和选择器关联起来。HTML文件头元素内的特殊链接元素可以关联样式表和文档。
- FirefoxWeb开发者工具栏提供了宝贵的工具，可以在底层浏览页面结构及其样式表。

自我检查2.3.1. 对或错：每个*HTML*元素必须有一个*ID*。
◇ 错——ID是可选的，但只要提供了ID，就必须具备唯一性。

自我检查2.3.2. 如果有如下的*HTML*标记：

代码文件：*ch_arch/code/htmlexercise.html*

```
<p class="x" id="i">I hate <span>Mondays</span></p>
<p>but <span class="y">Tuesdays</span> are OK.</p>
```

只**能** 选择*Mondays*作为*CSS*的选择器。
◇ 三种可能性，从最具体到最不具体依次是：**#i span**、**p.x span**和**.x span**。其他选择器也有可能，但它们要么冗余，要么约束过度：**p#i span**和**p#i.x span**对于这个HTML片段是冗余的，因为至多只有一个元素的ID可以是 **i**。

自我检查2.3.3. 在自检*2.3.2*中，为什么**span**和**p span 不是** 有效的答案？
◇ 这两个选择器也可以匹配 **Tuesdays** 。因为**span**也在**p**中。

自我检查2.3.4. 对于关联*CSS*样式表和*HTML*或*HTML*文档来说，哪种方式最为常见？
提示：参考之前的视频示例。
◇ HTML或者HTML文档的`HEAD`元素中包括`LINK`元素，它至少有以下三个属性：`REL="STYLESHEET"`、`TYPE="text/css"`和`HREF="` **uri** `"`，其中 **uri** 是样式表的全部或部分URI。也就是说，样式表必须是以URI命名的可访问资源。

2.4 5000英尺：3层体系结构和横向扩展

由于应用服务器位于Web服务器的（表示层）和实际的应用程序代码之间。它们有时被称为 **中间件（middleware）** 。

到目前为止，我们已经了解了客户端与服务器进行通信和信息交换的方式，但我们并没有提及任何服务器自身情况。图2.2中服务器部分的第二个层次上进行扩展，Web应用程序由如下三个逻辑 **层** 组成。 **表示层** 通常包含 **HTTP服务器** （或者简单的“ **Web服务器** ”）接受外界（即用户）的请求，并通常提供静态的数据。我们一直在使用WEBrick履行这一任务。

Web服务器向 **逻辑层** 发送动态内容请求。在逻辑层中，实际的应用程序运行会产生动态内容。该应用程序通常由一个 **应用服务器** 支持。它的工作是对应用程序员

隐藏底层HTTP的机制。例如，应用程序服务器可以将传入的HTTP请求直接发送至对应的应用程序相应的代码片段，从而省去了监听和解析传入HTTP请求的必要。现代应用程序服务器支持一个或多个 **Web应用程序框架** 。它可以通过一种特定的语言来简化Web应用程序特殊类的创建。我们将使用Rails框架以及Rails衍生的Rack应用程序服务器。WEBrick可以直接与Rack“对话”; Apache Web等其他Web服务器则需要额外的软件模块才能进行对话。如果你正在使用PHP、Python或Java，则可以使用处理这些语言的应用服务器。例如，在Python和Java应用程序上运行的谷歌AppEngine拥有一个专属中间件，可以将应用程序中的Python或Java代码与谷歌运营的对外的基础架构连接起来。

LAMP. 早期的SaaS网站是用Perl和PHP脚本语言创建的，其实用性与开源操作系统Linux和开源数据库MySQL的早期成功互为呼应。现在有成千上万的网站仍由 **LAMP 堆栈** （Linux、Apache、MySQL、PHP或Perl）支持。

最后，由于HTTP是无状态的，必须在HTTP请求之间存储数据（例如会话数据、用户登录和个人资料；应用程序数据存储在 **持久层** ）。对于持久层，开源的MySQL或PostgreSQL等传统数据库成为了普遍的选择。在这一选择激增之前，Oracle或IBM DB2等商业数据库也是非常普遍的选择。

三层模型中所谓的“层”指的是 **逻辑** 层。在内容较少和低流量的网站上，三个层的软件可能会在同一台物理计算机上运行。事实上，Rotten Potatoes已经这么做了，它的表现层就是WEBrick，它的持久层是一个名叫SQLite的简单开源数据库。它可以直接把信息存储在本地计算机的文件中。在生成过程中，每层跨越一个或多个物理计算机是比较常见的。如图2.7所示，在一个典型的网站中，HTTP传入请求被指向了众多Web服务器中的一个，轮流从多个可用应用程序服务器中选择其中一个服务器来处理动态内容，允许计算机根据处理需求在每层中进行添加或删除内容。

然而，如谬论和陷阱章节所述，要实现持久层的无共享要复杂得多。图2.7显示了 **主从** 方法。仅在数据库中读取比写入更频繁的时候才使用该方法：任何从机都可以执行读取，只有主机才能执行写入，然后主机使用写入结果尽可能快的新从机。然而，这种技术只是延缓了扩展性问题的出现，而无法从根本上解决这个问题。http://heroku.com的创始人Heroku[9]写道：

> 经常有人问我关于Heroku的问题是：“你是如何扩展SQL数据库的？”我可以说方法很多，比如使用高速缓存、分片和其他技术加载数据库。但实际的答案是：我们没法扩展。SQL数据库基本上是不可扩展的，任何人都没有洒在SQL上就能突然实现扩展的魔法粉末。
>
> ——Adam Wiggins, Heroku[10]

现在我们可以给解释再添加一个细节层次；如图2.8中的步骤2a。

小结：

- 三层体系结构包括一个渲染视图并与用户进行交互的表示层、一个运行SaaS应用的码逻辑层，一个储应用程序数据的持久层。
- 无状态的HTTP允许表示层和逻辑层是 **无共享** 的，所以云计算可以根据每一层的需求增加更多物理计算机。然而持久层是难以扩展的。
- 根据部署的规模（大小），一台计算机可能会托管多层，或者单层就需要多台计算机。

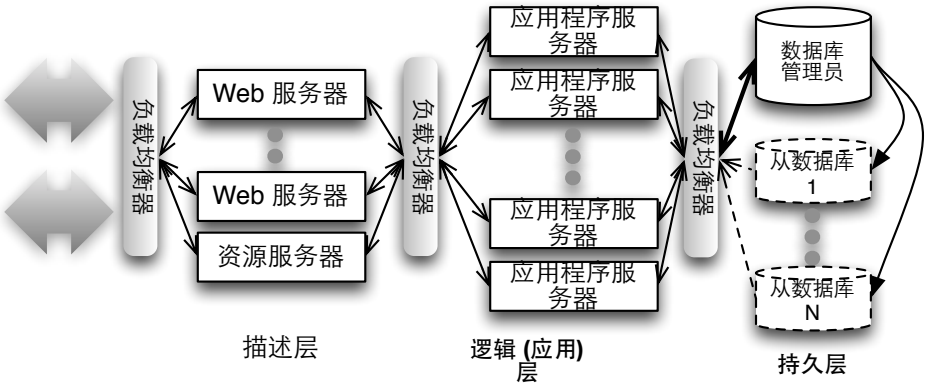


图 2.7: 三层 无共享 结构之所以称为无共享结构，是因为层内实体一般不进行内部沟通，它允许在每层中独立地增加计算机以满足需求。 负载平衡器 可以通过硬件设备或专门配置的Web服务器均匀地分配工作量。无状态的HTTP让无共享成为可能：因为所有请求都是独立的，可以向表示层或逻辑层中的任何服务器分配任何请求。然而，如正文所述，持久层的扩展则更具挑战性

1、Web 客户端（火狐）从Web服务器请求Rotten Potatoes主页（WEBrick）。
  a、火狐客户端构造一个HTTP请求使用URI ***http://localhost:3000*** 连接到HTTP 服务器（WEBrick）在同一个计算机的火狐客户端监听端口3000（**本地服务器**）。
  b、WEBrick，监听端口3000，接收HTTP 请求'/movies'资源（Rotten Potatoes中列出的所有电影）。

2、WEBrick从Rotten Potatoes 应用程序获取内容并且把这些内容传输到火狐客户端中
  a、通过Rack的中间件（由Ruby编写），WEBrick在应用层调用Rotten Potatoes代码。这些代码用存储在由当地的SQLite数据库实现的表示层中的电影信息生成页面内容。
  b、WEBrick会使用HTTP在HTML编码中返回其内容。HTML可能包含像图片嵌入显示页面那样引用其他的媒体。HTML 可能也会包含一个CSS样式表，它包含格式化的信息描述所需的视图属性页面（字体大小，颜色，布局等等）。

3、火狐浏览器显示内容并且关闭HTTP连接
  a、火狐浏览器通过重复之前的四个步骤来获取有用的东西（CSS，图片等等）。
  b、火狐浏览器根据CSS格式化命令显示页面，包含格式化的信息描述所需的视图属性页面。

图 2.8: 5000英尺SaaS。相比图2.6，已经插入了步骤2a，以便根据三层架构描述SaaS服务器的动作

■ **详细阐述：为什么是数据库?**

最早的Web应用程序有时为了存储数据而直接操作文件，为什么绝大多数数据库很早就接管了这个角色呢？这里有两个原因：首先，数据库一直为存储信息提供很高的**耐久性** ——一旦发生系统崩溃或瞬态数据损坏突发事件，也不会造成数据丢失。对于存储着数百万用户数据的Web应用程序来说，这是至关重要的。其次，数据库把信息存储在结构化格式中——例如目前最流行的 **关系数据库** 类型，每个对象类型存储在一个表中，表的每行代表对象实例，每列代表对象属性。有趣的是，如今最大的Web应用程序，例如Facebook已经远远超出了关系数据设计的规模，不得不考虑设计一个处于统治地位的关系数据库的替代品。

自我检查2.4.1. 如果大多数*SaaS*应用没有遵循无共享架构，为什么云计算对*SaaS*的影响会更小呢?

◇ 云计算可以轻松地添加和删除计算机，只需要为你所使用的应用付费。由于是无共享架构，它可以直接将新计算机“吸进”到一个正在运行的应用程序，然后在不需要的时候“释放”。

自我检查2.4.2. 三层*SaaS*应用的____层的扩展要比添加计算机要复杂得多。

◇ 持久层

2.5 1000 英尺：模型——视图——控制器架构

到目前为止，我们还没有提到Rotten Potatoes的应用程序代码结构。事实上，就像我们使用的客户端——服务器架构模式描述“10万英尺视图”的SaaS的特征一样，我们可以用一种称为 **模型-视图—控制器** （通常简写为MVC）的架构模式来描述“1000英尺视图”的特征。

根据MVC模式编写的应用程序主要包含三种代码类型。 **模型** 与应用程序操纵的数据有关：如何存储、如何操作以及如何去改变数据。MVC应用程序一般为应用程序操作的每种实体类型配备了一个模型。精简版的Rotten Potatoes程序中只有Movie模型，但是我们往后可以添加其他模型。因为模型可以处理应用程序数据，它们包含与存储层沟通的代码。

呈现给用户的 **视图** 包含用户用来交互的模型的信息。该视图可作为系统用户及其数据之间的接口；例如，在Rotten Potatoes中可以制作电影列表，然后单击视图链接或按钮添加新电影。Rotten Potatoes只有一种模型，但它与多个视图有关联；一个视图列出所有电影，另一个视图显示特定电影的细节，当创建新电影或者编辑已有电影的时候，会出现其他视图。

最后， **控制器** 解调互动是双向的：当一个用户与视图进行互动（例如通过单击页面上的按钮），一个特殊的控制器对此做出 **行动** 以回应用户的调用行为。每一个控制器回应一个模型，在Rails中控制器行为是由一个特殊的Ruby方法控制器控制的。这个控制器可以要求模型来检索或修改信息，这样做的结果是让控制器来决定展示什么样的视图给用户并且提供该视图的任何必要的信息。由于Rotten Potatoes只有一个模型（电影），它同样也只有一个控制器（电影控制器）。其控制其所定义的行为以处理每种类型的用户交互视图（例如单击链接或者按钮），并包含必要的逻辑来获取模型数据来 **渲染** 电影视图。鉴于SaaS应用一直以视图为中心并且一直依赖于持久层，MVC选择Rails作为底层架构似乎非常合适。但也可以选择其他架构，正如图2.9所示，该图摘自Martin Fowler的 **企业应用架构模式目录** [11]。应用程序主要是静态内容，只有少部分是动态生成的内容，例如天气网站的应用程序有可能选择

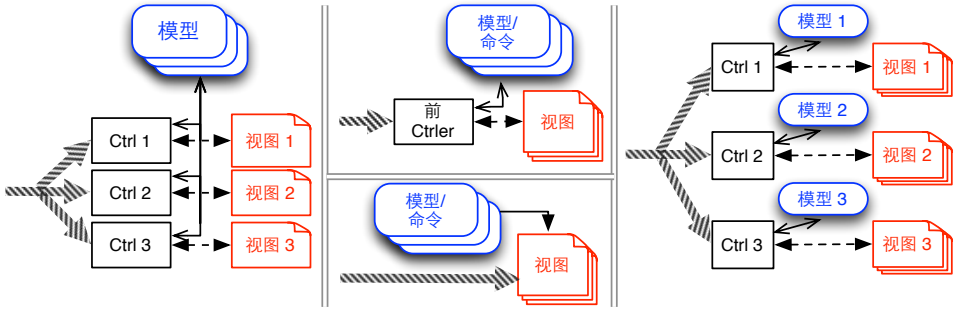


图 2.9: 比较Web应用程序架构的模式。模型用圆角矩形表示，控制器用矩形表示，视图用文档图标表示。Sinatra使用的网页控制器（左）为应用程序的每个逻辑页配备了一个控制器。Java 2企业版（J2EE）servlets（顶部中心）使用的前端控制器有一个控制器。该控制器可以根据多种模型中的方法生成一组视图。PHP使用的模板视图（底部中心）强调通过围绕视图来构建应用程序，模型中的逻辑生成动态的内容会取代部分的视图；该控制器是隐含在框架中。Rails和Java Spring使用的模型-视图-控制器（右）将控制器和一组视图用每个模型类型关联起来

模板视图 模式。 **页面控制器** 模式非常适合只有少量不同页面的应用程序的结构，每个页面都有一个自己简单的控制器，只处理该生成页面本身。对于那些用户要浏览一系列网页（如订阅邮件列表）但是只有少数模型的应用程序， **前端控制器** 模型就已经足够，其中一个控制器处理所有传入的请求，而非分离的控制器处理每个模型的请求。

图2.10小结了我们对SaaS应用结构最新的认识。

小结：

- **模型-视图-控制器** 或MVC设计模式清晰地区分了实现业务逻辑的 **模型** 、将信息传递给用户并允许用户与应用程序交互的 **视图** 以及传递视图和模型之间交互的 **控制器** 。
- 在使用MVC的SaaS应用中，每个用户可以在网页上单击链接或按钮，提交填好的表单，或者进行拖放操作——最终由控制器动作进行处理。控制器动作根据需要询问模型，以获取信息并生成视图来作为回应。
- MVC适合包含各种模型类型的SaaS应用的交互，这使得控制器和视图与每种模型类型的匹配变得很有意义。其他结构模型可能更适合模型类型更少或操作指令表更小的应用程序。

自我检查2.5.1. 在三层架构中，哪些模型参与了下列内容的处理：
*(a)*模型 *(b)*控制器 *(c)*视图？
◇ (a)模型：逻辑层和持久层。
(b)控制器：逻辑层和表示层。
(c)视图：逻辑层和表示层。

2.6 500英尺：模型的活动记录

模型、视图和控制器到底是如何工作的呢？我们将从模式角度来进行深入描

1、Web 客户端（火狐）从Web服务器请求Rotten Potatoes主页（WEBrick）。
  a、火狐客户端构造一个HTTP请求使用URI ***http://localhost:3000*** 连接到HTTP 服务器（WEBrick）在同一个计算机的火狐客户端监听端口3000（**本地服务器**）。
  b、WEBrick， 监听端口3000，接收HTTP 请求'/movies'资源（Rotten Potatoes中列出的所有电影）。

2、WEBrick从Rotten Potatoes 应用程序获取内容并且把这些内容传输到火狐客户端中
  a、通过Rack的中间件（由Ruby编写）， WEBrick在应用层调用Rotten Potatoes代码。这些代码用存储在由当地的SQLite数据库实现的表示层中的电影信息生成页面内容。
    1） Rack 需要电影控制器的**索引**行为；其资源是由所有电影列表路由命名的。
    2）当电影控制器询问电影模型列表以及属性时，Ruby 的函数实现**索引**行为。
    3）如果运行成功，控制器会识别视图，这个视图包含由HTML标记的电影列表，并且传递电影信息以便HTML构造页面。 如果运行失败，控制器识别视图并且显示错误消息。
    4）Rack把控制视图传递给WEBrick，这会像HTTP中回复那样回传信息给火狐浏览器。
  b、WEBrick会使用HTTP在HTML编码中返回其内容。HTML可能包含像图片嵌入显示页面那样引用其他的媒体。HTML 可能也会包含一个CSS样式表，它包含格式化的信息描述所需的视图属性页面（字体大小，颜色，布局等等）。

3、火狐浏览器显示内容并且关闭HTTP连接
  a、火狐浏览器通过重复之前的四个步骤来获取有用的东西（CSS，图片等等）。
  b、火狐浏览器根据CSS格式化命令显示页面，包含格式化的信息描述所需的视图属性页面。

图 2.10: 步骤2a已经扩展到在显示SaaS应用请求MVC架构所扮演的角色

述。

每个重要的应用程序都需要存储和操控持久性数据。无论是使用数据库、普通文件或其他持久性存储单元，都需要一种方法来回转换这些数据结构和对象或应用程序代码操控的对象以及数据的存储方式。在本章使用的Rotten Potatoes版本中，电影的相关信息是唯一的持久性数据。每部电影的 **属性** 包括标题、发布日期，MPAA分级、电影简介。一个简单的方法是在纯文本文件中存储电影信息，每行对应一部电影，并由逗号分隔属性：

代码文件：ch_arch/code/schema1.txt

```
Gone with the Wind,G,1939-12-15,An American classic ...
Casablanca,PG,1942-11-26,Casablanca is a classic and...
```

要获取电影信息，我们会读取文件的每一行，然后用逗号分隔成 **字段** 。当然我们会碰到一个问题：例如电影标题 **Food, Inc.** 中含有逗号。

代码文件：ch_arch/code/schema2.txt

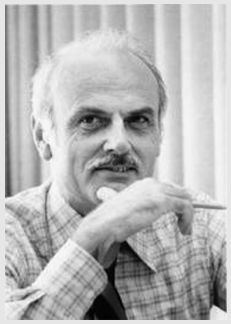
```
Food, Inc.,PG,2008-09-07,The current method of raw...
```

我们不妨在每个字段周围加上引号：

代码文件：ch_arch/code/schema3.txt

```
"Food, Inc.","PG","2008-09-07","The current method of raw..."
```

Edgar F. "Ted" Codd（1923——2003）因为基于 **关系代数** 形式主义的关系数据库的发明，Edgar F. "Ted" Codd荣获了1981年图灵奖。



.....一切都很顺利，直到我们试图输入电影 **等待"超人"** 。如本例所示，即便设计简单的存储格式都会碰到棘手的问题。需要编写代码，将内存中的对象转化成我们的存储表示方式（称为 **编组** 或者 **序列化** 对象，相应地还要进行相反的过程，即 **解组** 或者 **反序列化** 。

幸运的是，人们为满足保持对象这个十分普遍的需求，进而演化出了几个设计模式。这些模式的子集利用 **结构化存储** ——有了这种存储系统，只需指定想要的存储对象结构，而无须编写创建该结构的明确代码，在某些情况下，还可以指定不同类型的对象的连接关系。 **关系数据库管理系统** （RDBMS）始于20世纪70年代，其设计理念基于表示结构和关系的形式主义。在后续章节中我们将详细讨论RDBMSs。简言之，RDBMS存储了一组 **表** ，每个表存储着有一组共同 **属性** 的实体，每行对应于一个实体，每列对应于该实体的属性值。Rotten Potatoes的电影表包括标题、评价、发布时间和描述,表的各行如图2.11所示.

编号	标题	分级	上映日期	描述
1	Gone with the Wind	G	1939-12-15	An American classic ...
2	Casablanca	PG	1942-11-26	Casablanca is a...

图 2.11: 一种RDBMS表格能用于存储电影信息。id列会给出每行 主键 或者永久唯一的标识符。大多数数据库都可以通过各种方式实现主键分配的自动化；Rails使用了非常常见的方法——按照递增序列分配整数

因为模型负责管理应用程序中的数据，所以必须在对内存中模型的操作（如一个呈现电影对象）其在存储层表示操作的方法之间建立对应关系。内存中的对象通常是通过类来表示的，除此之外，还提供了表示对象属性的方法，例如电影标题和评级。Rails框架选择了 **活动记录架构模式** 。在这个模式中，模型类的单个实例（在我们的例子中是一部电影的条目）对应特定的RDBMS表中一行。模型对象的内置行为可以直接在对象数据库表示上进行操作：

- 在表中新建一行（表示一个新对象）。
- 将现有行读进一个对象实例。
- 通过修改的模型实例的新属性值更新已有行。
- 删除行（永久破坏对象数据）。

创建（create）、读取（read）、更新（update）和删除（delete）这四个命令通常缩写为 **CRUD** 。稍后将添加一个新功能，影迷可以评论喜欢的电影，因此会出现一对多的关系或者影迷与评论之间的 **关联** ：在基础上，活动记录利用了基于外键的RDBMS的现有机制（稍后介绍），以便轻松实现内存对象的这些关联。

小结：

- 在MVC架构中的SaaS应用的模型一个重要工作是永久化存储数据，这就需要对象在内存形式和永久存储形式之间转换。
- 各种设计模式已经发展到能够满足这样的需求，其中有很多利用了结构化存储，例如关系数据库管理系统（RDBMS），这不仅简化了模型数据的储存，也简化了模型间关系的维护。。
- 关系数据库管理系统支持四种基本操作，创建，读取，更新，删除（简称CRUD）。
- 在ActiveRecord设计模型中，每一个模型都知道怎样对应类型对象进行CRUD操作。Rails的ActiveRecord库为SaaS应用提供了丰富的功能。

自我检查2.6.1. 下列哪项是结构化存储的例子：（*a*）*Excel*电子表格，（*b*）一个包含邮件内容的文本文件，（*c*）一个包含名称的文本文件，每行一个对应一个名称。
◇（a）和（c）是结构化的，因为一个应用程序读取这些文件时可以仅基于结构就能解释其内容并做出假设。（b）是非结构化的。

2.7 500英尺：路由、控制器与REST

活动记录会告诉每一个模型如何在数据库中创建，读取，更新和删除自己（CRUD）。回想一下2.5节介绍的MVC模式，控制器负责处理发出CRUD请求的用户的实例Web浏览器的交互。在Rails中，每个控制器在一个特定的控制器文件中用一个特定的Ruby方法处理。因此，每一个HTTP传入请求必须被映射到相应的控制器和方法上。这种映射称为 **路由** 。

如图2.3所示，HTTP请求的特点结合了URI和 **HTTP方法** ，有时候也称之为 **HTTP verb** 。HTTP标准定义了大约6种方法，在Web应用程序和面向服务的架构中使用最广泛的是`GET`、`POST`、`PUT`和`DELETE`。由于 **方法** 可能意味着一个函数或者HTTP请求方法，所以讨论路由时我们会用 **方法** 来表示与HTTP verb相关的请求，使用 **控制器行为** 或者 **行为** 表示处理该请求的应用程序代码（方法或函数）。

一个路由将一个URI加上HTTP方法与特定的控制器和行为连接起来。在2000年Roy Fielding在他的博士论文提出了一种一致的方式去将请求映射到一个特别适合面向服务的体系结构的行为中。他的设想是将各种Web应用程序操纵的实体

看作 **资源**，然后设计出路由，让任何HTTP请求都包含识别特定资源和行为的所有必要信息。他称之为 **代表性状态传输**，或者简称为REST。

尽管解释起来很简单，REST对于SaaS应用有着非常强大的组织原则。因为需要让应用程序设计者仔细思考每个请求基于何种条件或假设才能获得自包含，以及如何把应用程序操纵的每种类型的实体表示为一个“资源”，让它可以执行各种操作。据说按照这一指导思想设计的应用程序暴露了RESTful API（应用程序编程接口），映射至特定动作的URI也被认为是RESTful URI。

在Rails中，路由映射由文件`config/routes.rb`中的代码生成。这些内容我们将在第4章学习。虽然Rails并没有强制要求必须使用REST路由，其内置路由支持默认的REST。当位于`rottenpotatoes`目录时，图2.12显示了在终端窗口中输入`rake routes`时显示的信息。

`rake`运行Rotten PotatoesRakefile中定义的维护任务。`rake --help`显示了其他选项。

在一个URI，例如`/movies/:id`中，以‘：’开头的符号是路由参数；在这个例子中，`:id`表示一个实例模型的`id`属性（主关键字）。例如，路由`GET /movies/8`将匹配图2.12中的第二行的，其中`:id`的值为8。因此，如果这个电影存在的话，需要显示电影列表中ID为8的电影的细节。同样，路由`GET /movies`匹配第一行，请求包含所有电影的列表（索引），然后路由`POST /movies`匹配第四行，并且在数据库中创建一个新的电影条目。（`POST /movies`路由不会指定`id`，因为新电影在被创建之前没有ID）。需要注意的是，INDEX和CREATE虽然URI相同，但是HTTP方法不同，因而路由也不同。

资源操作	方法& URI		控制器行为
索引（列表）电影	GET	/movies	**index**
阅读（显示）已经存在的电影	GET	/movies/:id	**show**
显示表中新加入的电影	GET	/movies/new	**new**
从填入的表中创建新的电影	POST	/movies	**create**
显示并编辑表中已经存在的电影	GET	/movies/:id/edit	**edit**
更新表中的电影	PUT	/movies/:id	**update**
删除电影	DELETE	/movies/:id	**destroy**

图 2.12: `rake routes`输出概要显示了Rotten Potatoes识别的路由以及每个路由表示的CRUD行为代表。最右列显示，当请求与给定的URI 和HTTP匹配时，电影控制器中的Rails控制器行为就会被调用。路由至方法的映射在很大程度上依赖于惯例而不是配置，我们将会在第4章看到

重要的是，RESTful接口简化了面向服务的架构的纳入，因为如果所有的请求都是自包含的，那么服务之间的交无须建立或依靠正在进行的会话上。当许多SaaS应用与人类用户通过Web浏览器进行交互时，Jeff Bezos要求（见1.3小）将亚马逊的所有内部服务拥有“外部化”的API。这是一个非常高瞻远瞩的决定。

事实上，当今的实践表明即使创建一个面向用户的SaaS应用设计并通过浏览器使用，我们也应该把应用程序想成是一系列能通过RESTful的API访问的资源，而RESTful的API恰好又能通过Web浏览器访问。不幸的是，这会出现一个小问题，如果有Web编程经验，你可能已经发现，路由在图2.12用的是四种不同的HTTP方法——`GET`、`POST`、`PUT`跟`DELETE`——甚至使用不同的方法来区分具有同一个URI的路由。然而，由于历史原因，Web浏览器只能实现`GET`（链接）和`POST`（提交表单）。为了弥补这一缺陷，Rails的路由机制让浏览器利用`POST`发出请求，这些请求通常需要`PUT`或者`DELETE`。Rails会给与这类请求有关的Web表单加注释，以便提交请求时Rails可以识别这个特殊情况，也可以从内部将控制器“看得见”的HTTP方法改变成适当的`PUT`和`DELETE`。即使浏览器不执行，Rails程序员也可以在支持`PUT`和`DELETE`的假设下进行操作。正如我们看到的，它的优势在于可以使用同一组路由和控制器方法来处理来自浏览器的请求（从人类的角度）和来自SOA的其他

其实，大多数浏览器还使用工具`HEAD`来请求资源的元数据，但在这种情况下我们不必担心这个问题。

代码文件：ch_arch/tables/rest_example.txt

```
                          Non-RESTful Site      RESTful Site
                          ----------------      -------------
Login to site:             POST /login/dave     POST /login/dave
Welcome page:              GET  /welcome        GET  /user/301/welcome
Add item ID 427 to cart:   POST /add/427        POST /user/301/add/427
View cart:                 GET  /cart           GET  /user/301/cart
Pay:                       POST /checkout       POST /user/301/checkout
```

图 2.13: Non-RESTful的请求和路径都依赖于先前请求的结果。在面向服务的架构中，RESTful网站的客户端可以立即要求查看购物车（第六行），而非RESTful网站的客户端必须执行运行第三到五排的内容，以便建立第六行需要的隐含信息

服务的请求。

进一步发掘这两重性的重要性，从图2.12中可以观察到new和create路由（表中的第三行和第四行）同时出现并参与新电影的创建与编辑。为什么需要两个路由才行？其原因是，面向用户的Web应用程序需要两者相互作用来创建一个新的电影，视频2.7.1；而在SOA中，远程服务可以创建一个包含所有信息的请求来创建新的电影，所以它永远不需要使用new路由。

视频2.7.1: 创建和更新都需要两者的相互作用.
光盘文件：ch_arch/mov/create_update.mp4
创建一个新电影需要两个相互作用的Rotten Potatoes，因为在可以提交电影相关信息之前，用户必须提供一个表单来输入这些信息。空的表格是由路由命名的资源，见图2.12的第三行。同时填好的表格的填充是第四行路由命名的一个资源。同样更新现有的电影需要一个表格资源，它包括一个可编辑的能显示现有的影片信息（第五行）的表格和已编辑好并提交的表单资源（第六行）。

RESTfulness似乎是一个理所当然的设计选择，直到Fielding crisply提出并发表了REST的理念。许多Web应用程序都采用了非-RESTful风格设计。图2.13显示，假想的非RESTfully电子商务网站是如何实现用户登录、在购物车中添加一个特殊商品以及结账这些功能的。对于假想的非RESTful网站，登录后（三排）每一个请求依赖于隐含信息：第四排假设网站“记住了”当前登录用户，然后进入了他的欢迎页面，第七排假设网站“记住了”将商品添加到了购物车结账的用户。相反，RESTful网站的每个URI包含了满足请求的所有信息，而且独立于此类信息：Dave登录后，他的用户ID 301出现在了每一次请求中，并且其购物车也通过用户ID得到了明确标识，而非基于当前登录用户概念的隐式信息。

路由和RESTful小结：

- 一个路由包含HTTP方法（`GET`、`POST`、`PUT`或`DELETE`）以及包括一些可能参数的URI。例如Rails应用程序框架就可以映射到控制器操作。
- 对于按照REST（具象状态传输）设计的应用程序，从外部可以视为一个可以在上面执行特定操作的实体集合，每个操作有一个对应的RESTful的请求。该请求包括完成该操作的所有必要信息。
- 如果路径和资源是REST型的，同一个控制器逻辑可以通过浏览器或者来自于SOA的其他服务请求为面向用户的页面提供服务。Web浏览器只支持`GET`和`POST`HTTP方法，框架逻辑可以弥补这个缺陷，这样程序员就可以在所有方法可行的基础上工作。

■ 详细阐述：REST vs. SOAP vs. WS-*

在20世纪90年代后期，人们对SOA越来越感兴趣，供应商和标准组织成立了为SOA互操作制定标准的委员会。致使出现一组为Web Services精心设计的协议，包括WS-Discovery（WS-发现）、WS-Description（WS-说明）等，有时被人们统称为 **WS-*** （Web服务规格列表），同时也被Rails的发明者David Heinemeier Hansson戏称为“WS-Deathstar”。自相矛盾的 **SOAP** 标准（简单对象访问协议）更加简单，但仍比REST复杂多了。总的来说，开发商认为SOAP和WS-*是委员会驱使的超安全标准设计，却因之前的CORBA和DCOM等基于企业的互操作标准的过时设计而负担过重。相比之下，REST极其简单，更像一种哲学而非一个标准。它立即吸引了大量的开发商，因此于如今SOA应用程序的绝大部分都建立在其上。

■ 详细阐述：路由的格式

在大多数路由中，`rake routes`的原始输出包含一个`(.:format)`符号，为了让图2.12更清晰，我们省去了这个符号。如果存在的话，格式说明符允许一个路由以输出格式而非默认的HTML格式请求资源——例如，`GET /movies.xml`会请求XML文档非HTML页面的所有电影列表。尽管在这个简单的应用程序中，我们没有包含生成HTML以外的其他格式的代码，有了这种机制，现有应用程序可以轻松地集成到面向服务的体系结构中——仅需修改几行代码，就可以让所有已有控制器动作成为外部的REST型API的一部分。

自我检查2.7.1. 对或错：如果一个应用程序有一个*REST*型*API*，它必须执行*CRUD*操作。

◇ 错误。只要该应用程序将其实体描述为资源，并制定每种资源类型允许哪些操作，REST原则适用于任何操作类型。

自我检查2.7.2. 对或错：对*REST*操作的支持简化了*SaaS*应用与面向服务架构中的其他服务的集成。

◇ 对

2.8 500英尺：模板视图

下面用一些视图来小结我们短暂的旅程。由于面向用户的SaaS应用主要提

供HTML页面，大多数框架提供了创建夹杂变量或非常简短的代码片段的静态标记页（HTML或其他）的方法。在运行时，变量值或代码执行的结果被取代或 **通过内插值替换** 到页面。这种架构被称为模板视图，它是包括Rails、Django以及PHP在内的许多SaaS框架的基础。

比起Rails内置的 erb 模板系统，我们更喜欢Haml的简洁，所以本书配套软件中预装了Haml。

我们将使用一个叫做Haml的模板系统（为了抽象HTML标记语言，其发音为“HAM-ELL”）以简化HTML模板视图的创建。在第4章中，我们将学习更多的细节并创建自己需要的视图，为了访问Rails应用程序的“移动部件”，需要在Rotten Potatoes目录中打开`app/views/movies/index.html.haml`。这个视图是电影的索引控制器的图；依赖于惯例而不是配置，后缀`.html.haml`表明视图应该让Haml去创建`index.html`，文件的位置和名称将它识别为用于**电影**控制器中的**index**操作的视图。视频2.8.1展示了Haml的基础知识，并在图2.14中进行了小结。

视频2.8.1: 用Haml将结果插入视图中.
光盘文件：ch_arch/mov/haml.mp4
在Haml的模板中，以%开头的行扩展到了对应的HTML起始标签，因为Haml使用缩进来确定结构，所以它没有结束标记。标签后面类似于Ruby的哈希表会成为HTML属性。行——**以破折号开头**会像Ruby代码一样执行并丢弃结果，行**以等号=开头**会像执行Ruby代码一样并把结果插入到HTML的输出中。

Haml	HTML
`%br{:clear => 'left'}`	`<br clear="left"/>`
`%p.foo Hello`	`<p class="foo">Hello</p>`
`%p#foo Hello`	`<p id="foo">Hello</p>`
`.foo`	`<div class="foo">...</div>`
`#foo.bar`	`<div id="foo" class="bar">...</div>`

图 2.14: 一些常用的Haml构成和生成的HTML。以%开头的Haml标签必须包含该标签和每一行的所有内容，例如表中的1～3行，或者必须像4～5行那样用简短的Haml语言表示一整行内容。在这种情况下，后续行里标签的所有内容必须缩进2个空格。注意，Haml特意使用与CSS选择器相似符号指定了`class`和`id`属性

根据MVC所述，视图包含的代码越少越好。尽管Haml技术允许在模板中运行复杂的Ruby代码，但其包含多行代码的语法是一件非常诡异的事情，我们不推荐程序员这样做。事实上，Rotten Potatoes的索引视图中唯一的“运算”仅限于在集合上迭代（通过控制器由模型提供）以及生成一个表示每个元素的HTML表格行。

相比之下，使用PHP框架编写的应用程序往往在视图模板中混杂了大量的代码，虽然一个训练有素的PHP程序员可以将视图从代码中分离，但是PHP框架本身没有为此提供特别的支持，而且这种努力也得不偿失。MVC的倡导者认为，将控制器从视图中分离，会使得人们更容易优先考虑构建一个应用程序作为一组RESTful风格行为，然后通过一个单独的视图步骤渲染这些操作的结果。1.3节讲述了一个面向服务的体系结构的案例；现在我们应该明白如何分离模型、视图和控制器，并遵照REST型控制器的风格，这样程序的操作自然会轻松地“外化”成单机API操作。

■ **详细阐述：模板视图的备选方案**

因为所有的Web应用程序最终必须为浏览器提供HTML，因此围绕一个静态的HTML“模板”建立输出（视图）对于Web应用总是有着良好的效果，所以现在非常流行用模板视图（Template View）模式来渲染视图。所以说，视图渲染阶段的输入包括HTML模板和一组Ruby变量，Haml将用于“填写”动态内容。另一种方法是转换视图模式（Fowler 2002），视图阶段的输入 只 是一组对象。视图代码包括所有把对象转换为所需的视图表示的逻辑。当出现多种不同的视图表示时这种模式会比原先的更有意义，因为视图层不再是“建立在”任何特定的表示上。一个在Rails方法中转换视图的例子是接受ActiveRecord的资源并且生成纯XML表示的资源——这样做并不实例化任何的“模板”，而是使用ActiveRecord对象生成XML。这些方法可以快速把HTML服务的Rails应用程序转换为一种面向服务体系的架构。

自我检查2.8.1. 对于电影描述的图*2.8.1*中，在索引视图中缩进扮演着什么样的角色？
◇ 当一个HTML元素包含了其他元素，缩进会告诉Haml其嵌套结构，以便在恰当的地方生成结束标签，例如`</tr>`。

自我检查2.8.2. 在电影的索引视图中，为什么*Haml*的标记在*11*行以——开头但在*13*∼*16*行中以=开头呢？
◇ 在第10行中，我们只需要执行代码去启动循环。而在13∼16行中，我们想将代码的执行结果替换到视图中。

2.9 谬论和陷阱

谬论：Rails（或Django、PHP等其它框架）不能扩展

如图2.7所示，无共享的3层架构利用云计算在每层添加计算机后，Web服务器和应用服务器层（Rails应用程序运行的地方）几乎可以随意扩展。挑战在于如何扩展数据库，在下一章“陷阱”里会讲到。

陷阱：所有模型数据都被放在了一台计算机服务器的RDMBS上，因而限制了它的可扩展性。

RDBMS的威力是一把双刃剑。它可以轻松创建数据库结构，但是我们之前并没有察觉到它容易受到可扩展性问题的困扰，直到用户服务增长到了成千上万后这个问题才显现出来。一些开发人员认为，Rails使得问题更加复杂，因为它的模型抽象很高效，以致于人们很容易忽略可扩展性问题就进行使用。不幸的是，与Web服务器和应用程序层不同，我们不能通过简单的部署大量数据库副本来解决问题，因为这可能会导致相同项目的副本拥有不同的值（**数据一致性** 问题）。虽然主从复制和数据库 **拆分** 等技术可以帮助数据库层变得更接近于无共享的表示层和逻辑层，但是极端数据库的可扩展性仍然需要理论和工程方面的共同努力。

陷阱：过早关注SaaS应用在单一计算机的性能。

尽管无共享架构使水平扩展更容易，但我们仍然需要物理计算机去实现。在过去，添置新计算机昂贵（购买计算机）、耗时（配置和安装计算机），而且还是永久的（一旦失去需求，就会多余出一台空闲的计算机）。有了云计算，这三个问题就得到了缓解，因为我们可以花很少的钱立即添加一台计算机，而且可以在不需要

时候释放。因此，只要SaaS应用没有大到需要数百台计算机来完成，SaaS开发者应该更专注 **水平扩展** 而不是每台计算机的性能。

2.10 结束语：模式、架构和长寿命的API

无法理解的API是没有用处的。

——James Gosling

了解一个软件系统体系结构就是去了解它的组织原则。我们通过识别许多不同层面的模式来学会它的组织原则，像客户端——服务器、三层体系结构、模型-视图-控制器，活动记录，REST。

模式是一种管理复杂的大型软件系统强有力的方式。1977年，Christopher Alexander在《**模式语言：城镇，建筑，施工**》一书中讲述了民用建筑的设计模式。1995年（Gamma et al. 1994），Erich Gamma, Richard Helm, Ralph Johnson和John Vlissides（“四人组”或者GOF）出版了 **设计模式：可复用的面向对象软件的元素** 一书。它讲述了一种专注于类——级别的结构和行为的模式，当今称为23 GOF设计模式。设计模式已经被普遍当作工具使用，但它们一直备受诟病。例如，谷歌现任研究总监Peter Norvig认为，某些设计模式仅仅弥补了静态类型编程语言的不足，如C++和Java。但对于Lisp或Ruby等动态语言它们就没有存在的必要。尽管有一些争议，对于软件工程师来说，各种模式仍然颇具价值，它能够在工作中识别结构，并可以为反复出现的问题带来行之有效的方案。

诚然，我们注意到，通过选择建立一个SaaS应用，我们预先确定了一些模式的使用并且排除了其他模式。通过选择使用Web标准，我们预先确定了客户端——服务器系统；通过对云计算的选择，我们预先确定了三层架构，以便进行水平扩展。模型-视图-控制器不是预先确定的，尽管可以选择图2.9等其他模式，但我们还是选择了它，因为它非常适合Web应用程序，它以视图为中心并且一直以来都依赖于持久层。REST也不是预先确定的，我们选择它是因为它简化了对面向服务的体系结构的集成，并与MVC的应用程序一样，可以轻松地应用到CRUD操作中。活动记录也许更有争议——正如第4章和第5章所述，其强大的工具大大简化了应用程序，但滥用这些工具会导致扩展性和性能的问题，但这些问题不太可能发生在更简单的持久性模型中。

事实上，Rails原本是由一个单机应用程序提取出来的。该程序是由37咨询集团信号（consulting group 37 signals）编写的。

如果在1995年建立一个SaaS应用，那么任何一个上述的问题都不会变得非常明显，因为从业者还没有积累足够多的成功的SaaS应用例子，用以将成功模式“提取”为框架，如Rails，软件组件Apache和中间件Rack。跟随之前的软件架构师的成功脚步，我们可以利用他们的能力去 **分离所有事物中变化的并保留不变的** 内容，参考SaaS的许多例子并向我们提供以这种方式建立起来的工具、框架和设计原则。正如我们前面提到的，分离是重用的关键。

最后，值得一提的是，Web成功的一个关键因素在于，它始终采用了定义良好的协议和格式，这种设计可以分离变化事物与不变事物。尽管TCP/IP、HTTP和HTML都经历了多次重大修改，但它们都提供了可以检测版本的途径，这样客户端就能知道它是否在与旧服务器对话，反之亦然，并相应地调整其行为。尽管处理多个协议和语言版本给浏览器带来了额外的负担，但它已经导致了一个非凡的结果：在2011年使用20世纪60年代的标记语言创建一个网页，可以使用1969年开发的网络协议检索，并用1992年发明的浏览器显示。分离变化事物与不变事物是创造长寿软件的途径之一。

Tim Berners-Lee, 一个在CERN[12]工作的计算机科学家，他在1990年引领了HTTP和HTML的发展。现在两者都由非营利中立机构控制World Wide Web Consortium（W3C）[13]。

2.11 深入学习

- W3Schools[14]是一个由广告支持的免费网站，几乎拥有所有与Web技术有关的教程。
- Nicole Sullivan[15]一个自称“CSS忍者”的人，他的博客中有不少对于更复杂的网站的不可或缺的CSS/HTML建议。
- 万维网联盟（W3C）[16]管理的正式文件描述Web的开放标准，包括HTTP、HTML和CSS。
- XML/XHTML Validator[17]可以用它来确认使用SaaS应用交付的页面是符合标准的。
- 面向对象的设计网站[18]为使用面向对象的编程语言的开发人员提供了海量的资源，其中包括一个漂亮的GoF设计模式目录，每个模式使用图形描述，部分内容将在第11章详细讨论。

M. Fowler. **Patterns of Enterprise Application Architecture**. Addison-Wesley Professional, 2002. ISBN 0321127420. URL `http://martinfowler.com/eaaCatalog/`.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1994. ISBN 0201633612.

注释

[1] `http://en.wikipedia.org/wiki/Pentium_FDIV_bug`
[2] `http://rottentomatoes.com`
[3] `http://projects.apache.org/projects/http_server.html`
[4] `http://www.iis.net`
[5] `http://docs.google.com`
[6] `http://iana.org`
[7] `http://www.w3.org/TR/html5/introduction.html#history-1`
[8] `http://csszengarden.com`
[9] `http://heroku.com`
[10] `http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/`
[11] `http://martinfowler.com/eaaCatalog`
[12] `http://info.cern.ch`
[13] `http://w3.org`
[14] `http://w3schools.com`
[15] `http://www.stubbornella.org/content`
[16] `http://w3.org`
[17] `http://validator.w3.org`
[18] `http://oodesign.com`

2.12 推荐作业

练习2.1. 如果域名解析服务停止了，你还能上网吗？解释为什么能或者不能。

练习2.2. 假设*HTTP cookies*不存在。你能不能设计一种其他的方法来追踪访问网页的用户？
提示：这个方法与修改*URI*有关，在发明*cookies*以前是一种广泛使用的方法。

练习2.3. 找到一个网页使得*W3C*的*XHTML*验证器[1] 至少找到一个错误。这应该是很容易的。通读验证错误的消息，尽量理解每一条的意思。

练习2.4. 下列每一条*URI*指的是什么端口号，为什么？

(1) `https://paypal.com`

(2) `http://mysite.com:8000/index`

(3) `ssh://root@cs.berkeley.edu/tmp/file`
提示：召回*IANA*建立各种网络服务的默认端口号。

String#ord 返回字符串的第一个 **码点**（在字符集里与字符对应的数值）。如果字符串以 **ASCII** 来编码，那么**ord**返回第一个字符的ASCII码。所以**"%".ord**显示**%**的ASCII码，而**"%".ord.to_s(16)**显示它等价的十六进制数。

练习2.5. 如谷歌的*API*文档所描述[2]你可以通过创建一个将搜索词条作为参数的*URI*来进行谷歌搜索。比如，用`http://www.google.com/search?q=saas`来搜索*SaaS*。然而如图*2.3*所示，一些字符是不能出现在*URI*中的，因为它们很“特殊”比如空格，‘?’，和‘&’。在这样的限制下，请创建一个合法的*URI*来搜索“$M\&M$”和“100%”。

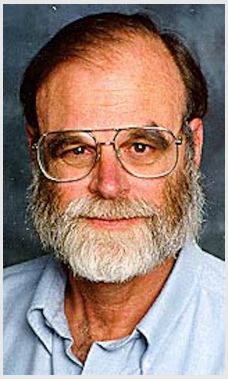
练习2.6. 为什么*Rails*的路由映射到控制器，但不是模型或视图？

练习2.7. 给定一个高层次的设计，通过区分常用的软件架构，比如三层、管道和过滤器、客户端服务器，来指出软件架构。

练习2.8. 调查大家对软件架构选择影响，其架构是用来设计一个简单的系统。从客户端-服务器的替代品着手，然后替代品的请求-答复，并依此类推。

3 SaaS架构：针对Java编程者的Ruby简介

Jim Gray
（1944——2007年于海上失踪）是计算机科学领域的一个友好的伟人。他是第一个从加州大学伯克利分校计算机科学系获得博士学位的人，指导了世界上数百名的博士和教师。因为在数据库和事务处理研究以及系统实现方面的技术领导作用，他于1998年获得了图灵奖。



嗯，这篇<匿名>的论文不是唯一的（出于相同的原因）。
B-tree相关的论文开始也被拒绝。
事务相关的论文开始也被拒绝。
数据立方体的论文开始也被拒绝。
五分钟原则的论文开始也被拒绝。
但是仅对之前工作的线性拓展的论文被接受了。
所以，请重新提交！一定！！！

——Jim Gray，发给Jim Larus的关于一篇被拒绝文章的邮件，2000

3.1 概述以及Ruby的三大支柱 64
3.2 一切都是对象 68
3.3 每一个操作都是一个方法调用 69
3.4 类、方法和继承 71
3.5 所有的编程都是元编程 75
3.6 块：迭代器、函数式习语以及闭包 77
3.7 mix-in和鸭子类型 81
3.8 用Yield构造自己的迭代器 82
3.9 谬论和陷阱 85
3.10 结束语：惯用语言使用 86
3.11 深入学习 87
3.12 建议的项目 87

重要概念

本章介绍例如Java等语言的基本面向对象的功能在Ruby中的等价机制，以及一些Java没有的有助于重用和简洁的Ruby机制。

- 在Ruby中 **任何事物** 都是一个对象，即使一个不起眼的整数，而且Ruby的所有操作都是通过调用对象的方法来完成的。
- **反射** （Reflection）让你的代码在运行的时候进行自我检查，而 **元编程** （metaprogramming）允许它在运行的时候产生和执行新的代码。
- Ruby从 **函数程序设计** （functional programming）里借鉴了一些不错的想法，尤其是 **块** 的使用——参数化的代码块称作 **lambda表达式** ，这些表达式自带一个作用域，组成 **闭包** 。
- 对Ruby的**yield**语句，Java中没有与之对应的部分，通过分离数据结构的遍历和对数据结构中元素的操作，用一种类似于"协同"的机制增强了重用。
- 因为Ruby的 **动态类型** ，为了判断对某一个对象是否可以调用某种方法，你不需要考虑对象的类型——仅仅需要判断它是否对这种方法有响应。有些人称这种特性为 **鸭子类型** （duck typing）："如果它看起来像一个数组，走路起来像一个数组，而且叫起来也像一个数组，那么你就可以把它当成一个数组。"

元编程、反射、动态类型以及使用**yield**的块，可以巧妙结合起来去编写更加无重复的以及更加简洁的也更加漂亮的代码。

3.1 概述以及Ruby的三大支柱

编程只能通过实践来学习，页面上放置这个图标的地方，表示我们强烈鼓励你自己尝试一些例子。由于Ruby是解释语言，没有编译步骤——当尝试一些例子、探索和实验都很容易时，你立刻就能体会到满足感。每个例子都包含在本书的配套光盘中，在这里你可以复制实例的代码，再粘贴到Ruby的解释器或者编辑器窗口。我们也鼓励你去查看官方说明文档来获取关于3.11节里介绍的内容的更多细节。

Ruby 是一种极简主义的语言：虽然它的库很丰富，但是 **语言本身** 的机制却很少。这些机制相关的三个原则将会帮助你阅读和理解惯用的代码：

(1) 一切都是对象。在Java中，一些简单的类型，例如整数必须“封装”起来以便让它们像对象那样操作。

(2) 每一个操作都是调用某些对象的方法并且返回一个值。在Java中，操作符重载和方法重载是不一样的，并且**void** 函数可能没有返回值。

(3) 所有的编程都是元编程：类和方法可以在任何时候添加或者改变，即使程序已经在运行。在Java中，所有的类必须在编译时声明，基类甚至不能在编译时修改。

这三个原则中的每一个都将会在它们各自的部分讲到。#1和#2简单直白。#3给了Ruby提高生产力的推动力，但是必须得注意能力越大，责任越大。使用Ruby的元编程特性将会让代码优雅、简洁、无重复，但是滥用它们会使代码可靠性差并且让人费解。

如果你熟悉其他现代脚本语言，那么Ruby的基本语法应该是不足为奇的。图3.1给出了基本的Ruby元素的语法。语句通过换行（最常用）或者分号（少用）分开。缩进无关紧要。而Ruby足够简洁，一行代码很少超过一个屏栅线，在不会导致解析歧义的情况下分开一个带换行的语句是可以的。如果你不清楚断句是否合法，那么具有良好的语法高亮显示的编辑器能够给你提供帮助。

一个 **符号** ，例如**:octocat**，是一个值为本身的不可变字符串；在Ruby中通常用于枚举，类似于C或者Java中的**enum**类型，尽管它还有其它用途。但是符号和字符串是不一样的——它是它自己的基本类型，尽管可以通过调用**to_s**来转换成字符串，但是不能对它执行字符串操作。例如，**:octocat.to_s**得到**"octocat"**，而**"octocat".to_sym**得到的是**:octocat**。

Regular expressions 或者 **regexps** （为便于发音通常写作 **regex** 和 **regexes** ）是程序员工具箱的一部分。正则表达式针对可能包含有“通配符”的模式允许匹配字符串。Ruby的正则表达式与其他现代编程语言支持一些相似点：正则表达式出现在斜杠之间，并且可能随后出现一个或者多个字母修改它们的操作，例如，**/ regex /i** 用来表明正则表达式在匹配的时候应该忽略大小写。如图3.2 所示，在正则表达式中的特殊结构，可以匹配多种类型的字符，并且指定一个匹配必须出现的次数以及匹配是否必须“固定”在字符串的开头或者结尾。例如，一个匹配一天中的时间（独占一行）的正则表达式，“8:25pm”：

代码文件：ch_ruby_intro/code/ruby/regexp_time.rb

```
time_regex = /^\d\d?:\d\d\s*[ap]m$/i
```

这个正则表达式匹配字符串（**^\d**）开头的一个数字，后面可以选择跟另外一个数字（**\d?**），后面再跟一个冒号，恰好两个数字，0个或者多个空白字符（**\s***），不论是**a** 还是**p**，然后是字符串（**m$**）结尾的**m** ，忽略大小写（结束斜

变量	local_variable，@@class_variable，@instance_variable
常量	ClassName，CONSTANT，$GLOBAL，$global
布尔变量	false，nil是错; true 和 其他所有 （0，空字符串等等）是对。
字符串和符号	"string"，'also a string'，%q{like single quotes}，%Q{like double quotes}，:symbol 特殊字符（\n）在双引号里会被转义，而在单引号里则不会
双引号 字符串里的表达式	@foo = 3 ; "Answer is #{@foo}"; %Q{Answer is #{@foo+1}}
正则表达式匹配	"hello" =~ /lo/ or "hello".match(Regexp.new 'lo')
数组	a = [1, :two, 'three'] ; a[1] == :two
散列	h = {:a =>1, 'b' =>"two"} ; h['b'] == "two" ; h.has_key?(:a) == true
实例方法	def method(arg, arg)...end （使用 *args 来指定可变数目的参数）
类(静态) 方法	def ClassName.method(arg, arg)...end def self.method(arg, arg)...end
特殊方法名称 以 ? 和 ! 结束这些方法的名称 是可选的但是通常就这样用	def setter=(arg, arg)...end def boolean_method?(arg, arg)...end def dangerous_method!(arg, arg)...end

条件句	迭代（见3.6节）	异常
if cond (or unless cond) statements [elsif cond statements] [else statements] end	while cond (or until cond) statements end 1.upto(10) do |i|...end 10.times do...end collection .each do |elt|...end	begin statements rescue AnError => e e 是类型为 AnError 的异常； 这里允许多个 rescue 语句 [ensure 这个代码总是执行] end

图 3.1: 基本Ruby元素和控制结构，方括号[]内为可选项目

	符号	意义	例子	匹配	例子	不匹配
计数	*	0或更多	a*		aaaa	b
	+	1或更多	a+	a	aaaa	
	?	0或者1	a?		a	aaaa
位置，集合，范围，附加	^	行首，也不在集合里	^a	a	ab	ba
	$	行尾	a$	a	dcba	ab
	()	组，也捕获 Ruby中的组	(ab)+	ababab	ab	b
	[]	集合	[ab]	a	b	ab
	[x-y]	字符范围	[0-9]	3	9	a
	|	或者	(It's|It is)	It's	It is	Its
	[^a]	不在集合里（相反）	[^"]	b	9	"
	.	任意字符（除了换行符）	.{3}	abc	1+2	aa
	\	用来匹配元字符，也用于类	\.$	The End.	.	a
	i	添加到模式来指定不区分大小写的匹配	\ab\i	Ab	ab	a
类	\d	十进制数字（[0-9]）	\d	3	9	a
	\D	非十进制数字（[^0-9]）	\D	a	=	3
	\s	空白字符	\s			a
	\S	非空白字符	\S	a	=	
	\w	“单词” 字符 ([a-zA-Z0-9_])	\w	a	9	=
	\W	“非单词”字符 ([^a-zA-Z0-9_])	\W	=	$	a
	\n	换行	\n	--	--	a

图 3.2: Ruby正则表达式回顾

杠之后的**i**）。匹配一个或者两个数字的另外一种方式是**[0-9][0-9]?**，而另外一种 **恰好** 匹配两个数字的方式是**[0-9][0-9]**。

Ruby允许在正则表达式中使用括号来捕获匹配的字符串或者子字符串。例如，下面这个包含有三个捕获组的正则表达式：

代码文件：ch_ruby_intro/code/ruby/regexp_capture.rb

```
x = "8:25 PM"
x =~  /(\d\d?):(\d\d)\s*([ap])m$/i
```

第二行尝试匹配字符串**x**。如果匹配成功，则操作符**=~**返回字符串（0是第一个位置）中匹配成功的位置，全局变量**$1**值为**"8"**，**$2**值为**"25"**，**$3**值为**"P"**。当执行下一个正则表达式的匹配时，上一个匹配的变量会被重置。如果匹配失败，则**=~**返回**nil**。注意**nil**和**false**实际上并不相等，但是在条件表达式中都是用来评估“错误”的（实际上，在Ruby中 **只有** 这两个值这样表达）。一般来说，真正的布尔变量方法（即，返回值只能是“真”或“假”）返回**false**，比较而言，这里的方法成功的时候返回一个对象，而失败的时候返回**nil**。

最后，请注意，**=~**不仅可以用在字符串上，也可以在**正则表达式**中的对象上，所以下面的情况是合法且等价的，而在代码中应该选择最易理解的一种。

代码文件：ch_ruby_intro/code/ruby/regexp_two_ways.rb

```
"Catch 22" =~ /\w+\s+\d+/
/\w+\s+\d+/ =~ "Catch 22"
```

小结：

- Ruby中的一个独有的基本数据类型是符号，一个值为本身的不可变的字符串。符号在Ruby中常用来表示“特殊性”，例如作为一组固定选择如枚举中的一项。符号和字符串不一样，但是它们可以通过**to_s**和**to_sym**方法互相转换。
- Ruby语句通过换行符来分隔，或者用分号来隔开，但这种方法不太常用。
- Ruby的正则表达式和其他的现代语言有一些相似点，包括对于使用括号的捕获组，以及匹配修改符，例如用于“匹配时忽略大小写”的后导**i**的支持。

自我检查3.1.1. 下面的Ruby表达式哪些相互等价：

（a）**:foo** （b）**%q{foo}** （c）**%Q{foo}** （d）**'foo'.to_sym** （e）**:foo.to_s**

◇ （a）和（d）相互等价，（b）、（c）和（e）相互等价。

自我检查3.1.2. 当字符串**25 to 1**和下列的每一个正则表达式匹配时哪些被**$1**捕获：

（a）**/(\d+)$/**

（b）**/^\d+([^0-9]+)/**

◇ （a）字符串“**1**”，（b）字符串“ **to** ”（包括首尾空格）。

自我检查3.1.3. 在什么情况下这种写法是正确的？

Fixnum num=3

初始化变量**num**：（a）第一次使用时；（b）任何时候，只要每次都是同一类**Fixnum**；（c）任何时候都不正确。

◇ 任何时候都不正确，因为Ruby中不用变量声明。

3.2 一切都是对象

Ruby支持常见的基本类型——定点整数（**Fixnum**类）、浮点数（**Float**类）、字符（**String**类）、线性阵列（**Array**类）、关联数组或者哈希表（**Hash**类）。但是和Java不同的是，Ruby是 **动态类型**：变量类型直到运行时才能推理出来。也就是说，虽然对象有类型，引用它们的变量却没有。所以在同一个代码块中**s = 5**可以出现在**s = "foo"**后。因为变量没有类型，一个数组或者哈希表可以由所有不同类型的元素组成，如图3.1所示。我们仅仅会说“一个数组”而不会说“一个整数数组”，以及“一个哈希散列”而不是“键为Foo类型，值为Bar类型的哈希”。

Ruby的对象模型源于Smalltalk语言，而Smalltalk的设计灵感又来自于Simula。Ruby中的一切，甚至于一个普通的整数，都是某个类的实例的对象。所有的操作，毫无例外，都是通过调用基于对象的方法来实现的，并且每一个这样的调用（也就是，每个Ruby语句）返回一个值。表达式**obj.meth()**调用对象**obj**上的**meth**，obj被称为 **接收器** 并且希望它能够 **回复 meth**。我们很快就会看到，方法参数周围的括号通常都是可选的。例如：

代码文件：ch_ruby_intro/code/ruby/call_method.rb

```
5.class     # => Fixnum
```

（我们强烈推荐你通过输入`irb`来启动虚拟机终端上的Ruby解释器，这样你就可以尝试这些实例。）上面的调用把**类**的调用方法发给对象**5**。这种**类**方法返回对象所属的类，上述情况返回的是**Fixnum**。（我们将在代码实例中用表达式**# =>**来表示给定一个语句时解释器应该打印出的结果。）

Ruby中的类本身也是一个对象——它是**类**的一个实例，也就是一个实例是类的类（一个 **元类**）。

每一个对象都是某个类的实例。类可以像Java中一样继承超类，所有的类最终都继承**BasicObject**，有时也被称为 **根类**。Ruby不支持多重继承，所以每一个类仅有一个超类，除了**BasicObject**，它没有超类。和大多数支持继承的语言一样，如果一个对象受到类中没有定义的方法调用，那么调用将会被传给超类，直到到达根类。如果没有一个类，包括超类，可以调用这个方法，那么就会引发 **未定义的方法** 异常。

尝试用**5.class.superclass**来找出**Fixnum**的超类；这描述了 **方法链**，一个非常常用的Ruby习语。方法链是左结合的，所以这个实例可以写做**(5.class).superclass**，意为：“对接收者**5**调用**class**方法，不论结果如何，调用**superclass**方法”。

面向对象（OO）和类继承是 **完全不同的特性**。因为很多流行的语言，例如Java把两者结合起来，很多人混淆了二者。Ruby恰好也有这两种特性，但是并不像Java中那样相互融合。特别的是，和Java相比，通过继承来重用并不是那么重要，但是面向对象的影响更为重要了，例如，Ruby 支持广泛的 **反射** ——对象询问自身的能力。**5.respond_to?('class')** 表明对象**5**能够在需要时回应**类**方法。**5.methods** 列出了所有对象**5**能够作出响应的方法，包括这些在它的祖先类中定义的。假设一个对象能够响应一个方法，那么你如何来判断这个方法是否在对象的类或者祖先类中被定义过呢？**5.method(:+)**表明+在类**Fixnum**中被定义过，而**5.method(:ceil)**表明**ceil**在类**Integer**中定义过，这是**Fixnum**的祖先类。判断哪个类的方法能够处理一个方法调用的过程叫做在接收者上 **查找一个方法**，这和Java中的虚拟方法调度类似。

小结：

- 表达式**a.b**表示“调用对象**a**上的**b**”。对象**a**是所谓的 **接收者**，如果它不能处理这个方法调用，那么它把这个调用传递给它的超类。这个过程称为在接收者上 **查找一个方法**。
- Ruby有全面的 **反射**，让你能够询问对象关于它们本身的内容。

■ **详细阐述：查找方法**

之前我们说如果在接收者的类中方法查找失败，那么这个调用会被传到祖先（超类）。事实其实有点微妙：我们稍后将会介绍的mix-in，可以在传递给超类 **之前** 处理一个方法调用。

自我检查3.2.1. 为什么**5.superclass**导致“未定义方法”错误?
提示：考虑调用5本身的**超类**和调用**5.class**返回的对象上的**超类**的不同。
◇ **superclass**是定义在类上的一种方法。对象**5**本身不是一个类，所以你不可以对其调用**superclass**。

3.3 每一个操作都是一个方法调用

为了巩固每一个操作都是一个方法调用的概念，我们需要注意即使基本的数学操作符，例如**+**、*****、**==**（等式比较）对于方法调用来说也是 **糖衣语法**（syntactic sugar）：操作符实际是对接收者的方法调用。数组间接引用如**x[0]**以及数组赋值，例如**x[0]="foo"**也是这样的。

图3.3中的表格展示了这些表达式以及3.1节中引入的正则表达式语法的去糖衣版本，同时也展现了Ruby的核心方法**send**是如何把任意的方法调用发给任意的对象的。包括**send**在内的很多Ruby方法接受一个符号或者一个字符串参数，所以表格中的第一个例子也可以写做**10.send('modulo',3)**。

“每一个操作都是一个方法调用”的重要影响就是像 **类型转换** 这样的概念在Ruby中很少用。在Java中，如果写出式子**f+i**，其中**f**为浮点数，**i**为整数，那么根据类型转换原则，**i**会在内部被转换为一个浮点数使之可以和**f**相加。如果写出**i+s**，其中**s**是**字符串**，那么编译时就会出现错误。

相反的是，Ruby中**+**可以像其他任何方法一样，在每个类中的定义都不同，所以它的行为完全依赖于接受者执行的方法。由于**f+i**对**f.+(i)**来说是糖衣语法，它完全取决于**+**方法（假设会在**f**的类或者它的祖先类中定义到）去决定该如何处理不同类型的**i**值。因此，**3+2**和**"foo"+"bar"**都是合法的Ruby表达式，值分别为**5**和**"foobar"**，但是第一个是调用了**Numeric**（**Fixnum**的祖先类）中的**+**，而第二个调用了**String**中的**+**。如上所示，你可以验证**"foobar".method(:+)**和**5.method(:+)**使用了不同的方法。尽管这很像其他语言中的操作符重载，但是它更为通用：因为对调度来说只有方法的名称起到作用了，我们将会在3.7节中看到这种特性是如何提供一个强大的mix-in重用机制的。

Ruby中表达式**ClassName#method**用来表示**ClassName**中的实例方法**method**，而**ClassName.method**表示**ClassName**中的类（静态）方法**method**。使用这个表达式，表达式**3+2**就是调用接受者**3**上的**Fixnum#+**，而表达式**"foo"+"bar"**是调用接受者**"foo"**上的**String#+**。

有糖衣的	去糖衣的	显式的send
10 % 3	10.modulo(3)	10.send(:modulo, 3)
5+3	5.+(3)	5.send(:+, 3)
x == y	x.==(y)	x.send(:==, y)
a * x + y	a.*(x).+(y)	a.send(:*, x).send(:+, y)
a + x * y	a.+(x.*(y))	a.send(:+, x.send(:*, y)) （操作符优先级保存）
x[3]	x.[](3)	x.send(:[], 3)
x[3] = 'a'	x.[]=(3,'a')	x.send(:[]=, 3, 'a')
/abc/, %r{abc}	Regexp.new("abc")	Regexp.send(:new, 'abc')
str =~ regex regex =~ str	str.match(regex) regex.match(str)	str.send(:match, regex) regex.send(:match, str)
$1...$ n （正则表达式捕捉）	Regexp.last_match(n)	Regexp.send(:last_match, n)

图 3.3: 第一列是Ruby常规操作的糖衣语法，第二列是显式的方法调用，第三列给出了如何用Ruby的send执行相同的方法调用，对方法名称来说它可以是一个字符串或者一个符号（更习惯的用法）

同样的，在Java中变量到子类的显式类型转换也很常见，例如**Foo x = (Foo)y**，其中**y**是**Foo**的子类的一个实例。在Ruby中这是毫无意义的，因为变量没有类型，而且响应方法是在接受者的类中还是祖先类中都无关紧要。

方法是以**def method_name(arg1,arg2)**开始定义的，以**end**结束；二者之间的所有语句都是方法定义。Ruby中的每一个表达式都有值——例如，赋值表达式的“值”是表达式右手边的值，所以**x=5** 的值是5——如果一个方法不包括显式的**return(blah)**，那么返回的是方法中的最后一个表达式的值。所以下面这个简单的方法返回的是5：

代码文件：ch_ruby_intro/code/ruby/trivial.rb

```
def trivial_method    # no arguments; can also use trivial_method()
  x = 5
end
```

例子中的**x**是一个局部变量；它的作用域被限制在定义的块中，此例中的作用域是这个方法，**x**在方法外是未定义的。换句话说，Ruby对局部变量使用了 **词法作用域** 。当探讨到Ruby中的类时，我们将会看到怎样用类和实例变量替换局部变量。

一个重要的Ruby习语是 **诗歌模式** ：当解析明确时可以省略括号和大括号。大多数情况下，Ruby编程者可能会省略方法调用时参数周围的括号，并且当最后一个参数是哈希时省略哈希的大括号。因此对一个方法**link_to**（我们将会在4.4节中碰到），它有一个字符串参数和一个哈希参数，以下两种方法调用是等价的：

代码文件：ch_ruby_intro/code/ruby/poetrymode.rb

```
link_to('Edit', {:controller => 'students', :action => 'edit'})
link_to 'Edit',  :controller => 'students', :action => 'edit'
```

诗歌模式（Poetry mode）在有经验的Ruby开发者中极为普遍，在Rails中也极为常见，当使用习惯后，它就能够给你提供一个友好的消除杂乱符号的方式。

小结：

- Ruby中的一切都是对象，即使整数这种基本类型。
- Ruby对象有类型，但是引用它们的变量没有类型。
- Ruby对局部变量使用词法作用域：一个变量定义在它第一次被赋值的作用域内以及在其闭包内的所有作用域，但是在一个内部作用域内暂时地重用相同的局部变量名称会“遮蔽”包含这一作用域的作用域中相同的名称。
- 通过允许你在生成的代码句法明确时省略方法参数周围的括号和哈希周围的大括号，诗歌模式可以减少杂乱的符号。

■ **详细阐述：参数数目**

尽管在方法定义和方法调用时方法参数周围的括号都是可选的，但是参数 **数目** 对其有影响，方法调用时参数数目错误也会发生异常。下面的代码片段给出了当你需要更多灵活性时的两个习惯用法。第一个就是，把最后一个参数变成哈希类型，并且赋给它一个默认值**{}**（空哈希）。第二个就是使用星号（*****），它会将任何额外的参数收集到一个数组里。面对这么多的特定写法，哪种让代码更可读，哪种就是正确的用法。

代码文件：ch_ruby_intro/code/ruby/optional_args_example.rb

```
# using 'keyword style' arguments
def mymethod(required_arg, args={})
  do_fancy_stuff if args[:fancy]
end

mymethod "foo",:fancy => true # => args={:fancy => true}
mymethod "foo"                # => args={}

# using * (splat) arguments
def mymethod(required_arg, *args)
  # args is an array of extra args, maybe empty
end

mymethod "foo","bar",:fancy => true # => args=["bar",{:fancy=>true}]
mymethod "foo"                      # => args=[]
```

自我检查3.3.1. 显式的-**send**和下列每个表达式中的哪部分相等：**a<b**、**a==b**、**x[0]**、**x[0]='foo'**。
◇ **a.send(:<,b)**，**a.send(:==,b)**，**x.send(:[],0)**，**x.send(:[]=,0,'foo')**

自我检查3.3.2. 假设方法**foo**有两个哈希参数。解释一下为什么我们不能用诗歌模式来写
foo :a => 1, :b => 2
◇ 没有大括号，我们就无法判断这个调用是要传递有两个键的哈希参数还是各有一个键的两个哈希参数。因此诗歌模式仅可以用于参数表中只有一个哈希参数且是最后一个参数的情况下。

3.4 类、方法和继承

图3.4中**Movie**的一个类定义的摘录说明了Ruby中定义一个类的某些基本的概念。下面逐行分析。

代码文件：ch_ruby_intro/code/ruby/oo_1.rb

```
1  class Movie
2    def initialize(title, year)
3      @title = title
4      @year = year
5    end
6    def title
7      @title
8    end
9    def title=(new_title)
10     @title = new_title
11   end
12   def year ; @year ; end
13   def year=(new_year) ; @year = new_year ; end
14   # How to display movie info
15   @@include_year = false
16   def Movie.include_year=(new_value)
17     @@include_year = new_value
18   end
19   def full_title
20     if @@include_year
21       "#{self.title} (#{self.year})"
22     else
23       self.title
24     end
25   end
26 end
27
28 # Example use of the Movie class
29
30 beautiful = Movie.new('Life is Beautiful', '1997')
31
32 # What's the movie's name?
33 puts "I'm seeing #{beautiful.full_title}"
34
35 # And with the year
36 Movie.include_year = true
37 puts "I'm seeing #{beautiful.full_title}"
38
39 # Change the title
40 beautiful.title = 'La vita e bella'
41 puts "Ecco, ora si chiama '#{beautiful.title}!'"
```

图 3.4: Ruby中的一个简单的类定义。第12和13行提醒我们使用分号把单行上的简短语句结合起来是常用方法；大多数Ruby老手利用Ruby的简洁性在分号周围加入空格以提高程序的可读性

第1行开始了**Movie**类。正如我们将看到的一样，和Java不同，**class Movie 不是**一个声明，实际上是一个方法调用创建了一个对象，而这个对象代表着一个新类，并且将这个对象赋值给常数**Movie**。随后的方法定义将会出现在这个新建的类的上下文中。

第2行定义了类的默认构造函数（当**Movie.new**的时候会被调用），它 **必须** 命名为**initialize**。把一个方法命名为**initialize**，但是却称它为**new**，这种不一致是很不幸的特质，但是你必须习惯。（在Java 中，你也可以用其他的名字来定义额外的构造函数。）这个构造需要两个参数，在**第3和4行**中，它为新的**Movie**对象的 **实例变量** 设置了对应的值。实例变量，例如**@title**，都和对象的每个实例相关联。传入的局部变量**title**和**year**在这个构造之外是未定义的，所以如果需要这些值，我们必须从实例变量中获取。

第6~8行给**@title**实例变量定义了 **getter方法** 或者是 **accessor方法** 。你可能想知道为什么，如果**beautiful**是**Movie**的一个实例，我们不能直接写**beautiful.@title**。因为在Ruby中，**a.b**也表示"对接受者**a**调用方法**b**"，而**@title**并不是**Movie**类中的任何方法的名称。事实上，对方法来说它并不是一个合法的名称，因为仅有实例变量和类变量的名称才可能以**@**开头。在这里，**title** getter是**Movie**类的一种实例方法。这就是说**Movie**类的任何一个实例的对象（或者是**Movie**的子类，如果有子类的话），都可以响应这种方法。

第9~11行定义了实例方法**title=**，它和实例方法**title**是有区别的。名称以**=**结束的方法是 **setter** 或者 **mutator** 方法，并且就和getter一样，我们需要这种方法，因为我们不能写**beautiful.@title = 'La vita e bella'**。但是，如第40行所示，我们 **可以** 写**beautiful.title = 'La vita e bella'**。小心！如果你习惯了Java或者是Python，很容易会认为这个语法是 **给一个属性赋值** ，但是实际上它和其他的一样仅仅就是方法调用，并且可以写做**beautiful.send(:title=, 'La vita e bella')**。并且因为它是一个方法的调用，它有一个返回值：在没有一个明确的**return**语句的时候，方法的返回值就是方法中最后一个表达式计算得到的值。因为在这种情况下，方法中的最后一个表达式是赋值**@title=new_title**，并且任何时候都是它的右边值，这个方法恰好返回了传递给**new_title** 的值。

和允许属性以及getters和setters的Java不同，Ruby的数据隐藏或者是 **封装** 是完全的：从一个类的外部访问实例变量或者类变量的 **唯一** 方法就是通过方法调用。这个限制是和Java相比Ruby被认为是一种更加"纯粹的"OO语言的原因之一。但是因为诗歌模式允许省略括号，比如写**movie.title** 而不是**movie.title()**，所以我们并不需要牺牲简洁性来实现这个更强的封装。

第12和13行给**year**定义了getter和setter方法，并且表明可以使用分号或者是换行来分开Ruby语句，如果你认为这样更加简洁的话。但是，我们很快就会看到，通过元编程，Ruby提供了一个更加简洁的方式来定义getters和setters。

第14行是一个注释，在Ruby中以**#**开头，直到这一行的末尾结束。

第15行定义了一个 **类变量** ，在Java里也称为 **静态变量** 。这个变量定义了打印电影名字时是否需要包含年份。和**title**的setter方法类似，对**include_year=**（**第16~18行**）我们也需要一个setter，而方法（**Movie.include_year=**）的名称中出现的**Movie**也表明这是一个类方法。需要注意的是，对这一类变量我们还未定义一个getter；这意味着这个类变量的值不能从类的外部访问到。

第19~25行定义了实例方法**full_title**，它用**@@include_year**的值来决定如何打印出电影的全称。第21行表明语法**#{}**可以用来向带双引号的字符串中插入（替代）表达式的值，就像**#{self.title}** 和**#{self.year}**一样。更准确地说，**#{}**计算了括号内的表达式的值，并且对结果调用**to_s**，使其将本身转换为一个字符串，从而可以插入到包含这一变量的字符串中。**Object**类（除了**BasicObject**之外的所有类的

祖先类）定义了一个默认的函数**to_s**，但是大多数的类都重写了这个函数来定义一个更漂亮的自身的字符串的表示。

Java	Ruby
class MyString extends String	class MyString < String
class MyCollection extends Array implements Enumerable	class MyCollection < Array include Enumerable
静态变量：static int anInt = 3	类变量：@@an_int = 3
实例变量：this.foo = 1	实例变量：@foo = 1
静态方法：public static int foo(...)	类方法：def self.foo ... end
实例方法：public int foo(...)	实例方法：def foo ... end

图 3.5: 在Ruby和Java之间直译的一些特性总结

图3.5中Ruby和Java基本的OO构造比较小结：

- **class Foo**打开一个类（新的或者是已经存在的），从而添加或者改变其中的方法。和Java不同的是，它并不是一个声明，而是立即执行的实际代码，创建了一个新的**Class**对象，并且把它赋值给常数**Foo**。
- **@x**指定一个实例变量，**@@x**指定一个类（静态）变量。命名空间不同，所以**@x**和**@@x**是不同的变量。
- 一个类的实例变量和类变量仅仅可以在类内部被访问。任何来自于"外部世界"的访问都需要对getter或者setter的方法调用。
- 类**Foo**的类方法可以通过**def Foo.some_method**或者**def self.some_method**来定义。

■ **详细阐述：自定义一个类方法。**

我们很快就会看到，类方法定义**def Movie.include_year**实际上可以出现在 **任何地方**，甚至是**Movie**类定义之外，因为Ruby允许在类定义之后添加和修改类中的方法。然而，另一种在类中定义方法**include_year**的途径是**def self.include_year=(...)**。这是因为，正如之前提到的一样，第1行中的**类 Movie**并不是一个声明，而是所在文件一旦被加载就会被执行的实际代码，在**class Movie...end** （第2∼25行）的代码块中，**self**的值是由**class**关键字创建的新的类对象。（事实上，**Movie**本身仅仅是一个普通的指向这个类对象的Ruby常数，可以通过**c = Movie**与**c.new('Inception',2010)**来验证。）

■ **详细阐述：为什么self.title在Movie#full_title中？**

在第19～25行，为什么调用**self.title**和**self.year**，而不是直接指向**@title** 和**@year**，这在一个实例方法中也是完全合法的？原因就是将来我们可能想改变getters方法工作的方式。例如，当4.3节介绍Rails和ActiveRecord时，我们将会看到基本的Rails模型中getters是通过从数据库中检索信息来工作的，而不是使用实例变量来跟踪信息。使用getters和setters来封装实例和类变量，对调用它们的代码隐藏这些属性的具体实现，并且破坏一个实例方法内部的封装没有任何好处，即使这样做是合法的。

自我检查3.4.1. 为什么**movie.@year=1998 不能** 替代**movie.year=1998**？
◇ 表达式**a.b**总是表示"对接受者**a**调用方法**b**"，但是**@year**是一个实例变量的名称，而**year=** 是一个实例方法的名称。

自我检查3.4.2. 假设删除了图3.4中的第12行。执行代码**Movie.new('Inception',2011).year**，得到的结果是什么？
◇ Ruby将会报错，因为**year**方法未定义。

3.5 所有的编程都是元编程

因为针对实例变量定义的简单的getters和setters非常常见，所以可以用单行**attr_accessor :title** 来替换第6～11行，用**attr_accessor :year**替换第12和13行，从而让这个例子更加像Ruby风格。**attr_accessor**并不是Ruby语言——它是一个正常的定义了动态的getters和setters的方法调用。也就是说，**attr_accessor:foo**定义了实例方法**foo**和**foo=**，它们获取和设置了实例变量**@foo**的值。相关的方法**attr_reader**仅仅定义了一个getter而没有setter，**attr_writer**则相反。

这里是 **元编程** （Metaprogramming）的一个例子——在运行时创建定义了新方法的代码。事实上，在一定意义上 **所有的** Ruby编程都是元编程，因为即使一个类定义并不像Java中一样是一个声明，而是在运行时执行的实际代码。假设这是正确的，你可能想知道在运行时是否能够 **修改** 一个类。事实上你可以通过添加或者改变实例方法或者类方法做到，即使是对Ruby的内置类也可以。例如，图3.6给出了一种做时间算法的方式，这种方式利用了标准Ruby库中的**Time**类的**now**方法，它返回了从1/1/1970开始的秒数。

在这个例子中， **重新打开** 了类**Fixnum**，这是一个之前碰到过的核心类，并且给它添加了六个新的实例方法。由于每一个新的实例方法都会返回一个整数，所以它们可以被很好的"链接"来写一些表达式例如**5.minutes.ago**。实际上，Rails包括了这个例子的更完整版本，可以做全面的时间运算。

当然不能写**1.minute.ago**，因为我们仅仅定义了一个叫做**minutes**的方法，而不是**minute**。可以用单数的名称来定义额外的方法，来复制我们已经有的方法的功能，但是这在"无重复"上做得并不好。相反，可以利用Ruby的重型元编程来构建**method_missing**。如果一个方法调用不能在接受者的类或者任何它的祖先类中被找到，那么Ruby将会对接收者尝试调用**method_missing**，并且把这个不存在的方法的名称和参数传递给接收者。**method_missing**的默认实现方法和超类类似，但是可以重载它来实现上述的时间计算方法的"单数"版本：

代码文件：ch_ruby_intro/code/ruby/time_extension_example.rb

```
#  Note: Time.now returns current time as seconds since epoch
class Fixnum
  def seconds  ; self ; end
  def minutes  ; self * 60 ; end
  def hours    ; self * 60 * 60 ; end
  def ago      ; Time.now - self ; end
  def from_now ; Time.now + self ; end
end
Time.now
# => Mon Nov 07 10:18:10 -0800 2011
5.minutes.ago
# => Mon Nov 07 10:13:15 -0800 2011
5.minutes - 4.minutes
# => 60
3.hours.from_now
# => Mon Nov 07 13:18:15 -0800 2011
```

图 3.6: 重启Fixnum类来做简单的时间算法。Unix是1970年发明的，所以它的设计者们选择了用1/1/1970午夜（格林威治标准时间）到现在的秒数来表示当前时间，由此也被称为是一个 时代 的开始。为了方便起见，如果可能的话，一个RubyTime对象通过对这个时间表示操作来响应算术运作符，尽管Ruby可以表示出过去或者将来的任意时间

代码文件：ch_ruby_intro/code/ruby/method_missing_example.rb

```
class Fixnum
  def method_missing(method_id, *args)
    name = method_id.to_s
    if name =~ /^(second|minute|hour)$/
      self.send(name + 's')
    else
      super # pass the buck to superclass
    end
  end
end
```

我们把方法ID（作为一个符号传递）转换成一个字符串，并且使用正则表达式来看字符串是否和 **hour** 、 **minute** 、 **second** 中的任何一个匹配。如果匹配，则取名称的复数形式，并且把复数形式的方法名称发送给**self**，也就是说，发送给最开始接收到调用的对象。如果不匹配，应该怎么做呢？你可能认为应该发出错误信号，但是因为Ruby有继承，所以必须考虑到祖先类可能能处理这一调用。不带参数的调用**super**传递了原始的方法调用，以及继承链完整的原始参数。

尝试用**days**方法来扩充这个例子，使得你可以写**2.days.ago**以及**1.day.ago**这样的表达式。本章的后面部分我们将会看到**method_missing** 在构建XML文档以及Rails架构中的ActiveRecord部分里增强**find**方法的作用。

小结：

- **attr_accessor**是元编程的一个例子：它在运行时创建了新的代码，在这里是针对实例变量的getters和setters。这种元编程的风格在Ruby中极为常见。
- 当既没有一个接收者也没有任何一个它的祖先类可以处理一个方法调用时，**method_missing**可以检查到这个不存在的方法的名字以及它的参数，然后或者处理这个调用，或者像**method_missing**的默认实现那样把调用传递给祖先。

■ **详细阐述：动态语言特性的陷阱**

如果**Bar**类实际上已经使用了实例变量**@fox**，但是你不小心写的是**attr_accessor:foo**（而不是**attr_accessor:fox**），那么当你写**mybar.fox**的时候就会有错误。因为Ruby不需要你声明实例变量，**attr_accessor**不能检查到这么命名的实例变量是否存在。因此，和所有的动态语言一样，我们必须小心使用它，并且不能像在Java中那样"依赖编译器"。我们将会在第8章里看到，测试驱动开发（TDD）有助于避免这样的错误。此外，在某种程度上应用程序应该是一个更大的面向服务架构的生态系统的一部分，你 **总是需要** 警惕程序依赖的其他服务中出现的运行时错误，这将会在第5章以及第12章中介绍。

■ **详细阐述：可变长度参数列表**

像**1.minute**这样的调用没有任何的参数——唯一起作用的就是接受者**1**。所以当该调用再次在第5行中的**method_missing**被调度时，不需要传递在***args**中收集的任何参数。星号是Ruby处理有0个或者更多参数的参数列表的方式：***args**是向原来的方法传递所有参数的数组，而如果没有参数要传递，它就是一个空数组。如果不确定参数列表的长度，那么第5行中读取**self.send(name+'s', *args)**总是正确的。

自我检查3.5.1. 在上述的**method_missing**例子中，第4行的正则表达式匹配中，为什么**$**和**^**是必要的?
提示：考虑一下,如果忽略了其中一个，再调用**5.milliseconds**或者**5.secondary**会发生什么
◇ 如果没有**^**来限制字符串开头的匹配，像**5.millisecond**这样的调用就会匹配，那么当**method_missing**试图重新调度**5.milliseconds**时就会出现错误。

自我检查3.5.2. 为什么**method_missing**自己不能处理没有的方法调用，总要调用**super**呢?
◇ 可能一个祖先类打算处理这个调用，但是你必须明确地通过**super**来"传递这个方法调用到链上"，从而给你的祖先类一个机会来做这件事。

自我检查3.5.3. 在图3.6中，**Time.now**是一个类方法还是一个实例方法呢?
◇ 它的接收者是一个类名（**Time**），这个事实告诉我们这是个类方法。

3.6 块：迭代器、函数式习语以及闭包

Ruby使用 **块** 和其他的语言有一点不同。在Ruby中，一个块仅仅是一个没有名称的方法，或者是编程语言术语中的 **匿名lambda表达式** 。和普通的有名字的方法一样，它有参数并且可以使用局部变量。这里给出一个简单的例子，假设**movies**是我们之前定义的**Movie**对象的一个数组：

代码文件：ch_ruby_intro/code/ruby/each_example.rb

```
1 movies.each do |m|
2   puts "#{m.title} was released in #{m.year}"
3 end
```

方法**each**是所有类似于聚合类型风格的Ruby类中都可用的一个 **迭代器** 。**each**有一个参数——块——并且把聚合类型中的每一个元素依次传给块。你可以看到，块

是由**do**和**end**括起来的部分；如果块带有参数，参数列表是包含在**do**之后的**|**管通符号**|**中。这个例子中的块带有一个参数：每次通过块时，**m**就被设置为**movies**中的下一个元素。

和有命名的方法不同，块可以访问块所在的作用域中可访问的任何变量。例如：

代码文件：ch_ruby_intro/code/ruby/block_example_1.rb

```
separator = '=>'
movies.each do |m|
  puts "#{m.title} #{separator} #{m.year}"
end
```

在上述代码中，**separator**的值在块内部可见，尽管这个变量是在块 **外部** 创建和赋值的。相反，下面这个用法就 **不** 对，因为**separator** 在**print_movies**内部不可见，因此对**each**块也是不可见的：

代码文件：ch_ruby_intro/code/ruby/block_example_2.rb

```
def print_movies(movie_list)
  movie_list.each do |m|
    puts "#{m.title} #{separator} #{m.rating}"  # === FAILS!! ===
  end
end
separator = '=>'
print_movies(movies) # FAILS!
```

在编程语言的说法中，Ruby中的块就是一个 **闭包** ：每次执行时，它可以“看到”程序文本中块出现的地方可用的整个词法作用域。换句话说，就像是块的存在创建了一个作用域的快照，每当块执行时，这些作用域都会重建。这一事实被Rails的很多提高简洁性的特性所利用，包括视图呈现（在本章的后半部分将会讲到），以及模型验证和控制器过滤器（第5章），因为它们允许分开定义即将发生什么事情（**what**）、什么时间发生（**when**）以及在应用程序的哪个结构发生（**where**）。

块就是闭包这一事实能够帮助我们理解下面的看似不寻常的例子。如果 **第一个** 引用的局部变量在块内部，那么这个变量能够被块的作用域“捕获”，并且在块退出之后就变成了 **未定义** 的。所以，举个例子，假设第二行是块范围内 **第一次** 引用**separator**，下面的这种情况将 **不能** 工作：

代码文件：ch_ruby_intro/code/ruby/block_example_3.rb

```
movies.each do |m|
  separator = '=>'  # first assignment is inside a block!
  puts "#{m.title} #{separator} #{m.rating}"   #  OK
end
puts "Separator is #{separator}"     # === FAILS!! ===
```

像Ruby这样有词法作用域的语言，变量在它们被创建的作用域内以及被这个作用域包括的所有作用域内都是可见的。在上面的代码片段中，因为**separator**是在块作用域内被**创建的**，所以它的可见性是限制在块作用域内的。

总之，**each**仅仅是聚合类型上的一个实例方法，这个聚合类型有单一参数（块），并且一次给这个块提供一个元素。块的相关应用就是聚合类型上的操作，这是一个Ruby从 **函数式编程** 里借用来的常见习语。例如，对聚合类型中的每一个元素重复一遍，可以写做：

代码文件：ch_ruby_intro/code/ruby/map_example.rb

```
new_collection = collection.map do |elt|
  2 * elt
end
```

如果句法明确无歧义，可以用花括号来划定短（在同一行内）的块，而不是用**do...end**：

代码文件：ch_ruby_intro/code/ruby/map_example_oneline.rb

```
new_collection = collection.map { |elt| 2 * elt }
```

Ruby有各种各样的这样的聚合类型运算符，图3.7列出了其中一些最有用的。稍加练习之后，你就会习惯用这些函数式习语的方式来表达对聚合类型的操作，而不是命令式循环的方式。例如，返回某个文件中的所有以元音开头的单词的列表（也就是说，令牌完全由文字字符组成，由非文字字符分隔），进行排序并且去除重复：

代码文件：ch_ruby_intro/code/ruby/method_chaining.rb

```
# downcase and split are defined in String class
words = IO.read("file").
  split(/\W+/).
  select { |s| s =~ /^[aeiou]/i }.
  map { |s| s.downcase }.
  uniq.
  sort
```

难道没有for循环？ 尽管Ruby允许**for i在聚合类型里**，但是**each**允许我们更好地利用 **动态类型** 来提高代码重用，我们很快就会看到这一点。

（回顾一下，只要在语句结束的地方没有歧义，为了可读性Ruby允许跨行分开一个语句。每行结尾的“.”能够使未完的语句清晰明了，因为一个“.”之后一定会跟随着一个方法调用。）

一般说来，如果发现在Ruby中写了显式循环，那么应该重新检查一下你的代码结构，看看这些聚合类型习语是否会让代码的简洁性以及可读性增强。

小结：

- Ruby包括了 **函数式编程** 的一些内容，例如用**map**和**sort**的方法对整个聚合类型进行操作。普遍情况下都会用这样的方法，而不是用for-循环遍历来操作聚合类型。
- **each**聚合类型方法依次返回聚合类型的一个元素，并且把它传递给 **块**。Ruby中的块仅仅能够作为方法的一个参数，就像**each**也期望块作为参数一样。
- 块是闭包：在块定义的地方对块代码可见的所有变量，在块执行的任何时间都将是可见的。
- 大多数修改聚合类型的方法，例如**reject**，实际上会返回修改之后的新副本。一些方法会有名字以**!**结束的破坏性版本，比如**reject!**。

方法	参数数目	返回一个 新的 包括……的聚合类型
c.map	1	通过应用块到**c**里的每个元素来获取元素
c.select	1	**c**的子集，其中块值为真
c.reject	1	**c**的子集，通过去掉块值为真的元素获取
c.uniq		去重复之后的**c**的所有元素
c.reverse		与**c**中元素顺序相反
c.compact		**c**中所有非-**nil**的元素
c.flatten		**c**的元素以及它的子阵列中的任何一个，递归地精简到只包含非数组元素
c.partition	1	两个聚合类型，第一个包含**c**中块值为真的元素，第二个包含哪些块值为假的元素
c.sort	2	**c**中的元素，根据块来排序，这个块带了2个参数，并且如果第一个元素应该先被排序，则返回-1；如果第二个元素应该先被排序，则返回+1；而如果两个元素都可以先排序则返回0
接下来的方法需要 **聚合类型元素** 来响应<=>；见3.7节		
c.sort		如果**sort**在块 **外** 被调用，那么元素会根据它们如何响应<=>来排序
c.sort_by	1	把块用于**c**的每个元素，并且给结果排序。例如，**movies.sort_by { |m| m.title }**根据它们的名称响应<=>的方式来排序**Movie**对象。
c.max, c.min		聚合类型中的最大的或最小的元素

图 3.7: 一些常见的聚合类型上的Ruby方法。对这些需要块的，我们给出块期望的参数的数目；如果是空的话，说明这个方法不需要块。例如，调用sort，它的块希望有两个参数，可能是这样的:c.sort { |a,b| a <=> b }。这些方法都会返回一个新的对象，而不是修改接收者。一些方法也会返回一个以!结尾的破坏性的变型，例如sort!，这些变型就直接修改了它们的参数（并且返回新值）。使用破坏性的方法时要极为小心

■ 详细阐述：XML生成器中的块和元编程

把元编程和块很好地结合起来的一个例子就是XML生成器[3] 类。（我们在2.3节中简单提到过，HTML和XML密切相关。）在接下来的例子中，第~行的XML标记是由第9~18行的Ruby代码生成的。方法调用**name**、**phone**、**address**以及使用**method_missing**来把每个方法调用转换为一个XML标签，并且块用来指示标签嵌套。

代码文件：ch_ruby_intro/code/ruby/builder_example.rb

```
1  <person type="faculty">
2    <name>Barbara Liskov</name>
3    <contact>
4      <phone location="office">617-253-2008</phone>
5      <email>liskov@csail.mit.edu</email>
6    </contact>
7  </person>
8
9  # Code that generates the above markup:
10 require 'builder'
11 b = Builder::XmlMarkup.new(:indent => 2)
12 b.person :type => 'faculty' do
13   b.name "Barbara Liskov"
14   b.contact do
15     b.phone "617-253-2008", :location => 'office'
16     b.email "liskov@csail.mit.edu"
17   end
18 end
```

自我检查3.6.1. 写出一行Ruby代码来判断一个字符串**s**是否是回文，也就是说，它向前读和向后读都是一样的。
提示：用图3.7中的方法，这里大小写不重要：比如`ReDivider`就是一个回文。

◇ **s.downcase == s.downcase.reverse**

你也许会这样写**s.reverse=~Regexp.new(s)**，但是如果**s**恰好包括正则表达式元字符比如**$**，就会失败。

3.7 mix-in和鸭子类型

你可能会很奇怪，图3.7中总结的聚合类型方法（包括不在图中的其他几个）并不是Ruby的**Array**类的一部分。事实上，它们甚至不在**Array**和其他的聚合类型继承的超类中。相反，它们采用了一个更强大的复用机制 **mix-in** 。

mix-in是一个可以添加到任何类中的相关行为的聚合类型，尽管在某些情况下类必须满足"合同"才能使用mix-in。这可能听起来和Java中的接口差不多，但是还是有两个不同点。第一，mix-in更容易复用：如果有"合同"的话，只在mix-in的文档中指定，而不是像Java接口一样正式声明。第二，mix-in都是关于如何让复用一个实现变得更容易，与Java接口不同，它对 **如何** 实现接口只字未提。

如果使用Emacs编辑器，你可以把Emacs的次要模式（自动填充、简写支持等等），看作是依赖主要模式提供的合约的mix-in，并且使用Lisp 的动态类型来允许把它们混合进任意的主要模式。

模块 是一种Ruby把一组方法打包成mix-in的方法。（Modules也有其他的用途，但是mix-in是最重要的）当一个以**include ModuleName** 的方法被包含到一个类中时，模块中的实例方法、类方法以及变量在类中都变成了可用的。

图3.7中的聚合类型方法是**Enumerable**模块的一部分，而这个模块也是Ruby标准库里的；要把**Enumerable**混合进你自己的类，只需要在类定义时声明**include Enumerable**即可。

小心！ 因为Ruby允许在运行时添加和定义方法，所以**include**不能检查出这个类是否满足模块的合约。

正如其文档[4] 所述的一样，**Enumerable**要求它要混合进的类为其提供**each**方法，因为**Enumerable**的聚合类型方法是以**each**的方式实现的。和Java的接口不同的是，这个简单的合约是混合进模块的 **唯一** 的要求；你把它混合进什么样的类并不重要，只要这个类定义了**each**实例方法，并且类和mix-in都不必提前声明它们的目的。例如，Ruby的**Array**类中的**each**方法遍历所有的数组元素，而**IO**类中的**each**方法遍历文件的每一行或者其他的I/O流。因此mix-in允许定义跨越不相关的类的行为聚合类型。

"鸭子类型"是对这种能力的一个流行的描述，因为"如果某个事物看起来像个鸭子，并且叫起来也像个鸭子，那么它就可以被当做一个鸭子".也就是说，从**Enumerable** 的角度来看，如果一个类有**each**方法，那么它就可以被当成一个集合，因此允许**Enumerable**提供一些以**each**方式来实现的其他方法。与Java的**接口**不同，mix-in不需要正式的声明；如果我们发明了一个新的mix-in，它依赖一个类来实现引用操作符**[]**，那么就可以把它混合进任意这样的类中，而不需要对类本身进行修改。当Ruby编程者说某个类"叫起来像一个**Array**"，它们通常的意思就是说，它并不一定就是一个**Array**，也不一定是**Array**的后代，但是它们可以像**Array**一样响应大多数的方法，因此在**Array**被用到的任何地方它们都可以用。

因为**Enumerable**可以把图3.7 （以及一些其它的）中的所有方法都提供给实现了**each**的任意类，为了方便，所有的"嘎嘎叫起来像聚合类型" 的Ruby类都混合进了**Enumerable**。**sort** （没有块）方法，**max**以及**min**也需要聚合类型中的 **元素**（并不是聚合类型本身）能够响应**<=>** 方法，它会返回-1、0或者1，这取决于它的第一个参数是小于、等于或者大于第二个参数。即使聚合类型的元素不能响应**<=>**，仍然可以混入**Enumerable**；但是你不能用**sort**、**max**、或者**min**。相比而

<=> 有时候也被称作**飞船操作符**，因为有些人认为它看起来像一个飞碟。

言，在Java中，每一个实现了**Enumerable**接口的聚合类型类，都必须满足元素具有可比性，不论这个特性是否需要。

第8章将介绍mix-in、开放类以及**method_missing**的结合，能够让你用RSpec工具写出非常具有可读性的单元测试。

小结：

- **mix-in** 是相关行为的集合，这些行为可以被添加到任何满足mix-in的合约的类中。例如，**Enumerable**是一个可枚举的聚合类型上行为的集合，而这些聚合类型的类定义了**each**迭代器。
- Ruby用模块来给mix-in分组。一个模块通过在**class ClassName**语句之后添加**include ModuleName**来混合进一个类。
- 一个类实现了另一个类的一些行为特征，可能通过mix-in方式，有时候也被说成"嘎嘎叫起来"像它的那个类。Ruby允许mix-in不进行静态类型检查，这种方式也被称为 **鸭子类型** 。
- 和Java中的接口不同，mix-in不需要正式的声明。但是因为Ruby没有静态类型，所以你需要保证包括mix-in的类能够满足mix-in的文档中的条件，否则就会有运行时错误。

■ **详细阐述：Time类中的鸭子类型**

Ruby可以表示出过去或者将来的任意长的时间，可以使用时区，并且也可以处理非公历日历系统。然而在3.5节中，当一个**Time**对象接收到带有整数参数的**+**方法调用时，它将试着返回本身的一个表示，这个表示和Unix（从一个时间开始的秒数）表示兼容。换句话说，虽然**Time**对象不是一个简单的整数，但在必要的时候，它"叫起来像"整数。

自我检查3.7.1. 假如在不支持**each**方法的类**Foo**中混合了**Enumerable**，那么当你调用**Foo.new.map { |elt| puts elt }**会出现什么错误?

◇ **Enumerable**中的**map**方法尝试对接收者调用**each**，但是因为新的**Foo**对象没有定义**each**，Ruby将会给出一个未定义方法错误。

自我检查3.7.2. 哪一个语句是正确的，请给出原因：

（a）**include 'enumerable'** （b）**include Enumerable**

◇ （b）是正确的，因为**include**需要一个module的名字，这就像类名一样是一个常量而不是一个字符串。

3.8 用Yield构造自己的迭代器

虽然Ruby为**Array**这样的内置聚合类型类定义了**each**，但是也可以用**each**定义自己的迭代器。让迭代成为编程语言的首要特性的想法第一次出现在Barbara Liskov发明的CLU语言中。Ruby的**yield**让你可以定义自己的带有块的**each**方法，这给聚合类型提供了一种优雅简洁的管理自己遍历的方式。

代码文件：ch_ruby_intro/code/ruby/yield_example.rb

```
# return every n'th element in an enumerable
module Enumerable
  def every_nth(count)
    index = 0
    self.each do |elt|
      yield elt if index % count == 0
      index += 1
    end
  end
end

list = (1..10).to_a # make an array from a range
list.every_nth(3) { |s| print "#{s}, " }
# => 1, 4, 7, 10,
list.every_nth(2) { |s| print "#{s}, " }
# => 1, 3, 5, 7, 9,
```

图 3.8：使用Ruby的yield的一个例子，它是基于一个CLU语言引入的结构。注意我们在Enumerable module中定义了every_nth，正如3.7节所描述的一样，大多数聚合类型都混合了Enumerable module

不要混淆 yield 的使用和操作系统中无关用法，操作系统中一个线程或进程放弃CPU来 **让步于** 另外一个。

图3.8显示了这种不常用的结构是怎么工作的。当一个包括了**yield**的方法被调用时，它就会开始执行，直到到达**yield**语句；在这时，控制转移到传递给方法的块上。如果**yield**有参数，这些参数都会变成块的参数。

yield的常见用法就是实现迭代器，例如**each**和**every_nth**。和Java不同，你需要传递某个类型的聚合类型才能创建一个迭代器，然后重复地调用**while (iter.hasNext())** 和**iter.getNext()**，Ruby迭代器允许控制结构“由内到外”，并且让数据结构管理 它们自己的 循环。

yield使一些复用成为可能，例如，当需要把一些自定义功能加到常用功能中时。例如，一个创建HTML网页的应用程序，大部分页面都使用了这样一种标准HTML模板，只有第8行略有不同：

代码文件：ch_ruby_intro/code/ruby/html_example.html

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4    <title>Report</title>
5    </head>
6    <body>
7      <div id="main">
8       ...user-generated content here...
9      </div>
10   </body>
11 </html>
```

在大多数语言中，可以把第1~7行和9~11行生成样板的代码封装成方法**make_header**和**make_footer**，然后每个想要生成一个页面的方法都按照下面来做：

代码文件：ch_ruby_intro/code/ruby/yield1.rb

```
def one_page
  page = ''
  page << make_header()
  page << "Hello"
  page << make_footer()
end
def another_page
  page = ''
  page << make_header()
  page << "World"
  page << make_footer()
end
```

这段代码看起来有点重复，可以用一个单一的方法来封装这两个调用：

代码文件：ch_ruby_intro/code/ruby/yield1a.rb

```
def make_page(contents)
  page = ''
  page << make_header()
  page << contents
  page << make_footer()
end
#
def one_page
  make_page("Hello")
end
def another_page
  make_page("World")
end
```

但是在第2章里我们了解到有用的设计模式源于把 **变化的事物和不变的事物分开** 的需求。**yield**提供了一个更好的方式去封装“围绕”用户内容的相同的部分——用这些相同内容自己的方法：

代码文件：ch_ruby_intro/code/ruby/yield2.rb

```
1  def make_page
2    page = ''
3    page << make_header()
4    page << yield
5    page << make_footer()
6  end
7  def one_page
8    make_page do
9      "Hello"
10   end
11 end
12 def another_page
13   make_page do
14     "World"
15   end
16 end
```

在这个例子中，当**one_page**调用**make_page**时，第4行的**yield**将控制返回到第9行的块。块执行后的返回值（**"Hello"**）作为**yield** 的结果返回到第4行，并且被添加到**page**（用**«**运算），在这之后**make_page**继续执行。

可以利用Ruby的单行块的习惯用法来简写：

代码文件：ch_ruby_intro/code/ruby/yield3.rb

```
def make_page
  make_header << yield << make_footer
end

def one_page
  make_page { "Hello" }
end
def another_page
  make_page { "World" }
end
```

我们可以看到，**yield**就是Rails实现视图的HTML模板的方法：每页开头和结尾的共同的HTML代码呈现出来后，**yield**用来呈现两者之间的特定的页面内容。在第11章中，我们将会了解到，在把变化和不变的内容分开时，块和Factory设计模式的结合如何让代码变得非常简洁漂亮。

有了这个对Ruby的最鲜明的特点的简单介绍，我们就可以去接触Rails架构了。

小结：

- 在把块作为参数的一个方法的主体中，**yield**将控制转移到块，并且选择性地传给它一个参数。
- 因为块是一个闭包，所以即使给这个块让步的方法是在一个完全不同的作用域内执行的，块作用域仍是块被定义时的那个作用域。
- Yielding是迭代器背后的通用机制：迭代器就是一个简单的方法，这个方法遍历某个数据结构，并且用**yield**每次传一个元素给迭代器的接收者。

自我检查3.8.1. 在图3.8中，在第5和6行中**every_nth**是把**elt**作为一个实例变量名来使用的。假设在第13行的块中用**elt**代替**s** 作为局部变量的名称。这个改变将会造成影响吗？如果有的话，原因是什么？

◇ 不会有影响。**every_nth**和我们传递给它的块在不同的作用域内运行，所以局部变量名称**elt**不会产生“冲突”。

3.9 谬论和陷阱

陷阱：以Java的方式来写Ruby程序。

需要一个过程去学习一个新语言的习惯以及它和其他的语言的不同之处。对Java编程者很常见而对Ruby编程者来说是新的例子包括：

- 以类型转换的方式思考而不是方法调用的方式：**100.0 * 3**不是把3转换为浮点数，而是调用**Float#***。
- **a.b**读作“对象**a**的属性**b**”而不是“调用对象**a**上的方法**b**”。
- 以类和传统的静态类型思考，而不是鸭子类型。当对一个对象调用方法时，或者做一个mix-in，只有对象是否能够响应这个方法会起作用。对象的类型或者类都是无所谓的。

- 写出显式的for循环，而不是用**each**这样的迭代器以及通过像**Enumerable**这样的mix-in来利用迭代器的聚合类型方法。请使用函数式习语，例如**select**、**map**、**any?**、**all?**等等。
- 把**attr_accessor**当作属性的声明。**如果**你想让一个属性公开可读可写，这个简写可以减少你的工作。但是你完全不需要以任何的方式"声明"一个类属性（实例变量是存在的就足够了），并且十有八九某些属性**不应该**是公开可见的。要主动抵制像在Java中声明类属性一样声明每个**attr_accessor**。

陷阱：认为符号和字符串是可互换的

虽然很多的Rails方法构建时支持字符串或者符号，但是一般情况下这两者不可互换。一个需要字符串的方法可能会抛出一个错误，如果你给的是一个符号，也可能直接崩溃。例如，**['foo','bar'].include?('foo')**是正确的，而**['foo','bar'].include?(:foo)**是合法却是错误的。

陷阱：当你想调用一个局部方法时却命名了一个局部变量。

假设类**C**定义了一个方法**x=**。在**C**的一个实例方法中，写**x=3**将不会得到你调用参数为3的**x=**方法时期望得到的结果；相反，它会给3设置一个局部变量**x**，这可能并不是你想要的。为了得到需要的结果，应该写**self.x=3**，这个方法调用就很明确了。

陷阱：混淆require和include。

require加载任意一个Ruby文件（通常是主文件），而**include**表示混合进一个module。在这两种情况下，Ruby有自己的方法来定位包含代码的文件；Ruby文档描述了**$LOAD_PATH**的使用，但是如果你用Rails作为自己的架构，以及用Bundler去管理gem库，那么你应该几乎不需要对**$LOAD_PATH**直接操作。

3.10 结束语：惯用语言使用

> 丑陋的程序就像丑陋的吊桥：它们比漂亮的桥更容易坍塌，因为人类（尤其是工程师）认为美感和我们处理以及理解复杂性的能力是紧密相关的。一门很难写出优雅代码的语言，也很难写出好的代码。
>
> ——Eric S. Raymond

如果对Lisp或者Scheme等相关的语言没有了解的情况下，接触Ruby，那么很多函数编程习惯用语对你而言可能就是新鲜的。除非你很熟悉JavaScript，你可能之前没有用过闭包。并且除非你了解**CLU**，否则可能需要一段时间去适应Ruby的**yield**。

编程人员之间流传着一个古老的说法，"你可以用任何语言来写Fortran。"这个说法可能对Fortran并不公平——你可以用任何语言写出好的或者坏的代码——但是这句话的意图就是不鼓励把编程习惯从一种语言带到另外一种并不合适它的语言里，因此在新的语言里不能使用这种机制，尽管这种机制可能会提供一个更加完美的解决方案。

我们的建议就是坚持使用一门新的语言，直到你习惯了它的常见用法。要首先考虑是否有一个更加Ruby的方式来写出你需要的代码，而不是把代码从其他语言直译过来。当我们在第6章里讲到JavaScript时，我们会重复这个建议的。

学习使用一门新的语言，并且使用它的大部分的习惯用法，这对软件专业人士来说是一项很重要的技能。这个任务并不容易，但是我们希望在我们在Ruby和Javascript 中专注于独特美丽的特点能够唤起人们的求知欲，而不是在这里无病呻吟，并且希望你能够认识到使用各种各样的专门工具的价值以及为每一个新工作选择一个最有效率的工具。

3.11 深入学习

- Ruby[5]在线文档给出了这门语言的细节、类以及Rails架构。
- 很多Ruby初学者学习**yield**很困难，因为它在Java、C或者C++中没有对应的模块（尽管最新版本的Python和JavaScript有类似的机制）。维基百科上的文章**coroutines** 在**yield**支持的一般的协程机制方面给出了一些很好的例子。
- Ruby标准库[6]上的文档值得花一些时间去略读。其中有一些非常有用的类，包括**IO**（文件和网络I/O，包括CSV文件）、**Set** （聚合类型运算符，比如集合差、集合相交等等）、以及**Time** （表示时间的标准类，和**Date**相比我们更推荐它，即使你只需要表示不带时间的日期）。
- Programming Ruby[7] 和 **The Ruby Programming Language** （Flanagan and Matsumoto 2008），由Ruby的发明者Yukihiro “Matz” Matsumoto合著，都是Ruby方面权威的参考书目。

D. Flanagan and Y. Matsumoto. **The Ruby Programming Language**. O'Reilly Media, 2008. ISBN 0596516177. URL `http://www.amazon.com/Ruby-Programming-Language-David-Flanagan/dp/0596516177`.

注释

[1]`http://validator.w3.org`
[2]`http://code.google.com/apis/searchappliance/documentation/50/xml_reference.html#request_parameters`
[3]`http://builder.rubyforge.org/`
[4]`http://ruby-doc.org/core-1.9.3/Enumerable.html`
[5]`http://ruby-doc.org/core-1.9.3/`
[6]`http://ruby-doc.org`
[7]`http://ruby-doc.org/docs/ProgrammingRuby`

3.12 建议的项目

OO和类

练习3.1. 对象**5**有多少祖先类?
提示：使用方法链去跟踪超类链直到**BasicObject**

练习3.2. 已知调用**BasicObject**上的**superclass**时返回**nil**，否则返回非**nil**值，写一个Ruby方法，当传递过去任何一个对象时，都将打印出对象的类和它的祖先类，直到遇到**BasicObject**。

练习3.3. Ben Bitdiddle问道："如果**i**是Ruby中的整数，**f**是Ruby中的一个浮点数，我写下**i+f**，在这种情况下**i**会转换成一个浮点数还是**f** 转换成一个整数呢？"解释一下，为什么在Ruby中Ben的问题是不合语法规范的。

练习3.4. 从练习3.3的答案中得到了启发，Ben现在发现，$i+=f$是合法的Ruby表达式，他的问题是："$+=$在Ruby中是一个单独的操作符呢，还是一个纯粹的语法糖，对$i=i+f$表达式来说？"设计并进行一个实验来得到这个问题的答案。

元编程

练习3.5. 以3.5节中的例子为基础，利用**Time**的鸭子类型来定义一个**at_beginning_of_year**方法，这个方法让你能写出如下代码：

代码文件：ch_ruby_intro/code/ex/time_at_beginning_of.rb

```
Time.now.at_beginning_of_year + 1.day
# => 2011-01-02 00:00:00 -0800
```

提示1： **Time** 文档[1]表明**local**类方法可以用来创建一个新的**Time**对象，这个对象指定了年月日。

提示W： 上面代码中**at_beginning_of_year**的接收者是**now**，和3.5节中的例子一样。但是和那个例子不同的是，仔细想想你应该让**now**"怎样叫"。

练习3.6. 定义**attr_accessor_with_history**方法，这个方法提供了和**attr_accessor**一样的功能，但是同时也跟踪属性曾有过的每一个值。

代码文件：ch_ruby_intro/code/ex/attr_accessor_with_history.rb

```
class Foo
  attr_accessor_with_history :bar
end
f = Foo.new      # => #<Foo:0x127e678>
f.bar = 3        # => 3
f.bar = :wowzo   # => :wowzo
f.bar = 'boo!'   # => 'boo!'
f.history(:bar)  # => [3, :wowzo, 'boo!']
```

mix-in和迭代器

练习3.7. **Enumerable**模块包括迭代器**each_with_index**，其中的每一个枚举变量都有一个索引，索引从0开始（回想一下**Enumerable**是默认混合进Ruby的内置集合类的）：

代码文件：ch_ruby_intro/code/ex/each_with_index.rb

```
%w(alice bob carol).each_with_index do |person,index|
  puts ">> #{person} is number #{index}"
end
>> alice is number 0
>> bob is number 1
>> carol is number 2
```

在模块**Enumerable**中建立一个迭代器**each_with_custom_index**，这个模块会让你决定起始值和索引步长：

代码文件：ch_ruby_intro/code/ex/each_with_index_1.rb

```
%w(alice bob carol).each_with_custom_index(3,2) do |person,index|
  puts ">> #{person} is number #{index}"
end
>> alice is number 3
>> bob is number 5
>> carol is number 7
```

练习3.8. 回忆一下，斐波那契数列的头两个数字是1和1，后续的每个斐波那契数都是之前两个数的和。建立一个类，这个类是能够返回前n个斐波那契数的一个迭代器。你应该可以这样使用这个类：

代码文件：ch_ruby_intro/code/ex/fib.rb

```
# Fibonacci iterator should be callable like this:
f = FibSequence.new(6) # just the first 6 Fibonacci numbers
f.each { |s| print(s,':') }  # => 1:1:2:3:5:8:
f.reject { |s| s.odd? }      # => [2, 8]
f.reject(&:odd?)             # => [2, 8] (a shortcut!)
f.map { |x| 2*x }            # => [2, 2, 4, 6, 10, 16]
```

提示：只要类中的对象实现了**each**，就可以混合进去**Enumerable**，从而得到**reject**、**map**等等。

练习3.9. 实现一个迭代器**each_with_flattening**，表现如下：

代码文件：ch_ruby_intro/code/ex/each_with_flattening.rb

```
[1, [2, 3], 4, [[5, 6], 7]].each_with_flattening { |s| print "#{s}," }
>> 1, 2, 3, 4, 5, 6, 7
```

关于迭代器的接收者必须做什么样的假设？而关于接收者的元素又应该做什么样的假设？

练习3.10. 给**Enumerable**模块增加一个新的迭代器**each_permuted**，它以随机顺序返回集合中的元素。这个迭代器假设集合能够响应**each**，但是不能对元素有其他任何的假设。
提示：你可能会用到Ruby标准库中的**rand**方法。

练习3.11. 有序二叉树中每个节点都有一个元素值，并且最多有两个子节点，每一个子节点本身都是一个有序二叉树，并且某个节点的左子树的元素小于右子树的任何一个元素。

定义**BinaryTree**集合类，这个类提供了实例方法<<（插入元素）、**empty?**（如果树没有元素的话返回真）以及**each**（标准迭代器以任何你想要的顺序反向产生每个元素）。

练习3.12. 增强有序二叉树的类，使它能够提供如下的方法，每一个方法处理一个块：**include?(elt)**（如果树包括**elt**的则为真）、**all?**（给定的块对所有元素都是真，则结果为真）、**any?**（给定的块对任何元素为真则结果为真）、**sort**（把元素按照带有两个参数的block分类，这个block返回-1、0、1，具体依赖于第一个元素是小于、等于还是大于第二个元素）。

提示：一行代码就可以实现这个功能。

练习3.13. 和3.5节中的**days.ago**例子类似，定义欧元、美元、日元之间的适当的转换，使得你可以输入如下的转换：

代码文件：ch_ruby_intro/code/ex/currency_conversion_exercise.rb

```
# assumes 1 Euro=1.3 US dollars, 1 Yen=0.012 US dollars
5.dollars.in(:euros)  # => 6.5
(1.euro - 50.yen).in(:dollars)  # => 0.700
```

练习3.14. 在下面的这些方法中，哪一个能够以你期待的方式引起突变?

代码文件：ch_ruby_intro/code/ex/mutations.rb

```
def my_swap(a,b)
  b,a = a,b
end

class Foo
attr_accessor :a, :b
  def my_swap_2()
    @b,@a = @a,@b
  end
end

def my_string_replace_1(s)
  s.gsub( /Hi/, 'Hello')
end

def my_string_replace_2(s)
  s.gsub!( /Hi/, 'Hello')
end
```

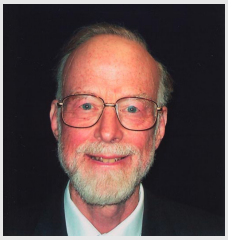
练习3.15. 用**humanize**方法来扩展**Time**类，而**humanize**方法能打印出信息短语，描述出离整15分钟最近的时间，我们使用12小时模式，并注意午夜这一特殊处理的情况:

代码文件：ch_ruby_intro/code/ex/humanize_time.rb

```
>>  Time.parse("10:47 pm").humanize
# => "About a quarter til eleven"
>>  Time.parse("10:31 pm").humanize
# => "About half past ten"
>>  Time.parse("10:07 pm").humanize
# => "About ten"
>>  Time.parse("23:58").humanize
# => "About midnight"
>>  Time.parse("00:29").humanize
# => "About 12:30"
```

4 SaaS架构：Rails简介

Charles Antony Richard Hoare
（1934——,基本上所有人都称他"Tony"）
在1980年因为在"编程语言的定义和设计上的重要的贡献" 而获得了图灵奖。



构建软件设计有两种方式：一种方式是让它很简单，使得它 **明显地没有不足**，另一种方式就是让它变得很复杂，使得它 **没有明显的不足** 。第一种方法远比第二种困难.....可靠性的代价就是最大限度地追求简洁性。

——Tony Hoare

4.1 Rails基础：从0到CRUD **94**
4.2 数据库和迁移 **98**
4.3 模型：Active Record基础 **100**
4.4 控制器和视图 **104**
4.5 调试：当出现错误时 **110**
4.6 表单提交：New和Create **112**
4.7 重定向和Flash **114**
4.8 完成CRUD：编辑/更新和销毁 **117**
4.9 谬论和陷阱 **121**
4.10 结束语：SOA的设计 **122**
4.11 深入学习 **122**
4.12 建议的项目 **123**

重要概念

本章主要思想是Rails是如何简化SaaS应用的创建的。

- Rails应用了客户端-服务机、三层体系结构以及模型-视图-控制器模式，这三者在SaaS应用中都是常见的。
- Rails的ActiveRecord包使用Ruby的元编程以及 **惯例优先配置原则** ，使你只要遵循某些特定的类和变量的命名规，就可以不用再写任何代码去在你的模型上进行如下的基本操作：Create、Read、Update和Delete （CRUD）。
- Rails的ActionView和ActionController包有助于创建网页，处理填写表单，并且设置把URI映射到控制器行为的 **路由** （应用程序中的代码）。
- 一个正确构建的Rails应用程序可以很容易地应用到面向服务的体系结构中，同外部设备而不是同使用浏览器的人们交流。
- 调试SaaS需要了解在SaaS请求执行的过程中可能出现错误的不同的地方，并且使得这个信息对开发人员可见。

所有的这些Rails设施旨在精简在面向服务体系结构中工作的应用程序的创建，以及利用成熟的SaaS 设计模式。

4.1 Rails基础：从0到CRUD

正如第2章所介绍的，Rails是一个SaaS应用架构，它定义了一个特定架构来组织应用的代码，并且提供了一个Rails应用程序服务器，例如Rack的接口。应用程序服务器等待一个Web浏览器来连接应用程序，并且把每一个即将到来的请求（URI和HTTP方法）映射到应用程序的某个控制器中的一个特定的行为。Rails包括架构本身以及一个新的命令`rails`，这个命令是用来设立和操作Rails应用程序的。Rails对MVC支持的核心是由三个主要的模块组成：创建模型的ActiveRecord、创建视图的ActionView以及创建控制器的ActionController。

使用第2章中的模型-视图-控制器的解释作为参考框架，我们将会从0开始创建第2章中描述的Rotten Potatoes应用，以维护一个简单的电影信息数据库。我们将以如下的顺序，用单一模型来简单地访问基本的Rails应用中的每个“移动的部分”：

(1) 创建新应用的骨架

(2) 路由

(3) 数据库和迁移

(4) 模型和Active Record

(5) 控制器、视图、表单和CRUD

先登录到bookware VM，进入一个更加简单的目录，例如Documents（`cd Documents`），然后用`rails new myrottenpotatoes -T`创建一个新的空Rails应用。如果一切进展顺利，将会看到几条以“Your bundle is complete”结束的文件被创建的消息。现在可以打开新创建的`myrottenpotatoes`目录，在新应用里是名为 **app root** 的目录。从现在开始，除非特别提到，否则所有的文件名都和app root相关。在进一步深入之前，花一点时间来检查一下新应用目录`myrottenpotatoes`的内容，如图4.1所示，熟悉一下所有Rails应用中通用的目录结构。

本书中的示例代码都包含在了本书的配套光盘中，你可以方便的通过复制粘贴运行这些代码。

消息“Your bundle is complete”是指自动为应用程序创建的`Gemfile`。虽然Ruby的标准库包括大量有用的类的聚合类型[2]，但是Rubygems是一个管理着外部用户提供的Ruby库或者 **gem** 的系统。Bundler是一个预装了bookware的gem，在应用程序的根目录中寻找`Gemfile`，`Gemfile`不仅指定了应用程序依赖的是什么样的gem，还指定了这些gem的版本。你可能想不到尽管你还没有写任何的应用程序代码，但是这个文件里已经有了一些gem名称，这是因为Rails本身就是一个gem，并且也依赖于其他几个gem。例如，如果在一个编辑器中打开`Gemfile`，你可能会看到`sqlite3` 已经列出来了，这是因为默认的Rails开发环境需要使用SQLite3数据库。

通过添加如下的内容（文件里的任何地方，只要不在以**group**开头的块内即可）来编辑`Gemfile`。

代码文件：ch_rails_intro/code/gemfile_edits.rb

```
# use Haml for templates
gem 'haml'
# use Ruby debugger
group :development, :test do
  gem 'ruby-debug19'
end
```

这种变化做了两件事情。第一，它指出了我们将会使用Haml模板系统而不是内置的`erb`。第二，它指出了我们想在开发和测试的时候使用交互式调试器**ruby-debug19**，而不是在产品阶段。

Gemfile	这个应用程序使用的Ruby的gem列表（库）（第3章）
Rakefile	自动维护和部署的命令（第12章）
app	应用程序
app/models	模型代码
app/controllers	控制器代码
app/views	视图模板
app/views/layouts	应用程序里所有视图使用的页面模板（详见正文）
app/helpers	简化视图模板的助手方法
app/assets	静态资产（JavaScript、图像、样式表）
config	基本的配置信息
config/environments	运行在开发和产品模式里的设置
config/database.yml	开发和产品模式里的数据库配置
config/routes.rb	URI映射到控制器行为
db	描述数据库模式的文件
db/development.sqlite3	用于SQLite数据库开发的数据存储
db/test.sqlite3	用于运行测试的数据库
db/migrate/	迁移（描述数据库模式的改变）
doc	生成的文档
lib	M、V、C共享的额外应用程序代码
log	日志文件
public	Web服务器服务的错误页面
script	开发工具，不是应用程序的一部分
tmp	运行时保存的临时数据

图 4.1: 对应用的模型，视图以及控制器的实际应用逻辑子目录来说，**Rails**工程的标准的目录结构包括一个app目录，它给出了包括在工程文件的安排里**Rails**显示**MVC**体系结构的方式

一旦对Gemfile做了这些更改之后，再运行bundle install --without production，它会检查Gemfile指定的任何的gem是否消失以及是否需要安装。在这种情况下不需要任何安装，因为我们已经预加载了在bookware VM里需要的大多数的gem，所以像之前一样，你应该看一下“Your bundle is complete”。margin图标表明，Bundler将会是我们碰到的第一个 **自动化重复** 的例子：把它们列在Gemfile里并且让Bundler自动安装它们以确保可以在各种各样的环境中都能一致地重复这个任务，而不是手动安装应用程序所需要的gem，这样可以减少可能成为应用程序误差源的任务中发生的错误。这一点很重要，因为当配置应用时，这个信息用来让配置环境和开发环境匹配。

Address already in use? 如果看到这个错误，那么你已经有了一个应用程序服务器在默认端口3000上监听，所以如果有必要，找到你启动时的终端窗口，然后输入Control-C来停止它。

用rails server来启动应用程序，打开浏览器网址http://localhost:3000。回顾第2章，一个仅仅指定了主机名和端口的URI将会抓取主页。大多数的Web服务器遵循这个约定，除非应用程序另有指定，否则主页为index.html，并且实际上你看到的欢迎页面存储在public/index.html——新的Rails应用程序通用的欢迎页面。

如果你现在访问http://localhost:3000/movies，你可能会碰到一个来自于Rails的路由错误。实际上，你可以验证 **任何** 添加到这个URI中的内容都会导致这个错误，这是因为我们还没有指定任何 **路由** 去把URI映射到应用程序方法上。尝试rake routes并验证，可以看到和第2章中的结果不同，它不会打印任何东西，因为在全新的应用程序里还没有路由。（你可能想打开多个终端窗口，使得当你使用其它命令时应用程序可以一直运行。）更加重要的是，使用编辑器来打开文件log/development.log然后观察记录的错误信息；这是当出错时你查看详细错误信息的地方。我们将会在4.5节介绍其他的一些查错和调试的技术。

为了解决这个错误需要增加一些路由。因为我们起初的目标就是在数据库里存储电影信息，可以利用Rails的一个捷径，在模型上它为四个基本的CRUD行为（Create、Read、Update、Delete）创建了RESTful路由。（回顾一下，RESTful路由指定了独立请求，这些请求给出了执行什么操作以及在什么实体或者资源上执行。）编辑config/routes.rb，它是由rails new 命令自动生成的文件并且被大量注释过。用以下的内容替换文件中的内容（这个文件主要是注释，所以实际上并不用删太多内容）：

代码文件：ch_rails_intro/code/routes.rb

```
Myrottenpotatoes::Application.routes.draw do
  resources :movies
  root :to => redirect('/movies')
end
```

非常重要： 另外，如果文件public/index.html存在，那么请**删除**它。保存文档然后再次运行rake routes，可以看到因为改变了routes.rb，输出的第一行表明URIGET /movies将会尝试调用电影控制器的索引行为；很快就会看到，这个和表格中的大部分其他路由一样，都是**resources :movies**行的结果。根路由'/'，RottenPotato的“主页”，通过一个叫做 **HTTP重定向** 的机制，将会把我们带到主要的Movie列表页面。

Symbol or string?和很多的Rails方法一样，**resources 'movies'**也能工作，但是习惯上，一个符号表示一组固定选择中的一个的值，而不是任意一个字符串。

使用惯例优先配置原则，Rails希望这个控制器的行为在类**MoviesController**就被定义了，并且如果这个类在应用程序开始时间内还没有被定义，Rails将会尝试从文件app/controllers/movies_controller.rbk 加载它。果然，如果你现在在浏览器里重新加载页面http://localhost:3000/movies，应该会看到一个不一样的错误：uninitialized constant MoviesController。这是个好消息：Ruby的类名仅仅是一个指向类对象的常量，所以实质上Rails 是在抱怨它找不到**MoviesController**类，表明我们的路由正确工作了！和之前一样，这个错误信息和其他信息都存在日志文件log/development.log中。

了解了列表中的前两个步骤——设置应用程序骨架和创建某些初始路由——我们可以继续建立存储模型的数据库了，即MVC中的“M”。

小结：
使用如下命令来设置一个新的Rails应用程序。

- `rails new`建立新的应用程序；`rails`命令也有一些子命令用WEBrick（`rails server`）来在本地运行应用程序、以及其他的管理任务。
- Rails和你的程序依赖的其他的gem（添加了Haml模板系统和Ruby调试器）列在了应用程序的`Gemfile`中，Bundler利用这个`Gemfile`在开发模式和产品模式中都可以自动化为应用程序创建一致的环境的过程。
- 为了在`config/routes.rb`里添加路由，Rails路由系统提供的单行`resources`方法允许我们为RESTful资源上的CRUD行为设置一系列相关的路由。
- 当出现错误时，`log`目录下的日志文件收集了错误信息。

■ **详细阐述：自动重新加载应用**

你可能已经注意到在改变`routes.rb`之后，你不必停止然后重启应用程序来使得修改生效。在开发模式中，Rails在每一个新的请求上都会重新加载应用程序的所有类，从而使得修改立即生效。在产品模式中这会引起严重的性能问题，所以Rails提供了在开发和产品模式之间改变各种各样的应用程序行为的方式，我们将会在4.2节里看到。

Route	Example URI and behaviors
get ':controller/:action/:id' or **get 'photos/preview/:id'**	`/photos/preview/3` method: **PhotosController#preview** **params[]: {:id=>3}**
get 'photos/preview/:id'	`/photos/look/3?color=true` Error: no route will match (`look` doesn't match **preview**)
get 'photos/:action/:id'	`/photos/look/3?color=true` method: **PhotosController#look** (`look` matches **:action**) **params[]: {:id=>3, :color=>'true'}**
get ':controller/:action/:vol/:num'	`/magazines/buy/3/5?newuser=true&discount=2` method: **MagazinesController#buy** **params[]: {:vol=>3, :num=>5, :newuser=>'true', :discount=>'2'**

图 4.2: 和Elaboration里解释的一样，路由可以包括“通配符”标记，例如:controller和:action，它们决定了控制器以及将会引发的动作。任何其他的以:开头的标记，加上在URI中编码过的其他参数，在params哈希中都将变为可用的

■ **详细阐述：非资源型路由**

Rails提供的捷径**resources :movies**为CRUD创建了RESTful路由，但是任何一个重要的应用程序都将会有很多除了CRUD之外的控制器行为。Rails Routing from the Outside In guide[3]有更多的细节，但是设置路由的一种方式就是把URI的组件直接映射到控制器以及使用通配符的行为名称上，如图4.2所示。

自我检查4.1.1. 回想一下你第一次创建应用程序时看到的通用Rails欢迎页面。在`development.log`文件中，当`Started GET "assets/rails.png"`行打印出来时发生了什么？
提示：回想一下，2.3节中描述的当要呈现一个包含嵌入式资源的页面时需要的步骤。
◇ 浏览器请求在欢迎页面嵌入Rails的标志的图像。

自我检查4.1.2. 为了让应用程序使用一个特殊的Ruby gem，我们必须做哪两个步骤？
◇ 必须在`Gemfile`中添加一行来加入gem，并且再运行一下`bundle install`。

4.2 数据库和迁移

Rails应用程序的持久层（如图2.7所示）默认使用了关联数据库（RDBMS），原因我们在第2章里讨论过。令人惊讶的是，使用Rails的过程中你不需要知道关系数据库的详细内容，尽管当应用程序变得更加复杂时它确实能够起到帮助。就像我们使用“精简的”Web服务器WEBrick来开发一样，Rails应用程序默认配置使用SQLite3，这是一个“精简的”关系数据库。在产品中你可能使用一个产品成熟级别的数据库，例如MySQL、PostgreSQL或者Oracle。

但是比“轻量级”更重要的是，你不想用产品数据库来开发或者测试应用程序，因为代码中的错误可能会不小心毁坏有价值的客户资料。所以Rails定义了三种 **环境** ——产品、开发、和测试——它们都管理着自己的数据库副本，这在`config/database.yml`中有详细分配。测试 数据库完全被测试工具管理，并且绝对不能手动修改：在每一个测试开始时，都会被擦除干净然后重新填充，我们将会在第8章里看到这一点。

文件`db/development.sqlite3`中的命令`rails new`给我们创建了一个空的数据库，就像**config/database.yml**中指定的一样。我们需要创建一个电影信息的表。我们可以使用`sqlite3`命令行工具或者SQLite GUI工具来手动完成这件事请，但如何在配置时在产品数据库中创建一个表呢？再次输入相同的命令行并不符合无重复原则（DRY），并且很难记住准确的命令。更进一步说，如果产品数据库不是SQLite3（几乎肯定就是这样），具体的命令可能有所不同。并且如果在未来对数据库添加更多的表或者有其他的修改时，也会面临相同的问题。

一个更好的选择就是 **迁移** ——以一致和可重复的方式来改变数据库模式（表格和列的布局）的便携式脚本，就和Bundler以一致可重复的方式使用Gemfile来识别和安装必要的gem（库）一样。使用迁移来改变模式有三个步骤：

(1) 创建一个迁移来描述要做哪些变化。和`rails new`一样，Rails提供了生成样板代码的迁移 **生成器**，加上各种各样的辅助方法来描述这个迁移。

(2) 把迁移应用到开发数据库中。Rails为此定义了一个`rake`任务。

代码文件：ch_rails_intro/code/initial_migration.rb

```
class CreateMovies < ActiveRecord::Migration
  def up
    create_table 'movies' do |t|
      t.string 'title'
      t.string 'rating'
      t.text 'description'
      t.datetime 'release_date'
      # Add fields that let Rails automatically keep track
      # of when movies are added or modified:
      t.timestamps
    end
  end

  def down
    drop_table 'movies' # deletes the whole table and all its data!
  end
end
```

图 4.3：创建了一个新的Movies表格的一个迁移，指定所需要的字段和类型。ActiveRecord::Migration类（所有的迁移继承它）的文档是Rails文档[5]的一部分，它还给出了更多的细节以及其他的迁移选择

(3) 假设迁移成功了，运行`rake db:test:prepare`更新开发数据库的模式并运行测试，如果一切运行良好，把迁移应用到产品数据库，并且给产品配置新的代码。

应用迁移到产品的过程依赖于配置环境；第10章描述了怎样用Heroku来做这件事，这本书中的例子都是使用云计算配置环境的。

我们使用这个三步过程来添加一个新的表来匹配第2章中的描述。这个表格存储了每部电影的名字、评级、描述和发布日期。每个迁移需要一个名称，并且因为这个迁移将会创建电影表格，我们把它命名为CreateMovies。运行`rails generate migration create_movies`命令，如果成功，将会在`db/migrate`下看到一个以创建时间日期开头的并以你提供的名字结尾的新文件，比如，`20111201180638-_create_movies.rb`。（这个命名方案让Rails以它们被创建的顺序来应用迁移，因为文件名是以日期顺序分类的。）编辑这个文件让它看起来像图4.3一样。正如你看到的一样，迁移描述了块的习惯用法：**ActiveRecord::Migration#create_table**方法带了含1个参数的块并且将一个表示被创建的表格的对象让步给那个块管理。**string**、**datetime**等等方法都由这个表格对象提供，并且调用它们将会在新创建的数据库表格中创建列；例如，**t.string 'title'**创建了一个名为`title`的可以保存字符串的列，对大多数的数据库来说，这个字符串最多有255个字符。

保存文件然后输入`rake db:migrate`来实际执行这个迁移并且创建这个表。注意这个内务处理任务也保存了它本身在数据库里的迁移号码，并且默认它只应用还没有没应用过的迁移。（再次输入`rake db:migrate`来验证它第二次什么也没做。）`rake db:rollback`通过运行它的**down**方法来"撤销"最后一次迁移。（你可以试一下。）然而，某些迁移，例如这些删除数据的迁移，不能"撤销"；在这些情况下，**down**方法应该会引发**ActiveRecord::IrreversibleMigration**异常。

■ **详细阐述：环境**

不同的环境也可以重载特定的应用程序行为。例如，产品模式可能会优化以得到更好的性能，但用在开发模式中可能使调试更复杂。测试模式可能会“隔离”外部交互，例如，把要发出的邮件保存到一个文件夹内而不是实际发送出去。`config/environment.rb`文件指定了应用程序的常规启动指令，而`config/environments/production.rb`文件允许设定只使用在产品模式中的特殊的选项，在相同的目录下`development.rb`和`test.rb`也类似。

小结：

- Rails定义了三种环境——开发、产品和测试——每一个都有自己的数据库副本。
- 迁移是描述数据库特定的一组变化的脚本。随着应用程序的发展和新功能的添加，添加迁移来表示这些用来支持新功能的数据库变化。
- 使用迁移来改变数据库需要三个步骤：创建迁移，把迁移应用到开发数据库，以及（如果适用）在测试代码之后把迁移应用到产品数据库。
- `rails generate migration`生成器为新的迁移填写样板，而**ActiveRecord::Migration**类包含了一些定义它的有用的方法。
- `rake db:migrate`仅应用这些还没有应用到开发数据库中的迁移。应用迁移到一个产品数据库的方法取决于配置环境。

自我检查4.2.1. 图4.3中第3行给**create_table**传递了几个参数，参数是什么类型的？
◇ 两个参数：第一个是字符串，第二个是块。使用诗歌模式，允许我们省略括号。

自我检查4.2.2. 在图4.3中，____方法将____交给块控制。
◇ **create_table**；变量**t**

4.3 模型：Active Record基础

在**Movies**表格准备好之后，我们就完成了三个步骤中的第一个——应用程序创建、路由和初始迁移——所以是时候写应用程序代码了。数据库存储了模型对象，但是就像第2章所描述的一样，Rails使用了Active Record设计模式把模型“连接”到数据库，这就是接下来要探索的内容。创建`app/models/movie.rb`文件仅包括两行代码：

代码文件：ch_rails_intro/code/model/movie_1.rb

```
class Movie < ActiveRecord::Base
end
```

由于惯例优先配置原则，`movie.rb`中的这两行启用了大量的行为。为了探索其中的某些行为，要用Control-C停止运行应用程序，同时，要运行`rails console`，它会给你一个交互式的Ruby提示，比如`irb(main):001.0>`，而且这时Rails架构和你应用程序的所有类都已经加载完毕。图4.4描述了一些基本的ActiveRecord功能，通过

代码文件：ch_rails_intro/code/activerecord1.rb

```
1  #### Create
2  starwars = Movie.create!(:title => 'Star Wars',
3    :release_date => '25/4/1977', :rating => 'PG')
4  # note that numerical dates follow European format: dd/mm/yyyy
5  requiem =  Movie.create!(:title => 'Requiem for a Dream',
6    :release_date => 'Oct 27, 2000', :rating => 'R')
7  #  Creation using separate 'save' method, used when updating existing records
8  field = Movie.new(:title => 'Field of Dreams',
9    :release_date => '21-Apr-89', :rating => 'PG')
10 field.save!
11 field.title = 'New Field of Dreams'
12 #### Read
13 pg_movies = Movie.where("rating = 'PG'")
14 ancient_movies = Movie.where('release_date < :cutoff and rating = :rating',
15   :cutoff => 'Jan 1, 2000', :rating => 'PG')
16 pg_movies_2 = Movie.find_by_rating('PG')
17 Movie.find(3)  # exception if key not found; find_by_id returns nil instead
18 #### Update
19 starwars.update_attributes(:description => 'The best space western EVER',
20   :release_date => '25/5/1977')
21 requiem.rating = 'NC-17'
22 requiem.save!
23 #### Delete
24 requiem.destroy
25 Movie.where('title = "Requiem for a Dream"')
26 #### Find returns an enumerable
27 Movie.find_all_by_rating('PG').each do |mov|
28   mov.destroy
29 end
```

图 4.4：尽管MVC中的模型行为通常从控制器调用，但是这些简单的例子还是能够在你写控制器之前帮助你熟悉ActiveRecord的基本特性

在数据库里创建一些电影，搜索它们，改变以及删除它们（CRUD）来实现这些功能。像我们描述各行所起的作用时所讲的，应该复制并粘贴它们到控制台来执行代码。代码示例上包含了该代码文件在本书配套光盘中的路径，可以方便地复制粘贴使用。

第1~6行（Create）在数据库中创建了新的电影。**create!**是**ActiveRecord::Base**的一个方法，**Movie**继承自它，和所有Rails应用程序做法基本相同。ActiveRecord用三种方式使用惯例优先配置原则。第一，它使用类（**Movie**）的名称来决定和类（`movies`）对应的数据库表格名称。第二，它查询数据库找出表中有哪些列（我们在迁移过程中创建的），使得像**create!**这样的方法知道指定哪些属性是合法的以及它们应该属于哪种类型。第三，它给了每一个**Movie**对象属性与**attr_accessor**类似的getter和setter，只不过这些getter和setter不仅仅只会更新一个实例变量。为了证明getter和setter真的是基于数据库模式的，运行一下**Movie.create(:chunky=>'bacon')**。你得到的错误信息看起来惊人的长，但是第一行会告诉你实际错误就是你指定了一个未知属性；剩下的就是Ruby方式提供更多的上下文信息了，我们将会在4.5节中进一步讨论。在继续之前，输入**Movie.all**，将会返回和**Movie**类关联的表格中的所有对象的一个聚合类型。

为作为示范，我们在第6行中指定和第3行中不同格式的发布日期。因为Active Record从数据库模式知道**release_date**是`datetime`列（回顾图4.3里的迁移文件），所以它将会尝试把我们为该属性传递的任何值转换为一个日期。

回顾一下图3.1，名称以**!**结尾的方法是“危险的”。如果在创建对象以及保存到数据库的过程中的任何地方出错，**create!**就是危险的，这时就会引发异常。非危险版本**create**，如果所有的进展良好，则返回新创建的对象；如果有地方出错，则返

回**nil**。对于交互使用，我们更喜欢**create!**，所以就不必每次检查返回值，但是在应用程序中使用**create**并检查返回值更为常见。

第7～11行（Save）表明内存中的Active Record模型对象和数据库中的副本是独立的，它一定会被显式地更新。例如，第8～9行在 **内存** 中创建了一个新的**Movie**对象，而没有把它保存到数据库中。（在执行了第8～9行之后通过执行**Movie.all**可以看出来。在列出的电影里你不会看到 **Field of Dreams** 。）第10行实际上把这个对象保存到了数据库。区别是很关键的：第11行改变了电影的**title**域的值，但是 **仅在内存副本中** ——再次执行**Movie.all**，接着你就会看到数据库副本并没有改变。**save**和**create**都会引发将对象写到数据库的行为，但是仅仅改变了属性而值没有变化。

第12～15行（Read）给出了在数据库中查找对象的一种方式。**where**方法被指定为SQL中的WHERE关键字，其中SQL是一种被大多数关系数据库包括SQLite3使用的结构化查询语言。你可以直接指定像第13行中的字符串作为约束，或者使用第14和15行的关键字替换。关键字替换总是首选，因为，正如我们将在第12章里看到的一样，它允许Rails击败对应用程序的 **SQL injection** 攻击。和**create!**一样，时间会从字符串正确地被转换为**Time**对象，然后从那里被转换成数据库的时间的内部表示。因为这种指定可能会匹配多个对象，**where**总是返回**Enumerable**，在它上面你可以调用任何一个**Enumerable**的方法，例如图3.7所示的这些。

第16行(Read)给出了查找使用**method_missing**的对象的另一种方法。如果你以**find_by_attribute** 形式对一个模型类做一个方法调用，其中 **attribute** 与模型的一个属性匹配，那么ActiveRecord就会重载**method_missing** (就像我们在3.5节中做的一样)来拦截这个调用，把第16行转换成**find.where("rating='PG'")**这种形式。甚至可以说**find_by_release_date_and_rating** 并且传递两个参数。这些方法返回找到的 **一个** 这样的对象（你不应该依赖哪一个）；相反，如果用**find_all_by_attribute** ，你会得到所有匹配元素的一个**Enumerable**。

第17行（Read）给出了最基础的寻找对象的方式，它会返回一个与给定的主键相对应的单一对象。回顾图2.11，每一个存储在关系数据库中的对象都分配了一个主键，这个主键缺乏语义但是保证在表内是唯一的。当在迁移中创建的表之后，Rails默认包括了一个数字主键。因为一个对象的主键是永久以及唯一的，所以它往往能够识别出RESTful URI中的对象，我们在2.7节中已经了解到这一点。

第18～22行（Update）给出了如何**更新**一个对象。和**create**以及**save**之间的关系一样，我们有两种选择：使用**update_attributes**立即更新数据库，或者改变内存对象的属性值，然后用**save!**来保存它（和**create!**差不多，它有一个“安全的”副本**save**，如果出错，则返回**nil**，而不是引发异常）。

第23～25行（Delete）给出了**删除**一个对象的步骤。**destroy**方法（第24行）从数据库中永久地删除了对象。你仍然可以检查这个对象的内存副本，但是尝试修改它或者试图调用它的任何方法将会引起数据库的访问，这些都会引发异常。（在做了**destroy**之后，试试**requiem.update_attributes(...)**或者**requiem.rating='R'**来证明上面所说的事实。）

第26～29行显示出**find_all_by_attribute** 的结果确实表现起来像一个聚合类型——我们可以使用**each**来遍历它，还可以依次删除每部电影。

这种对Active Record旋风般的概述仅仅探索了表面的东西，但是它应该阐明了**ActiveRecord::Base**提供的方法是如何支持基本的CRUD行为的。

作为最后一步，应该使用图4.5中代码，向数据库里 **植入** 一些电影，以使得这章剩下的部分更加有趣。

代码文件：ch_rails_intro/code/seeds.rb

```
# Seed the RottenPotatoes DB with some movies.
more_movies = [
  {:title => 'Aladdin', :rating => 'G',
    :release_date => '25-Nov-1992'},
  {:title => 'When Harry Met Sally', :rating => 'R',
    :release_date => '21-Jul-1989'},
  {:title => 'The Help', :rating => 'PG-13',
    :release_date => '10-Aug-2011'},
  {:title => 'Raiders of the Lost Ark', :rating => 'PG',
    :release_date => '12-Jun-1981'}
]
# NOTE: the following line temporarily allows mass assignment
# (needed if you used attr_accessible/attr_protected in movie.rb)
Movie.send(:attr_accessible, :title, :rating, :release_date)
more_movies.each do |movie|
  Movie.create!(movie)
end
```

图 4.5: 往数据库中添加初始数据称作 播种 ，它和迁移有区别，迁移是用来管理模式的变化。把这个代码复制到`db/seeds.rb`并且用`rake db:seed`来运行它

小结：

- Active Record使用了惯例优先配置原则，从模型类的名称来推断数据库表格名称，并且推断给定的一种模型的列（属性）的名称和类型。
- 基本的Active Record支持集中在CRUD行为上：create、read、update和delete。
- 可以**创建**模型实例，通过调用**new**随后调用**save**，或者调用结合了前两者的**create**。
- 保存在数据库里的每一个模型实例都有一个独特的ID号，在它的表里称作主键，它的属性名称（表中的列名称）是**id**，并且从来不会被“回收”（即使对应的行已经被删除了）。表名和**id**的结合能够唯一地识别出存在数据库里的模型，因此也表明了在RESTful路由的对象通常是如何被引用的。
- 模型实例可**读**（查找），通过使用**where**来表示匹配条件，或者**find**直接查找主键（ID），这种情况可能发生在处理嵌入了对象ID的RESTful URI时。
- 模型实例可以用**update_attributes**来**更新**。
- 模型实例可以用**destroy**来**删除**，在这之后，内存副本仍然可读但是不能修改或者被用来访问数据库。

■ **详细阐述：它叫起来像一个聚合类型，但是它不是**

基于ActiveRecord的**all**、**where**和**find**方法返回的对象当然叫起来像一个集合，但是我们很快就会在第11章里看到，它实际上是一个 **代理对象** ，这个对象甚至不会做查询，除非强制性地要求获得集合中的一个元素，这样才能允许使用多路**where**建立其复杂的查询，而不需要每次查询。

■ **详细阐述：重载惯例优先配置原则**

惯例优先配置原则很不错，但是你总有一些时候需要重载它。例如，如果试图整合Rails应用程序和非Rails遗留程序，数据库表中可能已经有一些名称，这些名称和模型名称并不匹配，或者你可能想要一些比表中的列名称更加友好的属性名称。所有的这些默认项都可以重载，代价就是写更多的代码，就像ActiveRecord文档里说的那样。在这本书里我们坚持原来的配置选择简洁性。

自我检查4.3.1. 为什么**where**和**find**是类方法而不是实例方法?
◇ 实例方法对类的一个实例进行操作，但是直到查找一个或者更多的对象时，才需要对实例操作。

自我检查4.3.2. Rails模型获取**where**和**find**方法，是通过（a）继承还是（b）mix-in?
提示： 查看`movie.rb`文件
◇ （a）它们继承自**ActiveRecord::Base**。

4.4 控制器和视图

下面将通过创建一些视图来支持之前刚学过的CRUD行为来完成这一部分。我们之前定义的RESTful路由（`rake routes` 提醒你它们是什么）希望控制器能够提供一些行为支持**index**、**show**、**new/create** （回顾第2章里，创建一个对象需要与用户有两种交互）、**edit/update**（类似）和**destroy**。从最简单的两个行为开始，**index**应该给出所有的电影的列表，允许我们单击每一个，并且**show**能够给出所点击的电影的详细信息。

对**index**行为，从4.3节中练习的例子知道，**Movie.all**返回了Movies表格中所有电影的聚合类型。因此我们需要一个设置这个聚合类型的控制器方法，以及一个显示它的HTML视图。根据惯例优先配置原则，Rails需要方法做以下的操作来完成Movie资源上的Show RESTful行为（注意单复数以及`CamelCase`和`snake_case`的使用）：

- 模型代码在继承自**ActiveRecord::Base**的**Movie**类中，并且在`app/models/movie.rb`中定义。
- 控制器代码在**MoviesController**类里，定义在`app/controllers/movies_controller.rb`里（注意用模型的类名的复数形式来形成控制器文件名）。应用程序的控制器都是继承自应用程序的根控制器**ApplicationController**（在`app/controllers/application_controller.rb`）里，反过来它又继承自**ActionController::Base**。
- 控制器的每一个实例方法都是根据它操作的行为使用**snake_lower_case**来命名的，所以**show**方法控制了Show行为。
- Show视图模板在`app/views/movies/show.html.haml`中，其中`.haml`扩展表示使用了Haml渲染器。其他的扩展包括使文件中包含XML Builder代码的`.xml`（3.6节中已经讲过），用于Rails内置嵌入式Ruby渲染器的`.erb` （很快就会讲到），以及很多其他的内容。

代码文件：ch_rails_intro/code/controller/index.rb

```
1 # This file is app/controllers/movies_controller.rb
2 class MoviesController < ApplicationController
3   def index
4     @movies = Movie.all
5   end
6 end
```

代码文件：ch_rails_intro/code/view/index.html.haml

```
1  -#  This file is app/views/movies/index.html.haml
2  %h2 All Movies
3
4  %table#movies
5    %thead
6      %tr
7        %th Movie Title
8        %th Rating
9        %th Release Date
10       %th More Info
11   %tbody
12     - @movies.each do |movie|
13       %tr
14         %td= movie.title
15         %td= movie.rating
16         %td= movie.release_date
17         %td= link_to "More about #{movie.title}", movie_path(movie)
```

图 4.6: 支持Index RESTful的控制器代码和模板标记

Rails模块**ActionView::Base**设计出了操作视图的方式。因为我们一直在视图里使用Haml标记（回想一下，我们曾把Haml gem添加到`Gemfile`依赖性里），所以视图文件命名以`.html.haml`结尾。因此，为了完成Index RESTful行为，必须在`app/controllers/movies_controller.rb`中定义**index**行为，以及在`app/views/movies/index.html.haml`中定义一个视图模板。使用图4.6 里的内容来创建这两个文件（需要创建中间目录`app/views/movies/`）。

控制器方法使用之前章节里介绍过的**all**方法来检索Movies中的所有电影，并且把它分配给**@movies**实例变量。回顾第 2章里介绍的Rails应用程序，控制器行为中定义的实例变量对视图是可用的；`index.html.haml`中的第12行使用**each** 来遍历**@movies**聚合类型。这个简单的模板有三个地方需要注意。

第一，表头（`th`）中的列只有关于列的静态文字描述，但是表体（`td`）中的列使用了Haml的=语法表明标签内容应该算作Ruby 代码，其结果写入HTML文档中。在这种情况下，对**ActiveRecord**提供的**Movie**对象使用属性getter。

Sanitization
Haml的=语法审查[6] 在把Ruby代码插入到HTML输出之前得到它的评估结果，以帮助阻止第12章中描述的跨站点脚本和类似攻击。

第二，我们已经给出了电影表的HTML ID 电影，随后将会把它用于使用CSS视觉化页面上，和第2章中讲过的一样。

第三，就是第17行中的调用**ActionView**提供的创建视图的众多辅助方法之一的**link_to**。根据文档[7] 中描述的，第一个参数是一个将会以链接（可点击的文本）的形式出现在页面上的字符串，第二个参数是用来创建URI的，是实际的链接目标。这个参数有几种形式；我们使用过的形式利用了URI助手**movie_path()**（如**show**行为的`rake routes`），这个助手把参数作为RESTful资源的一个实例（在这里是**Movie**的实例），并且为对象的Show RESTful路由生成了RESTful URI。和`rake routes`提醒一样，电影的Show行为是通过URI `/movies/:id`表示出来的，其中`:id`是Movies表中电影的主键，所以这就是**link_to**创建的链接目标的样子。为了验证这一点，重启应用（应用程序的根目录里的`rails server`），然后访问网址`http://localhost:3000/movies/`，这是一个与`index`行为对应的URI。如果一切

都进行顺利，你应该会在数据库里看到所有电影的列表。如果使用浏览器的查看源文件选项来查看生成的源，就可以看到由**link_to**生成的链接都有URI，这些URI和每一部电影的**show**行为对应（可以试着点开一个，但是会出现错误，因为还没有创建控制器的**show**方法。）

我们在4.1节里添加的`resources :movies`行实际上为RESTful URI创建了各种各样的辅助方法，图4.7里总结了一下。你可能已经猜到了，惯例优先配置使用原则决定了辅助方法的名称，而作为调用`resources :movies`的结果元编程是用来定义它们的。这样的辅助方法的创建和使用在你意识到除了我们一直使用的标准的RESTful路由外，可能会去定义更多复杂和不规则的路由后，可能就不会那么无厘头了，或者是意识到在应用程序开发期间可能去决策一个不同的路由方案会更有意义。辅助方法让视图不受这样的变化的影响，并且让它们着重去展示 **什么** （what），而不是去包含这些 **如何** （how）显示它的代码。

助手方法	返回的URI	RESTful路由和行为	
movies_path	/movies	GET /movies	index
movies_path	/movies	POST /movies	create
new_movie_path	/movies/new	GET /movies/new	new
edit_movie_path(m)	/movies/1/edit	GET /movies/:id/edit	edit
movie_path(m)	/movies/1	GET /movies/:id	show
movie_path(m)	/movies/1	PUT /movies/:id	update
movie_path(m)	/movies/1	DELETE /movies/:id	destroy

图 4.7: 正如ActionView::Helpers类的文档中描述的那样，Rails使用元编程来创建基于ActiveRecord类名的路由助手。假设m是一个ActiveRecord Movie对象。RESTful路由和`rake routes`的输出显示的一样；调用不同的路由可能会有相同的URI，但是使用的是不同的HTTP方法，例如`create`和`index`

■ **详细阐述：为了简洁性的反射和元编程**

更确切地说，如果**link_to**的第二个参数是类**Movie**的ActiveRecord模型的话，**link_to**将会隐式地对它调用**movie_path**，所以你不需要再做。所以**link_to**调用可以被写做**link_to "More about #{movie.title}", movie**。

关于这些视图有最后一点需要注意的。如果在浏览器中查看源代码，将会看到它包括了在Haml模板中没有出现过的HTML标记，例如头元素包含了到`assets/application.css`样式表的链接以及一个`<title>`标签。这个标记来自于`application`模板，它默认“封装”了所有的视图。使用Rails的`erb`模板系统中的`rails new`命令创建默认文件`app/views/layouts/application.html.erb`，但是因为我们喜欢Haml 的简洁性，所以我们推荐删掉这个文件，并且用图4.8中的替换它，然后观看视频4.4.1，了解“封装”过程是如何工作的。

视频4.4.1: 应用程序布局.
光盘文件：ch_rails_intro/mov/application_layout.mp4
视频显示了`app/views/layouts/application.html.haml`模板在默认情况下被用来“封装”行为视图，并且使用了**yield**，这和3.8节里的例子很像。

你可以自己用相似的过程尝试创建控制器的行为和用于**show**的视图：

代码文件：ch_rails_intro/code/view/application.html.haml

```
!!! 5
%html
  %head
    %title Rotten Potatoes!
    = stylesheet_link_tag 'application'
    = javascript_include_tag 'application'
    = csrf_meta_tags

  %body
    = yield
```

图 4.8: 把这个文件保存为`app/views/layouts/application.html.haml`，然后删除掉这个目录中的`application.html.erb`；这个文件是它的**Haml**等价文件。第**6**行载入了一些基本的**JavaScript**支持；虽然我们直到第**5**章中才会介绍到**JavaScript**编程，但是**Rails**的一些内置助手透明地使用了它们。第**7**行介绍了防止 **cross-site request forgery** 攻击的方法，这个攻击在第**12**章会讲到。同时我们也会让`title`元素更人性化一点

代码文件：ch_rails_intro/code/controller/show_method.rb

```
# in app/controllers/movies_controller.rb

def show
  id = params[:id] # retrieve movie ID from URI route
  @movie = Movie.find(id) # look up movie by unique ID
  # will render app/views/movies/show.html.haml by default
end
```

图 4.9: 用于**Show**行为的控制器方法的一个示例实现。一个更健壮的实现方法使用了**find_by_id**，并且检测结果，正如在**4.3**节中给出的警告一样。我们将会在第**5**章里看到如何处理这样的情况

代码文件：ch_rails_intro/code/view/show_view.html.haml

```
-# in app/views/movies/show.html.haml

%h2 Details about #{@movie.title}

%ul#details
  %li
    Rating:
    = @movie.rating
  %li
    Released on:
    = @movie.release_date.strftime("%B %d, %Y")

%h3 Description:

%p#description= @movie.description

= link_to 'Back to movie list', movies_path
```

图 4.10: 与图**4.9**相配的一个示例视图。对未来的**CSS**样式，我们给形式细节（`ul`）以及段文字描述（`p`）分配了唯一的**ID**。我们用**strftime**库函数[9]来优化日期的格式化，然后用带有**RESTful**辅助方法**movies_path**（见图**4.7**）的**link_to**方法，给目录页面提供一个更方便的链接。一般说来，你可以在`rake routes`输出的最左侧一列把**_path**添加到任意一个**RESTful**资源助手里，以调用一个方法，产生相应的**RESTful URI**

(1) 用`rake routes`来提醒你应该给控制器方法分配什么样的名字，以及在URI里传递哪些参数。

(2) 在控制器方法中，使用4.3节中介绍过的合适的ActiveRecord方法，从数据库里检索适当的**Movie** 对象，然后把它赋值给一个实例变量。

(3) 在`app/views`层次结构的右侧位置创建一个视图模板，使用Haml标记显示出你在控制器方法中设置的**Movie**对象的各种属性。

(4) 通过点击`index`视图中的一个电影链接，测试一下你的方法和视图。

一旦这些都完成了，你可以检测一下图4.9中的样本控制器方法以及图4.10中的样本视图。实验一下当**link_to**和**strftime**给出其他的参数值时的情况，来理解它们是怎么工作的。

因为当前只有“干骨架”的视图很不美观，但是只要继续坚持在这个应用程序上工作，我们可能也会看到一些更有吸引力的内容。把下面简单的已经包含在第5行的`application.html.haml`模板里CSS样式复制到`app/assets/stylesheets/application.css`中。

小结：

- Haml模板语言允许在视图中用Ruby代码添加HTML标记。评估Ruby代码的结果就是或者丢弃或者插入到HTML页面中。
- 为了简洁，Haml依靠缩进来显示HTML元素嵌套。
- 惯例优先配置使用原则是用来决定在给定模型下控制器和视图的文件名称的。如果像在**resources :movies**里一样使用了RESTful路由助手，那么惯例优先配置使用原则也会把RESTful的行为名称映射到控制器行为（方法）的名称上。
- Rails提供了各种各样的辅助方法，这些方法利用了RESTful路由URI，包括**link_to**，用于生成URI指向RESTful行为的HTML链接。

代码文件：ch_rails_intro/code/simple.css

```
/* Simple CSS styling for RottenPotatoes app */
/* Add these lines to app/assets/stylesheets/application.css */

html, body {
  margin: 0;
  padding: 0;
  background: White;
  color: DarkSlateGrey;
  font-family: Tahoma, Verdana, sans-serif;
  font-size: 10pt;
}
div#main {
  margin: 0;
  padding: 0 20px 20px;
}
a {
  background: transparent;
  color: maroon;
  text-decoration: underline;
  font-weight: bold;
}
h1 {
  color: maroon;
  font-size: 150%;
  font-style: italic;
  display: block;
  width: 100%;
  border-bottom: 1px solid DarkSlateGrey;
}
h1.title {
  margin: 0 0 1em;
  padding: 10px;
  background-color: orange;
  color: white;
  border-bottom: 4px solid gold;
  font-size: 2em;
  font-style: normal;
}
table#movies {
  margin: 10px;
  border-collapse: collapse;
  width: 100%;
  border-bottom: 2px solid black;
}
table#movies th {
  border: 2px solid white;
  font-weight: bold;
  background-color: wheat;
}
table#movies th, table#movies td {
  padding: 4px;
  text-align: left;
}
#notice, #warning {
  background: rosybrown;
  margin: 1em 0;
  padding: 4px;
}
form label {
  display: block;
  line-height: 25px;
  font-weight: bold;
  color: maroon;
}
```

自我检查4.4.1. 在图4.7中，为什么New行为（**new_movie_path**）和Create行为（**movies_path**）的辅助方法没有像Show或者Update助手一样带一个参数?

◇ Show和Update对已经存在的电影进行操作，所以它们通过一个参数来识别应该对哪部电影操作。定义里的New和Create 是对不存在的电影进行操作。

自我检查4.4.2. 在图4.7中，为什么Index行为的辅助方法没有带一个参数？
提示：原因和自测题4.4.1中的不一样。

◇ Index行为仅仅显示所有电影的列表，所以不需要参数来区别是对哪一部电影操作。

自我检查4.4.3. 在图4.6中，第12行中为什么没有**end**和**do**相对应？
◇ 和Ruby本身不同，Haml依赖缩进来表示嵌套，所以当在**do**中执行Ruby代码时Haml提供了**end**。

4.5 调试：当出现错误时

今天的软件的惊人的复杂性让高效成为可能，但是有这么多的“变化的因素”，这也意味着有些地方不可避免会出现错误，尤其是在学习新的语言和工具时。你输错了某些东西、你环境或者配置发生变化或者是任何其他原因，都可能导致错误出现。尽管在这本书里我们采取措施来尽量减少错误的发生，例如使用测试驱动开发（第8章）来避免很多问题以及提供虚拟机镜像，但是错误 **依旧** 会发生。通过记住首字母缩写**RASP**：读、询问、搜索、发布、你可以高效地应对错误。

一个有趣的视角：关于盲目的“解决大范围内”的问题的就是Jargon File的黑客公案“Tom骑士和Lisp机。”[10]

读错误信息。Ruby的错误可能看起来非常长，但是一个长错误信息往往是你的朋友，因为它给出了 **回溯** ，不仅给出了显示错误发生的地方的方法，还给出了方法的调用者、调用者的调用者，等等。当你看到一个长错误信息时不要灰心；要用这个信息来了解出现错误（“stopped the show”的问题）的大致原因，并尽可能地找到错误根源的方法。这需要你对错误代码的语法有一定的了解，如果盲目地剪切和粘贴别人的代码而不去理解它怎么工作的，以及做了哪些假设，你可能会缺乏这个能力。当然了，由于剪切粘贴而导致的语法错误就像你把别人的代码作为自己的重用时出现错误一样，但是至少重用你还能理解你自己的代码（你觉得呢？）。

Ruby错误特别常见的一个原因是“Undefined method **foobar** for **nil:NilClass**”，它表示“你在值为**nil**以及类是**NilClass**的对象上调用**foobar**，但是这个对象没有定义**foobar**。”（**NilClass**是一个特殊的类，它唯一的实例就是常量**nil**。）这种情况通常发生在某些计算失败并且返回了**nil**而不是你想要的对象时，但是你忘了检查这个错误并且随后就尝试对一个你认为是有效的对象调用一个方法。如果计算发生在“上游的”一个方法里，那么回溯就可以帮你找出错误在哪里。

在使用Rails的SaaS应用中，如果控制器行为中发生了失败操作，但是无效的对象作为实例变量被传递，并且在视图中引用了，那么这种混乱就会加剧，就像下面的这段摘自控制器和视图的内容一样：

代码文件：ch_rails_intro/code/nil_object.rb

```
# in controller action:
def show
  @movie = Movie.find_by_id(params[:id]) # what if this movie not in DB?
  # BUG: we should check @movie for validity here!
end

-# ...later, in the Haml view:

%h1= @movie.title
-# will give "undefined method 'title' for nil:NilClass" if @movie is nil
```

询问同事。如果你是和别人一起编程的，两个人的大脑总比一个人好。如果你是在一个"开放式座位"的配置，或者启动了即时消息，那么请把消息共享出去。

搜索错误信息。你可能会感到很吃惊，有经验的开发者经常使用像Google这样的搜索引擎来寻找错误信息中的关键字或者关键短语来处理一个错误。你也可以搜索网站比如StackOverflow[11] 和ServerFault[12]，它们专门帮助开发者并且允许你给特定的问题投票选出最有用的答案，然后他们就把这个答案推到问题列表顶端。

发布你的问题到这其中某一个网站上，如果所有其他尝试都失败的话。尽可能具体地指出哪里出错了、你在什么环境下以及如何重现问题。

- **含糊的**："The `sinatra` gem doesn't work on my system." 这句话没有足够的信息能让别人帮助你。
- **好一点，但是有点让人烦**："The `sinatra` gem doesn't work on my system. Attached is the 85-line error message." 其他的开发者可能和你一样忙，他们可能不会去花长时间提取相关的有用信息。
- **最佳**：在网址上查看StackOverflow上这个问题的actual transcript[13]。在下午6:02，开发人员提供了详细的信息，例如操作系统的名称和版本号、他成功运行的详细命令以及导致的意想不到的结果。其他能帮助的人要求给出额外的具体信息，截止下午7:10，有两个答案已经确定了问题所在。

尽管很让人印象深刻的是，仅在一个小时之后他就获得了答案，但是这也意味着他浪费了一个小时的编程时间，这就是为什么你应该在已经没有其他方法可行之后再发布问题。怎么样才可以提高你自己的调试能力呢？问题分为两种。第一种，错误或者某种异常导致程序无法继续运行。因为Ruby是一门解释语言，语法错误会导致这个问题（和Java不一样，一旦有语法错误就不会编译）。如果应用程序突然终止你可以做如下尝试。

- 利用自动缩进和语法高亮显示。如果文本编辑器坚持和你打算的相比多缩进一行，那么你可能忘记了再写一个圆括号、大括号或者**do...end**块与之前的对应，或者你可能忘记了"避免"一个特殊字符（例如，单引号字符串里的单引号）。如果编辑器没有这些功能，你要么就只能把代码写在石碑上，要么换一个更高效的现代化编辑器，附录A中有一些推荐。
- 查看日志文件，通常是`log/development.log`，查找完整的错误信息，包括回溯。在产品应用中，这通常是你唯一的选择，因为Rails应用在产品模式下通常会显示一个更加友好的用户错误页面，而不是在开发模式中可以看到的错误回溯。

第二种，应用程序运行但是产生了一个不正确的结果或者行为。大多数的开发者使用两种方式结合来调试错误。第一种就是插入 **仪器** ——能够给出额外的报告，记录了程序运行期间的不同点处重要的变量值。在Rails SaaS应用里可以在很多地方插入调试仪器——尝试一下下面的每一个，了解一下它们是如何工作的：

printf调试 对这门技术来说是一个古老的名字，来自C库函数，能够在终端上打印出字符串。

- 在视图上显示对象的详细描述。例如，在任何一个视图中插入**=debug(@movie)**或者**=@movie.inspect**（开头的=告诉Haml执行代码并且把结果插入到视图中）。
- 在控制器方法中通过引发异常来"Stop the show"，异常消息表示的是你要检查的值，例如，**raise params.inspect** 就是查看控制器方法内的**params**哈希的详细值。Rails将会给出由请求引发的异常消息页面。

- 用**logger.debug(message)**把 **消息** 打印到日志。模型和控制器中的**logger**都是可用的，并且可以记录各种紧急的消息；比较`config/environments/production.rb`和`development.rb`，看看在产品和开发模式中默认日志级别有什么不同。

调试的第二种方式就是使用交互式调试。我们已经通过Gemfile安装了`ruby-debug19` gem；在Rails 应用程序里使用调试器，先用`rails server --debugger`命令启动应用程序服务器，然后在你想停下来调试的地方插入语句**debugger**。当碰到这个语句时，启动服务器的终端窗口将会给你一个调试提示。4.7节将会展示如何使用调试其来获取Rails的一些内部信息。

为了调试非Rails Ruby应用程序， 在程序开始的地方插入**require 'ruby-debug'**。

小结：

- 使用一个带有语法高亮和自动缩进来帮助寻找语法错误的语言识别编辑器。
- 通过在视图中插入**debug**或者**inspect**的输出调试应用程序，或者让它们变成**raise**的参数，这将会引起一个运行时异常，将会把 **消息** 显示成Web页面。
- 使用交互式调试器调试，确保应用程序的Gemfile包括`ruby-debug19`，以`rails server --debugger` 启动应用程序服务器，并且把语句**debugger**放在代码中你想中断的地方。

自我检查4.5.1. 为什么不能只使用**print**或**puts**来显示消息帮助你调试SaaS应用？
◇ 和命令行应用程序不一样，SaaS应用没有连接到终端窗口，所以没有一个明确的地方让print语句输出结果。

自我检查4.5.2. 本节中描述了三种调试方法，一旦应用程序部署好并且已经在产品阶段了，哪一种适合收集仪器或者诊断信息？
◇ 只有**logger**方法是合适的，因为如果是用在一个产品应用程序上的话，其他两种方法（在控制器里“stopping the show”或者在视图中插入诊断信息）会干扰真实的客户的使用。

4.6 表单提交：New和Create

我们在视图的最后一部分涉及一个稍微复杂一些的情况：提交表单，例如创建一个新的电影或者更新已经存在的。这需要处理三个问题：

(1) 怎样给用户显示一个需要填写的表单？

(2) 用户填写的信息怎样变成控制器行为可用的内容，使得**create**或者**update** ActiveRecord调用可以使用它？

(3) 作为一个创建或者更新项目的RESTful请求的结果，应该返回和显示出哪些资源？和我们要求电影的列表或者电影的详细信息不同，我们不能明显地给出创建或者更新之后的结果的显示方式。

当然，在深入之前，需要提供给用户一种获得我们将要创建的填写表单的方式。因为这个表单是用来创建一部新电影的，它和RESTful行为**new**对应，并且

代码文件：ch_rails_intro/code/view/form_helper_example.html.haml

```
%h2 Create New Movie

= form_tag movies_path, :method => :post do

  = label :movie, :title, 'Title'
  = text_field :movie, :title

  = label :movie, :rating, 'Rating'
  = select :movie, :rating, Movie.all_ratings

  = label :movie, :release_date, 'Released On'
  = date_select :movie, :release_date

  = submit_tag 'Save Changes'
```

图 4.11: 用户寻找的为Rotten Potatoes创建和增加一部新电影的表单

会遵循把表单放在`app/views/movies/new.html.haml` 中的惯例。因此我们可以利用自动提供的RESTful URI助手**new_movie_path**来创建一个到表单的链接。在`index.html.haml`结尾添加单行代码就可以实现：

代码文件：ch_rails_intro/code/view/add_link_to_new.html.haml

```
-# add to end of index.html.haml

= link_to 'Add new movie', new_movie_path
```

如果用户单击这个链接，将会触发什么样的控制器行为？因为使用了URI助手**new_movie_path**，所以它将会变成**new** 控制器行为。虽然我们还没有定义这个行为，但是这个时候，因为用户创建了一个全新的movie条目，行为需要做的唯一一件事情是为`new`行为呈现对应的视图。回顾一下，在默认情况下，每一个控制器都会自动加载对应名称的模板（这种情况下为`new.html.haml`），所以只可以添加下面的**new**方法到`movies_controller.rb`中：

代码文件：ch_rails_intro/code/controller/new_1.rb

```
def new
  # default: render 'new' template
end
```

Rails通过使用所有视图都可用的表单标记助手[14]，使得描述填写式表单变得简单。把图4.11中的代码放到`app/views/movies/new.html.haml` 中，并且观看描述了表单里正在发生的事情的视频4.6.1。

视频4.6.1: 带填写表单的视图.
光盘文件：ch_rails_intro/mov/form_helpers.mp4
生成表格的**form_tag**方法需要一个到表单提交的地方的路由——也就是说，一个URI和HTTP。就像`rake routes`提醒的那样，我们用RESTful URI助手HTTP `POST`方法生成一个到**create**行为的路由。

正如视频里提到的那样，表单标记助手并不支持所有字段的输入（在这里是不支持日期字段），在某些情况下你需要生成表格，这些表格的字段不一定要和某个ActiveRecord对象的属性相对应。

回顾一下进度，我们创建了**new**控制器方法，它会呈现给用户一个要填写的表单的视图，把这个视图放到`new.html.haml`里，并且安排表单提交给**create**控制器方法。剩下要做的就是使用**params**（表单字段值）中的信息在数据库里创建一条新的电影记录。

小结：

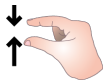
- Rails提供了表单助手来生成一个填写式表单，它的字段和一个特殊类型的ActiveRecord对象的属性相关。
- 当创建一个表单时，通过传递合适的RESTful URI和HTTP方法（`rake routes`里显示的一样）给**form_tag**，指定将会收到表单提交的控制器操作。
- 当表单提交之后，控制器行为可以检测到**params[]**，它包括了每个表单字段的关键字，它的值是用户提供的字段的内容。

自我检查4.6.1. 图4.11中第3行，把**:method=>:post**变成**:method=>:get**之后的影响是什么？为什么？
◇ 由于路由需要URI和方法，所以表单提交会导致列出所有的电影而不是创建一部新的电影。如图4.7 所示，使用`GET`方法的**movies_path**助手将会路由到**index**行为，而使用`POST`方法的**movies_path**助手将会路由到**create**行为。

自我检查4.6.2. 假设图4.11中的表单提交会创建一个新的电影，那么为什么视图被称作`new.html.haml` 而不是`create.html.haml`呢？
◇ RESTful和它的视图应该命名被请求的资源。在这里，用户 **加载** 这个表格时请求的资源是表格本身，也就是，创建一个新的电影的能力；因此`new`是这个资源的合适的名称。用户 **提交** 表格时请求的资源，以图中第3行指定给表单提交的路由来命名，它是实际创建的新电影。

4.7 重定向和Flash

回顾一下4.3节中的例子，**Movie.create!**调用需要一个有属性名称和值的哈希来创建一个新的对象。正如视频4.7.1里显示的一样，由表单标记助手创建的表单字段名称，名称都是**params['movie']['title']**，**params['movie']['rating']**等等形式的。因此，**params[:movie]**就是电影属性名称和值的哈希值，我们可以用**Movie.create!(params[:movie])**直接传递。视频也给出了有用的技术，即在控制器行为执行期间使用调试断点来详细查看“后台”内容。



视频4.7.1: Create行为.
光盘文件：ch_rails_intro/mov/create.mp4
在**create**控制器行为里，我们放置一个调试断点来检查发生的事情，并且使用图4.12中的一部分调试命令来检查**params**的哈希值。特别的是，因为我们的表单的字段名称形式都是**movie[...]**，**params['movie']**本身就是一个有各种各样的电影字段的哈希，可以赋给一个新的**Movie**对象。像很多Rails方法一样，**params[]**可以带一个符号或者一个字符串——实际上**params**并不是一个正规的散列，而是**HashWithIndifferentAccess**，"叫起来像"一个哈希的Rails类允许它的关键字作为符号或者字符串被访问。

n	执行下一行
s	执行下一个语句
f	完成当前的方法调用并返回
p **expr**	打印 **expr**，可以是当前堆栈框架作用域内的任何内容
eval **expr**	计算 **expr**；可以用于设置作用域范围内的参数，就像**eval x=5**里一样
up	把堆栈调用提高到调用者的堆栈框架内
down	把堆栈调用返回到被调用者的堆栈框架里
where	显示你在调用堆栈里的位置
b **file:num**	在 **file**（如果 **file:** 省略了则为当前文件）里的行 **num** 设置断点
b **method**	当 **method** 调用时设置断点
c	继续执行直到下一个断点
q	退出程序

图 4.12: 交互式Ruby调试器命令汇总

这带来了4.6节开始提出的第三个问题：当**create**行为完成时，应该呈现一个什么样的视图呢？为了和其他的行为（比如**show**）保持一致，可以创建一个包含了通知用户成功消息的友好视图`app/views/movies/create.html.haml`，但是去拿一个单独的视图来完成这件事情似乎有点不合理。相反大多数Web应用程序做的就是给用户返回一个更加有用的页面——主页，或者是所有电影的目录——但是它们也在页面上额外显示了成功的消息，让用户知道他们的修改保存成功。

Rails让实现这种行为变得简单起来。为了将用户送到一个不同的页面，**redirect_to**不是通过呈现一个视图来结束控制器行为，而是对不同的行为重启一个全新的请求。因此，**redirect_to movies_path**就像用户突然请求RESTful Index行为`GET movies`一样（也就是说，和助手**movies_path**对应的行为）：**index** 行为将会运行完成并且像平常一样呈现它的视图。换句话说，控制器行为必须呈现一个视图或者重定向到另外一个行为上。把调试断点从控制器行为移开（是根据视频4.7.1修改了代码后插入的断点），并且修改它使它看起来像下面的列表；然后重载电影列表页面来测试这个行为，点击Add New Movie，然后提交表单。

代码文件：ch_rails_intro/code/controller/create_method.rb

```
# in movies_controller.rb
def create
  @movie = Movie.create!(params[:movie])
  redirect_to movies_path
end
```

为了简便我们在5.2节里再讨论**attr_protected**，没有它攻击者可以操纵传给**Movie.create!**的表单去设置任意的属性[15]，而这些属性原本对于普通用户是不可修改的。

当然了，为了得到友好的用户界面，我们可能会给出一个消息，通知用户创建一个电影成功。（我们随后就会处理失败时的情况）问题是，当我们调

代码文件：ch_rails_intro/code/view/add_flash_to_template.html.haml

```
-# this goes just inside %body:
- if flash[:notice]
  #notice.message= flash[:notice]
- elsif flash[:warning]
  #warning.message= flash[:warning]
```

图 4.13: 注意使用CSS来给flash消息设计式样：每种类型的消息都显示在一个div里，它独特的ID为notice或者warning，这依赖于消息的类型，但是它们共享了相同的类message。这给了我们在CSS文件内的自由性，我们可以参考它们的类给两种类型的消息设计相同的样式，或者参考它们的ID给它们设计不同的样式。值得注意的是，Haml的简洁性允许在单行里显示出每个div的类、ID属性、 以及 消息文本

用**redirect_to**时，它启动了一个全新的HTTP请求；并且因为HTTP是无状态的，与**create**关联的所有的变量都消失了。

为了解决这个常见的情况，**flash[]**是一个和哈希函数类似的特殊方法，但是它会从当前的请求持续到下一个请求。（我们以后会探索Rails是如何实现它的）换句话说，如果在当前的控制器行为期间把某些内容放到**flash[]**里，就可以在 **后续的** 操作中访问它。整个哈希值都被保存下来，但是按照惯例，**flash[:notice]**是用于信息消息的，而**flash[:warning]**是用于出错的地方的消息的。你可以试一下在**flash**里修改控制器的行为存储一个有用的消息：

代码文件：ch_rails_intro/code/controller/create_method_with_flash.rb

```
# in movies_controller.rb
def create
  @movie = Movie.create!(params[:movie])
  flash[:notice] = "#{@movie.title} was successfully created."
  redirect_to movies_path
end
```

发生了什么？即使成功创建一个新的电影（新的电影出现在了所有电影的列表中），但是并没有迹象表明我们刚刚创建了有用的消息。你可能已经猜到了，那是因为我们实际上并没有修改任何视图来显示这个消息。

但是我们应该修改哪一个视图呢？在这个例子里，我们选择把用户重定向到电影列表里，所以我们可能应该在Index视图里增添代码来显示这个消息。但是在未来我们可能会将用户重定向到其他地方，并且在任何情况下，显示一个确认消息或者警告消息的想法非常常见，有必要把它单独分出来，而不是把它放到某个特定的视图中。

如果用CSS做任何重要的工作，那么你会想要使用一个专用的CSS编辑器，例如开源、跨平台的Amaya[16]或者很多商业产品之一。

回顾一下，`app/views/layouts/application.html.haml`默认情况下是用来“封装”所有视图模板的。这是一个显示flash消息的好方法，因为不论呈现的是什么视图，任何待定的消息都会被显示出来。让`application.html.haml`看起来和图4.13中类似——这需要在**%body**和**=yield**之间增添四行代码来在页面开头显示任何未定的flash消息。

尝试让所有的**flash**消息都以红色文本居中打印出来。你需要用`app/assets/stylesheets/application.css`添加合适的CSS选择器，以匹配在应用页面模板里显示出**flash**的HTML元素。CSS属性`color: red`和`text-align: center`能获得这些效果，但是也可以随意去实验其他的视觉风格、颜色、边界等等。

小结：

- 尽管完成一个控制器行为的最常见的方式就是把对应行为的视图呈现出来，但是对某些行为，例如**create**，更合理的方法是给用户一个不同的视图。用**redirect_to**重定向到一个不同的行为上来替换默认的视图呈现。
- 尽管重定向触发浏览器重启一个全新的HTTP请求，但是**flash**可以保存一小部分消息，这部分消息对新的请求是可用的，例如，要给用户显示有用的消息可以考虑重定向。
- 可以在应用程序的模板中放置标记来显示**flash**消息，而不是在每一个可能需要显示这样的消息的视图中复制它，这可以简化视图。

■ **详细阐述：会话**

实际上，flash仅仅是更普遍的**session[]**（会话）一个特例。像flash一样，会话表现得像一个哈希函数，其中的哈希内容贯穿来自同一浏览器的不同请求，但是和flash不同，flash在下一个请求后自动被擦除，但放在会话的任何信息都会保持不变，直到被删除。可以使用**session.delete(:key)**来删除个别项目，就像处理常规哈希一样，或者使用**reset_session**理整个会话。请记住会话是基于cookies的，所以来自不同用户的会话是独立的。此外，与谬论和陷阱中讲的一样，应注意你在会话中存了多少内容。

自我检查4.7.1. 为什么每一个控制器行为必须呈现一个视图，或者执行一个重定向？
◇ HTTP是一个请求-应答协议，所以每一个行为必须产生一个应答。一种应答是视图（Web页面），而另一种就是重定向，指示浏览器向一个不同的URI发送一个新的请求。

自我检查4.7.2. 在图4.13中，假设要输出一个HTML标记，那么为什么第2行以“-”开头而不是“=”呢？
◇ “=”表示Haml评估Ruby表达式，并且替代视图中的表达式。但是不需要替代视图中的**if**-表达式的值——我们需要实际的Haml从**#notice.message**中生成的HTML标记，以及计算**flash[:notice]**的结果，这些以“=”开头可以正确地实现。

4.8 完成CRUD：编辑/更新和销毁

我们现在可以根据一个类似的过程来为**update**功能增添代码。像**create**一样，这需要两个操作——第一个是显示带有可编辑消息的表单（**edit**），另一个是接受表单提交，并且应用更新的消息（**update**）。当然，首先需要给用户一种指定Edit行为的途径，所以在进一步深入之前，先修改`show.html.haml`视图，使得它的最后两行和下面的代码匹配，其中第2行使用助手**edit_movie_path**来生成一个RESTful URI，它会触发**@movie**的**edit**行为。

代码文件：ch_rails_intro/code/view/show_view_with_edit_link.html.haml

```
1 -# modify last 2 lines of app/views/movies/show.html.haml to:
2 = link_to 'Edit info', edit_movie_path(@movie)
3 = link_to 'Back to movie list', movies_path
```

实际上，如图4.14所示，new/create和edit/update行为对在很多方面相似。

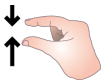
	创建	更新
传递给视图的参数	无	**Movie**的现有实例
默认表单字段值	空	现有的电影属性
提交按钮标签	“创建电影”（或者“保存变化”）	“更新电影”（或者“保存变化”）
控制器行为	**new**服务表单，**create**接收表单并修改数据库	**edit**服务表单，**update**接收表单并修改数据库
params[]	新电影的属性值	已有的电影更新后的属性值

代码文件：ch_rails_intro/code/view/edit.html.haml

```
%h2 Edit Movie

= form_tag movie_path(@movie), :method => :put do

  = label :movie, :title, 'Title'
  = text_field :movie, 'title'

  = label :movie, :rating, 'Rating'
  = select :movie, :rating, Movie.all_ratings

  = label :movie, :release_date, 'Released On'
  = date_select :movie, :release_date

  = submit_tag 'Save Changes'
```

图 4.14: edit/update行为对和我们之前已经实现的new/create（上图）非常相似；实际上，针对edit视图的Haml标签和new视图仅仅在第3行不同（下图）。在第5章我们将会发现新的方法来简化二者相同的部分

使用图4.14来创建`edit.html.haml`视图，它和`new`视图（见图4.11）几乎是相同的——唯一的不同在第3行，它指定了表单提交时的RESTful路由。正如`rake routes`告诉我们的一样，**create** 行为要求URI `/movies`上的HTTP `POST`，所以图4.11在表单行为中使用了**:method=>:post** 和URI助手**movies_path**。相反，**update**行为要求`/movies/:id`上的HTTP `PUT`，其中`:id`是即将更新的资源的主键，所以图4.14的第3行指定了**:method=>:put**，并且为了编辑这部特定的电影，使用URI助手**movie_path(@movie)**来构建URI。我们本可以使用`new.html.haml`里的**form_tag "/movies"**以及`edit.html.haml`里的**form_tag "/movies/#{@movie.id}"**，手动地构建URI，但是URI助手能够更加简洁清楚地表达意图，并且独立于实际的URI字符串，这些现象应该改变。我们很快就会看到，当应用程序引入不同种类资源间的关系时，例如观众有最喜欢的电影，RESTful URI会变得更加复杂，而助手则相应地变得更加简洁易读。



下面是一些需要添加到`movies_controller.rb`里实验这个功能的实际控制器方法，所以去试一下吧。

代码文件：ch_rails_intro/code/controller/edit_update.rb

```
# in movies_controller.rb

def edit
  @movie = Movie.find params[:id]
end

def update
  @movie = Movie.find params[:id]
  @movie.update_attributes!(params[:movie])
  flash[:notice] = "#{@movie.title} was successfully updated."
  redirect_to movie_path(@movie)
end
```

试着点击你在上面插入的编辑已存在的电影的链接`Edit`。当更新一个已存在的电影时，观察一下对应于电影的当前属性的表单字段的默认填入值。这是因为助手，例如`new`或者`edit`模板中第6行的**text_field**，默认情况下会寻找一个名称和它们的第一个参数匹配的实例变量，——在这种情况下，第一个参数是**:movie**，所以**text_field**助手将会寻找变量**@movie**。如果它存在并且和一个ActiveRecord 模型对应，那么助手会假定这个表单是用于编辑一个已存在的对象，并且**@movie**的当前属性值将用于填充表单字段。如果它不存在或者不能响应第二个参数（**'title'**）的属性方法，那么表单字段将会为空。这个行为可以成为把实例变量命名为**@movie**而不是（比如）**@my_movie**的有力支持：你依旧可以从助手里获得额外功能，但是你必须给它们传递额外的参数。

最后一个CRUD操作是Delete，图4.4显示出它可以通过对ActiveRecord模型调用**destroy**来完成。和Update操作类似，常见的应答Delete的方法，通常是销毁一个对象，然后返回给用户一些其他的有用的页面（例如Index视图），以及给出项目被删除的确认消息，所以我们已经知道如何写控制器方法了——把下面的这些行添加到`movies_controller.rb`里：

代码文件：ch_rails_intro/code/controller/destroy.rb

```
def destroy
  @movie = Movie.find(params[:id])
  @movie.destroy
  flash[:notice] = "Movie '#{@movie.title}' deleted."
  redirect_to movies_path
end
```

（回顾图4.4中的解释，即使销毁了数据库里的一个对象，内存副本仍然可以查询属性，只要我们不去修改它或者要求它永久保存。）

难道我们不应该简化相似的事情吗？ 第5章将我给出一种方式，利用这种相似性来简化视图，但是现在为了完成这个例子，请容忍这一点重复性。)

正如对Edit所做的处理一样，我们在电影的Show页面提供Delete行为的访问权限。应该使用哪种HTML元素呢？`rake routes`告诉我们Delete操作需要URI `/movies/:id`和HTTP动词`DELETE`。这看起来和Edit类似，Edit的URI类似但是它使用了`GET`方法。因为我们一直使用URI助手来生成路由，所以在这里仍旧可以使用**link_to**，但是它的行为和你期待的有点不同。

代码文件：ch_rails_intro/code/view/edit_vs_delete_link.html.haml

```
-# Our Edit link from previous example:
= link_to 'Edit info', edit_movie_path(@movie)
-# This Delete link will not really be a link, but a form:
= link_to 'Delete', movie_path(@movie), :method => :delete
```

如果检查这个代码生成的HTML，你将会发现Rails生成了一个包含了不常见属性`data-method="delete"`的链接。在第6章可以看到，这是支持JavaScript的浏览器处理破坏性的`delete`操作的一种更安全的方式。从用户界面的角度来看，破坏性的

操作，例如Delete，使用按钮比使用链接更好操作，所以下面的代码通常更加合适：

代码文件：ch_rails_intro/code/view/delete_button.html.haml

```
= button_to 'Delete', movie_path(@movie), :method => :delete, :confirm => 'Are you sure?'
```

button_to和**link_to**类似——视频4.8.1展示了它是如何生成一个表单的。

视频4.8.1: button_to助手.
光盘文件：ch_rails_intro/mov/button_to.mp4
button_to助手有三个参数：按钮的文本名称，指定路由的可选哈希 **或者** 编码了路由的URI，以及可选的HTML选择的另一个哈希。在我们的使用里，第二个参数是由**movie_path**助手提供的，并且我们在HTML选项里提供了关键词来指定HTTP提交方法以及为用户显示一个确认对话框。结果就是仅有按钮这一个用户可见的控制的表单，它使用了一些JavaScript技巧来提供确认对话框，我们将在第6章里讲到。

■ 详细阐述：GET和POST

在RESTfulness成为重要的SaaS概念之前很长一段时间，已经有一个通用的准则，就是使用GET的SaaS应用请求应该总是“安全的”——它们不会引起任何副作用，例如删除一个项目或者购买某些东西，并且应该是具有安全的可重复性的。浏览器反映了这条准则：你总可以加载或者刷新一个由GET获得的页面，但是如果你尝试加载一个由POST操作得到的页面，大多数浏览器将会给出一个警告，询问你是否真的想要重新提交表单。当搜索引擎使用普及后，“GETs应该是安全的”准则尤其重要，因为它们通过链接来访问Web。如果Google每一次访问一个商业网站就会触发了数百万虚假的采购，这将是非常可怕的。因此，在这个意义上说，Rails的助手仅仅是提供了一个快捷方式，实现现有的最佳实践。

尝试修改`index`视图（所有电影的目录），使得每一个表的行给出电影的名字，并给出一个能够给出电影编辑表单的Edit链接，以及产生一个确认对话框删除电影的Destory按钮。

自我检查4.8.1. 为什么`new.html.haml`里的表单提交给**create**方法而不是**new**方法？
◇ 在第2章中看到的一样，创建一个新的记录需要两个交互。第一个，**new**，用于加载表单。第二个，**create**，用于提交表单并实际创建新的记录。

自我检查4.8.2. 为什么在控制器操作的相同代码路径上，呈现和重定向同时存在（或者两个呈现器，或者两个重定向）是没有意义的？
◇ 呈现和重定向是两种不同的应答请求的方式。每个请求只需要一个应答。

小结：

- Rails参考ActiveRecord模型提供了各种各样的助手来创建HTML表单。在接收表单提交的控制器方法里，**params**哈希表中的关键字是表单字段的name属性，对应的值是用户对这些字段的选择。
- 创建和更新一个对象是资源，其可视性的体现就是请求的成功或者失败的状态。为了用户友好性，可以**redirect_to**到一个更加有用的页面，例如**index**。而不是仅仅给出一个有成功或失败状态并且要求用户点击后续的网页。重定向是完成一个控制器操作除了显示另外一个视图的另一种选择。
- 从用户友好性考虑，修改应用程序的布局来显示存储在**flash[:notice]**或者**flash[:warning]**里的消息是很典型的，这会持续到下一个请求到来，所以它们可以和**redirect_to**一起使用。
- 为了指定表单提交和重定向都需要的URI，可以使用RESTFUL URI助手，像**movies_path**和**edit_movie_path**，而不是手动地创建URI。

4.9 谬论和陷阱

陷阱：手动修改数据库而不是使用迁移，或者手动管理gem而不是使用Bundler。

特别地，是如果你刚使用过其他的SaaS架构，使用SQLite3命令行或者GUI数据库控制台来手动添加或者改变数据库表或者安装库可能很有诱惑力。但是如果你这样做，在将来可能就没有一个一致的方式来重现这些步骤，并且没有办法有条不紊地重新恢复这些变化。此外，因为迁移和Gemfiles仅仅是工程中的一部分的文件，你可以把它们纳入版本控制，并且查看你的所有修改。

陷阱：臃肿的控制器和视图。

因为当用户请求到达时，在应用程序代码中，首先被调用的是控制器操作，所以操作的方法非常容易变得臃肿——把所有种类的逻辑方法都放在真正属于模型的控制器里。同样，代码也很容易蔓延到视图里——最常见的是，一个视图可能会发现它调用了一个模型方法，例如**Movie.all**，而不是使用控制器方法来设置变量，如**@movies=Movie.all**，以及仅仅使用**@movies**来构建视图。除了违反MVC模型外，把视图耦合到模型也会干扰缓存，关于这一点我们将在第5章里探讨。视图应该注重显示内容和便于用户输入，而控制器应该注重视图和模型之间的协调，并且设置任何必要的变量以阻止代码泄漏到视图中。

陷阱：session[]哈希装填过度。

有两个原因使你应该尽量减少放到**session[]**里的内容。第一，默认Rails的配置情况下，在每个请求结束时会话被打包到一个cookie（见2.2.1节和视频2.2.1），当cookie接收到下一个请求时不会被打包，同时HTTP规范把cookies限制到4KB的范围。第二，尽管可以通过修改Rails设置把它们存储在自己的数据库而不是cookie里，来允许更大的会话对象，但是臃肿的会话是一个警告，表明应用程序的操作不是自

包含的。这意味着应用程序并不是非常RESTful，并且可能很难作为一个面向服务的体系结构的一部分来使用。尽管没有什么阻止你给会话分配任意的对象，但是你应该保持会话中必要的对象的**id**，以及保持数据库中模型表里的对象本身。

4.10 结束语：SOA的设计

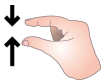
本章内容是Rails简介，这可能看起来像是介绍了很多通用的机制来处理一个相当简单而具体的任务：实现一个基于Web用户界面的CRUD操作。但是我们将会在第5中看到，这种扎实的基础会让我们对先进的机制有更深的了解，这些机制能够让Rails应用程序真正地减少重复且简洁美观。

我们可以立即给出一个与面对服务的体系结构相关的简单例子，这个结构的概念在第1章里介绍过，而且后面还会经常碰到。如果打算在SOA里使用Rotten Potatoes，那么它的RESTful操作有可能由期待看到操作之后出现的Web网页的人来执行，或者由期待（例如）一个XML回应的另一项服务来执行。为了简化应用程序使用SOA工作的工作，可以使用**ActionController**里的方法**respond_to**[17]，对相同的资源返回不同的格式（不要和3.2节里的Ruby内置**respond_to?**方法混淆）。**ActionController::MimeResponds#respond_to**产生一个对象，可以用于选择显示响应的格式。这里给出了**update** 操作是如何被立即转化成一个SOA-友好的RESTful API的，它更新电影的属性，并且返回更新后的对象的XML表示，同时为用户保留现有的用户界面：

代码文件：ch_rails_intro/code/controller/edit_update_with_respond_to.rb

```
def update
  @movie = Movie.find params[:id]
  @movie.update_attributes!(params[:movie])
  respond_to do |client_wants|
    client_wants.html {  redirect_to movie_path(@movie)  } # as before
    client_wants.xml  {  render :xml => @movie.to_xml    }
  end
end
```

类似的，**new**需要自己的控制器操作的唯一原因，就是用户需要去填写用于**create**的值。另外一项 **服务** 从来都不会调用**new**操作。在**create**操作之后重定向回到电影的列表，这样做也是没有意义的：**create** 仅仅可以返回创建的对象的XML表示，或者是被创建的对象的ID。



因此，和我们将会在这本书里使用的很多工具一样，一个简单任务的最初学习曲线可能看起来有点陡峭，但是你很快就会得到回报，因为有了这个强有力的基础，你可以迅速简洁地增添新的功能和特性。

4.11 深入学习

- Rails[18]的在线文档给出了语言、它的类以及Rails 架构的详细信息。
- Ruby Way（Fulton 2006）和Rails 3 Way（Fernandez 2010）深入探索了Ruby和Rails的高级特性。
- PeepCode[19]推出了高质量的视频，以一个非常合理的价格几乎涵盖了Rails系统中的每一个工具和技术（在作者看来）。两部分组成的Meet Rails 3[20] 视频是本章信息的一个很好的补充。

- 在编写不是特定针对你的应用的任何功能的代码之前，请查看一下rubygems[21]和rubyforge[22]（至少），看看是否有人已经创建了一个gem，做了大多数你需要的东西。正如我们在本章中看到的，使用一个gem和向Gemfile里添加一行并重新运行bundle install一样简单。

O. Fernandez. **Rails 3 Way, The (2nd Edition) (Addison-Wesley Professional Ruby Series)**. Addison-Wesley Professional, 2010. ISBN 0321601661. URL http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321601661.

H. Fulton. **The Ruby Way, Second Edition: Solutions and Techniques in Ruby Programming (2nd Edition)**. Addison-Wesley Professional, 2006. ISBN 0672328844. URL http://www.amazon.com/Ruby-Way-Second-Techniques-Programming/dp/0672328844.

注释

[1]http://ruby-doc.org/core-1.9.3/Time.html
[2]http://ruby-doc.org/core-1.9.3/
[3]http://edgeguides.rubyonrails.org/routing.html
[4]http://api.rubyonrails.org/
[5]http://api.rubyonrails.org/
[6]http://en.wikipedia.org/wiki/HTML_sanitization
[7]http://api.rubyonrails.org/classes/ActionView/Helpers/UrlHelper.html#method-i-link_to
[8]http://ruby-doc.org/core-1.9.3/Time.html#method-i-strftime
[9]http://ruby-doc.org/core-1.9.3/Time.html#method-i-strftime
[10]http://catb.org/jargon/html/koans.html
[11]http://stackoverflow.com
[12]http://serverfault.com
[13]http://stackoverflow.com/questions/2945228/i-see-gem-in-gem-list-but-have-no-such-file-to-load
[14]http://api.rubyonrails.org/classes/ActionView/Helpers/FormTagHelper.html
[15]http://guides.rubyonrails.org/security.html#mass-assignment
[16]http://www.w3.org/Amaya
[17]http://api.rubyonrails.org/classes/ActionController/MimeResponds.html#method-i-respond_to
[18]http://api.rubyonrails.org/
[19]http://peepcode.com
[20]https://peepcode.com/products/meet-rails-3-i
[21]http://rubygems.org
[22]http://rubyforge.org

4.12 建议的项目

除非特别声明，这些推荐的作业都是基于在本章中创建的myrottenpotatoes应用程序的。

练习4.1. 修改应用程序的路由，使得访问http://localhost:3000时链接到电影的列表，而不是通常的Rails欢迎页面。
提示：查询ActionDispatch::Routing[1]文档。

练习4.2. 在主要的应用程序布局里添加一个默认横幅，使其出现在Rotten Potatoes的每一页上。横幅以大号红色字体显示“Rotten Potatoes”，但是模板中不要出现任何视觉样式信息。实现它使得每次点击横幅都会链接到RP主页。
提示：挑选一个反映了这个横幅的元素类型，给它分配一个独特的ID，并且修改CSS样式文件来设置元素风格。

练习4.3. 在一个成功的**create**之后不是重定向到Index操作，而是给刚刚创建的新的电影重定向到**show**操作。
提示：可以使用**movie_path**的URI助手，但是你需要提供一个参数识别是哪部电影。为了获得这个参数，如果成功返回新创建的对象的话，就调用**Movie.create**，否则创建对象。

练习4.4. 按照下面操作修改电影目录。每一个修改任务都需要在不同的抽象层做点变化。

(1) 修改Index视图，使得电影表中的每一行都有一个行号。
提示：看视图模板第11行使用的`each_with_index`函数的文档。

(2) 修改Index视图，使得鼠标停留在电影表中的一行时，行背景暂时变成黄色。
提示：查看CSS 支持的`hover` **pseudo-class** 。

(3) 修改Index控制器操作，返回以字母顺序排列标题的电影列表，而不是发布日期。
提示： **不要** 尝试对控制器调用数据库的结果来排序。关系数据库提供了方式来指定结果传递的顺序，并且由于Active Record 和底层关系数据库的紧密耦合，Rails的`ActiveRecord`库里的`find`和`all`方法提供了让底层的关系数据库去做这件事的途径。

(4) 假设你没有Active Record的紧密耦合，你就不能假定底层存储系统可以以任何特定的顺序来返回聚合类型项目。修改Index控制器操作，返回以字母序排列标题的电影。
提示：查看Ruby的`Enumerable` 模块里的`sort`方法。

练习4.5. 如果用户在提交Create或者Update表单之前改变了主意，不准备继续进行下去了，该怎么办？在表单中添加“取消”链接，使得用户能够返回到电影的目录。

练习4.6. 修改“取消”链接，使得如果它是作为Create流程的一部分被点击的话，用户会被带回到电影的目录；但是如果是作为Update流程中的一部分被点击的话，用户就会被带回到他开始编辑的电影的Show模板（视图）。
提示：如果它的接收者是一个新的模型对象，那么实例方法**ActiveRecord::Base#new_record?**返回真，也就是说，一个从来没有在数据库里保存过的对象，没有ID。

练习4.7. Release Date的下拉菜单不允许添加发行日期早于2006年的电影。修改它使得可以添加早到1930年的电影。（提示：查看文档[2]中，在表单中使用的**date_select**助手。）

练习4.8. 电影的**description**字段是作为初始迁移的一部分创建的，但是目前这个字段还没有显示，也不能编辑。做一些必要的变化，使得description在Show视图中是可见的，在New和Edit视图中可编辑。
提示：你仅仅需要改变两个文件。

练习4.9. 当前的控制器方法还不够健壮：如果用户手动地进入一个URI来显示一个不存在的电影（例如`/movies/99999`），将会看到一个丑陋的异常消息。修改控制器中的**show**方法，使得如果请求的电影不存在，用户就会重定向到Index视图，并给出一个友好的提示消息指出给定的ID的电影找不到。
提示： 你可以使用**find_by_id**并检查结果，或者继续使用**find**，但是把它放到**begin...rescue...end**里，并且从**ActiveRecord::RecordNotFound**中恢复出来。哪个选择更地道，众说纷纭；一般来说，异常是用来显示未料到的事情，但是我们并不明确这样的事件是真的出乎意料还是单纯的少见。

练习4.10. 把所有的练习放到一起：编写和部署一个Rails应用程序，使用Nokogiri的XPath特性从Web页面上抓取某种信息，并且用Builder把它转换成RSS feed。验证一下否可以在浏览器或者RSS新闻阅读器里订阅RSS feed。

5 SaaS框架：Rails高级功能

编程就是理解。

——Kristen Nygaard

Kristen Nygaard（左，1926——2002）和**Ole-Johan Dahl**（右，1931——2002）一起获得了2001年的图灵奖，因为他们发明了基本的OO概念，包括对象、类和继承，并且在每一个面向对象语言的祖先Simula 里演示了这些概念。

5.1 去除MVC中的重复：Partials、验证和过滤器 128
5.2 单点登录和第三方认证 134
5.3 关联和外键 138
5.4 间接关联 142
5.5 关联的RESTful路由 145
5.6 组合可重用作用域的查询 148
5.7 谬论和陷阱 150
5.8 结束语：语言、生产力和美观 151
5.9 深入学习 151
5.10 建议的项目 152

重要概念

本章介绍了Rails的高级特性，可以使用它们让代码更加地简洁，包括如何重用外部的服务例如Twitter来与应用程序整合。

- Rails机制，例如控制器过滤器、模型生命周期钩子以及模型验证，提供了一个有限的 **面向方面编程** 的形式，这使得关于横切关注点的代码集中在一个地方，并且在需要时自动调用。
- ActiveRecord **关联** 使用了元编程和反射来映射应用程序中资源间的关系，例如"belongs to"或者"has many"，来查询应用程序数据库中的这些镜像关系。
- ActiveRecord **scopes** 是可以在模型数据中定义的可组合的"过滤器"，它能够在模型逻辑中通过重用实现无重复。

代码文件：ch_advanced_rails/code/view/form_partial.html.haml

```
-# in _movie_form.html.haml (the partial)

= label :movie, :title, 'Title'
= text_field :movie, 'title'

= label :movie, :rating, 'Rating'
= select :movie, :rating, Movie.all_ratings

= label :movie, :release_date, 'Released On'
= date_select :movie, :release_date
```

代码文件：ch_advanced_rails/code/view/new_with_partial.html.haml

```
-# new.html.haml using partial

%h2 Create New Movie

= form_tag '/movies', :method => :post do
  = render :partial => 'movie_form'
  = submit_tag 'Save Changes'
```

代码文件：ch_advanced_rails/code/view/edit_with_partial.html.haml

```
-# edit.html.haml using partial

%h2 Edit Existing Movie

= form_tag movie_path(@movie), :method => :put do
  = render :partial => 'movie_form'
  = submit_tag 'Update Movie Info'
```

正如6.6节解释的一样，partial也是JavaScript实现的页面的视图更新的基本单元。

图 5.1：partial（上），展示了new和edit模板共有的表单元素。修改后的new和edit模板都使用了partial（两个片段中的第6行），但是两个模板中的第5行是不同的，因为new和edit提交了不同的操作（请参见本书配套光盘中的代码）

5.1 去除MVC中的重复：Partials、验证和过滤器

我们将会集中讨论MVC的三个元素的无重复（DRY）特性，从视图开始。

partial 是Rails里一个视图的可重用块的名称。当类似的内容必须出现在不同的视图里时，把这个内容放到一个partial 里，把它“包含”在一个单独的文件里有助于减少重复。例如，在第4章里，图4.14 给出了**new**和**edit**操作之间的相似性，这二者的视图都显示出相同的表格来访问电影信息——并不是完全无重复。图5.1 把这种共同的视图代码放到一个partial里，并且给出了如何修改现有的`new.html.haml`和`edit.html.haml`来使用它。

难道用户没有从菜单中选择分级？ 不，他们选了，但是这个请求可能是被恶意用户或者 **bot** 构建的。有了SaaS，你不能相信任何人：服务器必须 **总是** 检查它的输入而不是去信任它们，否则就会有被方法攻击的风险，这一点会在第12章里讲到。

Partials非常依赖惯例优先配置原则。它们的名称必须以下划线开头（我们用的是`_movie_form.html.haml`），这在引用partial的代码里是 **没有的** 。如果一个partial和使用它的视图不在相同的目录，那么**'layouts/footer'**会导致Rails寻找`app/views/layouts/_footer.html.haml`。partial可以访问包含它的视图能访问的所有实例变量。partial的一个非常漂亮的用法是呈现一个表或者其他集合，其中所有的元素都是一样的，如图5.2所示。

Partials是简单直白的，但是Rails提供的消除模型和控制器中重复的机制确实是更加微妙和复杂的。在SaaS应用程序里，想要对一个给定类型的模型对象执行某个有效性约束，或者在实现某个操作时执行约束，这是常见的。例如，当一部新电影添加到RottenPotatoes里时，我们可能想确定一下标题不是空白的，发行年份是一个有效日期，以及分级是允许的分级中的一个。再举一个例子，也许我们想允许任何用户添加新电影，但是仅允许特殊的“管理员”用户删除电影。 这两个例子都涉及指定实体或者操作上的约束，虽然在应用程序里有很多地方可能需要考虑这样的约

代码文件：ch_advanced_rails/code/view/render_collection.html.haml

```
1 -# A single row of the All Movies table
2 %tr
3   %td= movie.title
4   %td= movie.rating
5   %td= movie.release_date
6   %td= link_to "More about #{movie.title}", movie_path(movie)
```

图 5.2: 如果这个partial存为views/movies/_movie.html.haml，原始的index.html.haml视图（见图4.6）中的第12～17行可以用render :partial=>'movie', :collection=>@movies来替换。按照惯例优先配置原则，没有前导下划线的partial名称（这里是电影）在partial里是作为一个局部变量来用的，它反过来被设为@movies中的每个元素

束，但是DRY 这一观点敦促我们把它们集中到 **一个** 地方。Rails提供了两个类似的设施来做这个：模型验证以及控制器过滤器。

模型验证，例如迁移，是在Ruby嵌入的迷你特定域语言（DSL）中表达出来的，如图5.3所示。调用实例方法**valid?** 或者尝试把模型保存到数据库里（在做这个之前也会调用**valid?**）时，模型检查就会被调用和触发。

任何的验证错误都会被记录在和每个模型关联的**ActiveModel::Errors**[5]对象里；这个对象是由实例方法**errors**返回的。如图5.3所示，可以查询个别属性上的错误（第18～20行），或者使用**full_messages**获取所有的错误，它们是一个字符串数组。当编写自定义验证时，如行7～10，可以使用**errors.add**来添加错误信息，这个信息可以是关于对象特定无效属性的，也可以是与关于对象本身的（通过传递**:base**而不是一个属性名）。

这个例子也表明验证可以是有条件的。例如，图5.3中的第6行保证了电影的分级是有效的， **除非** 电影是在分级系统生效（在美国是1968年11月1日）之前发行的，在这种情况下不需要验证分级的有效性。

可以利用模型验证，用更安全的版本替换控制器操作中的危险的**save!**和**update_attributes!**方法，**save**和**update_attributes**，如果验证失败，它们就会操作失败。图5.4给出了如何利用这一点来修改控制器的方法，使它们变得更加符合惯用法。修改**create**和**update**控制器操作来匹配，修改`app/models/movie.rb`使它包括图5.3中的验证，然后尝试添加一部违反了其中一个或多个验证的电影。

当然，如果能给用户提供一个有信息量的消息指出了哪里出错以及他应该做什么，这将是非常好的做法。这很简单：在视图里，我们 **可以** 获得要求呈现视图的控制器操作的名称，然后将它包含在错误信息里，像下面代码的第5行所做的那样。

代码文件：ch_advanced_rails/code/view/partial_with_message.html.haml

```
1 -# insert at top of _movie_form.html.haml
2
3 - unless @movie.errors.empty?
4   #warning
5     Errors prevented this movie from being #{controller.action_name}d:
6     %ul
7       - @movie.errors.full_messages.each do |error|
8         %li= error
```

视频5.1.1做了进一步的探索，它解释了Rails视图是如何利用能够分别查询每个模型属性的有效性，提供给用户一个更好的视觉反馈信息用以指出是哪些字段引起了这个问题。

代码文件：ch_advanced_rails/code/model/validation_example.rb

```
1  class Movie < ActiveRecord::Base
2    def self.all_ratings ; %w[G PG PG-13 R NC-17] ; end #  shortcut: array of
         strings
3    validates :title, :presence => true
4    validates :release_date, :presence => true
5    validate :released_1930_or_later # uses custom validator below
6    validates :rating, :inclusion => {:in => Movie.all_ratings},
7      :unless => :grandfathered?
8    def released_1930_or_later
9      errors.add(:release_date, 'must be 1930 or later') if
10       release_date && release_date < Date.parse('1 Jan 1930')
11   end
12   @@grandfathered_date = Date.parse('1 Nov 1968')
13   def grandfathered?
14       release_date && release_date >= @@grandfathered_date
15   end
16 end
17 # try in console:
18 m = Movie.new(:title => '', :rating => 'RG', :release_date => '1929-01-01')
19 # force validation checks to be performed:
20 m.valid?  # => false
21 m.errors[:title] # => ["can't be blank"]
22 m.errors[:rating] # => ["is not included in the list"]
23 m.errors[:release_date] # => ["must be 1930 or later"]
24 m.errors.full_messages # => ["Title can't be blank", "Rating is not
25   included in the list", "Release date must be 1930 or later"]
```

图 5.3: 第3～5行在ActiveModel::Validations::ClassMethods[4] 里使用了预定义的验证行为。第6～15行给出了你应该如何创建你自己的验证方法，它接收要被验证的对象作为一个参数，然后添加描述任何问题的错误信息。注意首先验证release_date的存在，否则第10行和14行的比较会失败，如果release_date是nil的话（我们可能已经在released_1930_or_later方法里验证过这一点了，但是做两次验证强调了这两个条件都要是正确的）

代码文件：ch_advanced_rails/code/ctrl/controller_with_validation.rb

```
1  # replaces the 'create' method in controller:
2  def create
3    @movie = Movie.new(params[:movie])
4    if @movie.save
5      flash[:notice] = "#{@movie.title} was successfully created."
6      redirect_to movies_path
7    else
8      render 'new' # note, 'new' template can access @movie's field values!
9    end
10 end
11 # replaces the 'update' method in controller:
12 def update
13   @movie = Movie.find params[:id]
14   if @movie.update_attributes(params[:movie])
15     flash[:notice] = "#{@movie.title} was successfully updated."
16     redirect_to movie_path(@movie)
17   else
18     render 'edit' # note, 'edit' template can access @movie's field values!
19   end
20 end
21 # as a reminder, here is the original 'new' method:
22 def new
23   @movie = Movie.new
24 end
```

图 5.4: 如果create或者update失败，那么就使用一个显式的render来再次呈现这个表格，使得用户可以再次填写它。方便的是，@movie将会对视图可用，并且会保持用户第一次输入的值，所以表单将会用这些值来填充，因为（按照惯例优先配置原则）_movie_form.html.haml 使用的表单标签助手将会使用@movie来填充表单字段，只要它是一个有效的Movie对象

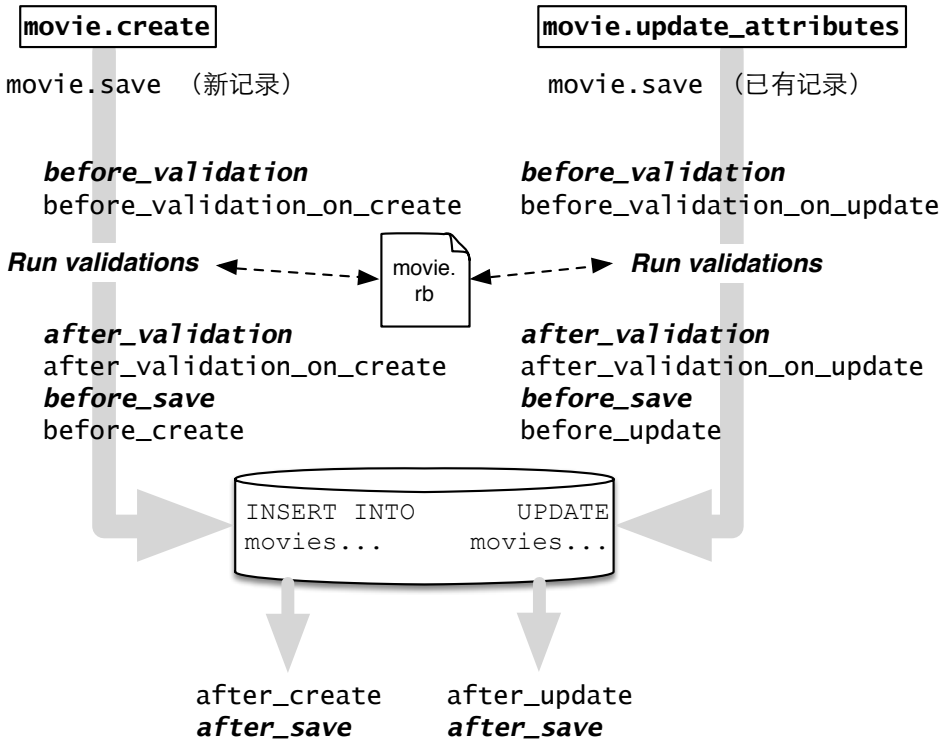


图 5.5: 各种各样可以"用钩子连接起"ActiveRecord模型对象的生命周期上的点。所有修改了数据库的ActiveRecord操作（update、create等等）最终都会调用save，所有before_save回调可以拦截数据库的每一个修改。查看这个Rails指南[7]中的额外的细节和事例

视频5.1.1: 模型验证是如何与控制器以及视图交互的.
光盘文件：ch_advanced_rails/mov/validation_in_views.mp4
视图中的表单助手使用**errors**对象发现哪些字段引起了验证错误，并且应用特殊的CSS类`field-with-errors`到任何这样的字段。从用户的利益考虑，通过在`app/assets/stylesheets/application.css`中的类中包含选择器，可以直观地高亮显示受影响的字段。

实际上，验证只是更加普遍的机制里的一个特殊的例子，Active Record 生命周期中的回调[8]，它允许你提供方法，在它的生命周期的各种相关的点上"拦截"一个模型对象。图5.5 给出了哪些回调是可用的；图5.6说明了在模型保存之前，如何使用这个机制来"规范化"（标准化格式）某个模型的字段。在第11章里讨论Observer设计模式以及在第12章里讨论缓存时，将会看到生命周期回调的另一种用法。

和验证类似的是控制器过滤器——一种在行为运行之前检查某些条件是否是真，或者设置一些很多行为依赖的常见条件的方法。如果条件不满足，那么过滤器可以选择通过渲染一个视图模板或者重定向到另外一个操作上"停止显示"。如果过滤器允许这个行为继续进行，那么和往常一样，提供一个应答就是这个行为的责任了。

作为一个例子，过滤器的一个非常常见的用途就是执行用户登录的请求，在某

代码文件：ch_advanced_rails/code/model/before_save_example.rb

```
class Movie < ActiveRecord::Base
  before_save :capitalize_title
  def capitalize_title
    self.title = self.title.split(/\s+/).map(&:downcase).
      map(&:capitalize).join(' ')
  end
end
# now try in console:
m = Movie.create!(:title => 'STAR  wars', :release_date => '27-5-1977')
m.title  # => "Star Wars"
```

图 5.6: 这个before_save的钩子将电影标题的每一个单词的首字母转为大写，把剩下的字母转换称小写，并且把词之间的多个空格压缩为一个，比如把STAR wars转换成Star Wars（对电影标题来说并不一定是正确的行为，但是有助于说明）

代码文件：ch_advanced_rails/code/auth/application_controller.rb

```
class ApplicationController < ActionController::Base
  before_filter :set_current_user
  protected # prevents method from being invoked by a route
  def set_current_user
    # we exploit the fact that find_by_id(nil) returns nil
    @current_user ||= Moviegoer.find_by_id(session[:user_id])
    redirect_to login_path and return unless @current_user
  end
end
```

图 5.7: 如果有一个已登录的用户，那么重定向 不会 发生，并且控制器实例变量@current_user对行为和视图都是可用的。否则会重定向到login_path上，假定它是响应把用户带到登录页面的路由的，在5.2节里会给出更多相关内容。（and和&&一样，但是它优先级更低，因此这相当于完整括号的版本是((redirect_to login_path) and (return)) unless...）

些操作执行之前。假设目前已经验证了某个用户的身份，并且在**session[:user_id]**里存储了他的主键（ID）来记住他已经登录的事实。图5.7给出了一个过滤器，强制要求有效用户登录。5.2节将会给出如何把前置过滤器和其他涉及的“移动部分”结合起来，来处理已登录的用户。

过滤器通常使用于控制器中的所有行为，但是**:only**可以用来指定过滤器仅仅用于某些行为，而**:except**可以用来指定某些行为是不用的。每一个都带了一个行为名称的数组。你可以定义多个过滤器：它们按照被声明的顺序执行。也可以定义后置过滤器，它们在某些行为完成之后运行，以及“包围”过滤器，它们在行为运行之前和之后运行，就像你做审计或者计时的那样。“包围”过滤器使用了**yield** 来实际运行控制器行为：

代码文件：ch_advanced_rails/code/ctrl/around_filter_example.rb

```
# somewhat contrived example of an around-filter
around_filter :only => ['withdraw_money', 'transfer_money'] do
  # log who is trying to move money around
  start = Time.now
  yield   # do the action
  # note how long it took
  logger.info params
  logger.info (Time.now - start)
end
```

Rails消除MVC中冗余的小结：

- Partials允许在不同的模板中重用视图块，把相同的视图元素收集到一个地方。
- 验证让你把模型上的约束集中到一个地方。每一次数据库要修改时都要进行验证；验证失败是非危险的**save**和**update_attributes**一种可能的失败的方式。
- 模型的**errors**字段是一个**ActiveRecord::Errors**对象，记录了验证期间出现的错误。视图表单助手可以使用这个信息来把特殊的CSS样式应用到这些值验证失败的字段里，它把检测错误的责任和为用户显示错误的责任彻底地分开了。
- 通过定义一个总是在控制器行为之前运行的方法，控制器过滤器让你把影响这些行为的条件收集到一个地方。控制器里声明的过滤器影响控制器中的所有行为，而**ApplicationController**里声明的过滤器影响所有控制器里的所有行为，除非**:only**或者**:except**被指定。

■ **详细阐述：面向方面编程**

面向方面编程 （AOP）是通过把 **横切关注点** ，例如模型验证和控制器过滤器，和关注点适用的行为主要代码分开，从而减少代码的重复的编程方法。在示例中，以声明的方式指定模型验证到一个地方，而不是在想要验证检查的代码中的每个 **切入点** 显式地调用它们。这个连接点的集合统称为 **切入点** ，而将要被插入到每个连接点（如示例里的验证）的代码称作 **完整的AOP** 。

Ruby不支持完整的AOP，完整的AOP允许你指定任意的切入点以及哪些建议适用于每个切入点。但是Rails使用Ruby的动态语言特性来定义方便的切入点，例如AR模型生命周期，它支持验证以及其他的生命周期回调，还有控制器行为周围的连接点，它支持前置和后置过滤器的使用。

对AOP的一个批评就是源代码不能再以线性顺序阅读。例如，当一个前过滤器阻止控制器行为执行时，这个问题会很难追踪，尤其对某些不熟悉Rails的人来说，他们没有意识到过滤器方法甚至没有被显式调用，而是一个由特定的连接点触发的建议方法。对这个批评的一个回应是如果保守且雅致地使用AOP，并且所有的开发者都能理解并且赞成切入点，那么它可以提高无重复性和模块化。验证和过滤器是Rails设计者对识别这个有益的中间区域的一个尝试。

自我检查5.1.1. 当你第一次使用**Movie#new**来实例化时，为什么Rails设计者没有选择立刻触发验证，而是等到你试图保存这个对象时？

◇ 当填写新对象的属性时，它可能是一个暂时无效的状态，所以在这个时候触发验证可能难以操作这个对象。保存这个对象就相当于告诉Rails“我相信这个对象准备好保存了”。

自我检查5.1.2. 为什么我们不能写**validate released_1930_or_later**，也就是说，为什么**validate**的参数必须是一个符号或者字符串？

◇ 如果这个参数仅仅是方法的一个“空的”名称，那么Ruby会在它执行**validate**时计算它的值，这不是我们想要的——我们需要在每一次验证即将发生时调用**released_1930_or_later**。

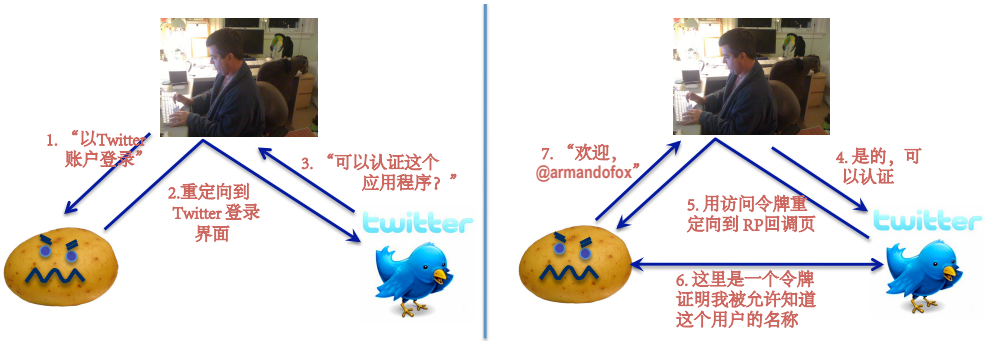


图 5.8: 第三方认证启用SSO，通过允许SaaS应用要求用户通过第三方提供商认证自己。一旦他这样做了，提供商发送一个令牌给提出请求的应用程序，证明用户认证正确，并可能编码用户授予给请求的应用程序的额外的特权。所示的流程是 OAuth 的一个简化版本，它是一个逐渐发展（以及稍有争议的）开放的认证和授权的标准，被Twitter、Facebook、Microsoft、Google、Netflix以及很多其他的站点使用。Twitter标志和图像版权为Twitter公司所有，仅用于教学目的

5.2 单点登录和第三方认证

授权 指的是一个主体是否被允许做某件事。尽管和认证独立，但是这两者常常混为一谈，因为很多标准同时使用二者。

一种更加无重复（DRY）和高效的方式是避免实现一些你可以从其他服务里重用来的功能。今天这个方面的一个例子就是 **认证** ——通过这个过程实体或称为 **主体** 能够证明它就是自己声称的实体或人。在SaaS中，终端用户和服务器是两种常见的可能需要认证的主体。通常，用户通过提供一个用户名以及（假定）别人不知道的密码来证明其身份，而服务器用 **服务器证书**（第12章会讨论到）来证明身份，证书的 **起源** 可以使用密码学来验证。

Facebook是SSO的一个早期例子。

在早期的SaaS中，用户必须对每个网站建立独立的用户名和密码。而今天，一个越来越常见的现象是 **单点登录**（single sign-on，SSO），其中一个站点建立的证书（**提供者**）可以用于登录与它在管理上无关的其他站点。显然，SSO是面向服务体系结构有用性的核心：如果每个服务都有它独立的身份验证方案，很难一起为你工作。鉴于SSO的普遍性和日益增长的重要性，我们的观点就是新的SaaS应用应该使用它，而不是“实现他们自己的”认证。

然而，SSO使我们陷入了进退两难的困境，当你很高兴地在站点A上使用证书去登录站点B时，你通常不想在站点B上显示这些密码。（试想一下，站点A是你的财务机构，而站点B是你想在上面买东西的外国公司。）图5.8以Rotten Potatoes和Twitter 作为例子，给出了 **第三方认证** 是如何解决这个问题的。首先，请求认证的应用程序（Rotten Potatoes）创建一个请求发送到认证提供者，在那里用户已经有一个账户了，在这个例子里是Twitter。请求通常包含应用程序想向提供商要求的特权相关的信息，例如，作为这个用户能上Twitter，或者了解这个用户的追随者都是谁。

SSO过程通常是以一个用户必须单击的链接或者按钮开始的。这个链接把用户带到 **提供商** 服务的安全登录页面；根据实现而定，登录页面可能是弹出菜单、HTML框架或iframe站点[9]，或者一个是提供商的站点服务的常规页面。然后用户就有机会登录到提供者，并且决定是否授予应用程序请求的特权。尤为重要的是，这种交互完全发生在用户和提供者之间：提出请求的应用程序没有权限参与交互的任何一个部分。一旦认证成功，提供者生成一个 **回调** 给发出请求的应用程序，给它一个 **访问令牌** ——使用加密技术创建的一个字符串，此令牌可以随后传回给提供者，提供者可以确信验证令牌一定是在成功登录时创建的。这时，发出请求的应用程序可以去做两件事情：

代码文件：ch_advanced_rails/code/model/create_migration.txt

```
rails generate model Moviegoer name:string provider:string uid:string
```

代码文件：ch_advanced_rails/code/model/moviegoer_auth.rb

```
# Edit app/models/moviegoer.rb to look like this:
class Moviegoer < ActiveRecord::Base
  attr_protected :uid, :provider, :name # see text for explanation
  def self.create_with_omniauth(auth)
    Moviegoer.create!(
      :provider => auth["provider"],
      :uid => auth["uid"],
      :name => auth["info"]["name"])
  end
end
```

图 5.9: 上图（a）：在终端中输入这个命令来创建一个moviegoers模型和迁移，然后运行rake db:migrate应用这个迁移。下图（b）：然后编辑生成的app/models/moviegoer.rb文件来匹配这个代码，文本中已解释过

(1) 可以相信用户已经对提供者证明其身份，并且选择性地为用户记录提供者的持久并全局唯一的ID，又可称为 **guid** （发音GOO-id），这通常是作为访问令牌的一部分提供的。例如，Armando Fox在Twitter上的guid恰巧是318094297，尽管除非它伴随一个访问令牌，授权获取关于这个guid的信息，否则这个信息没有用处。

(2) 可以使用令牌向提供者进一步请求用户信息，这取决于成功认证时一块授权的哪些特定的权力。例如，来自Facebook 的一个令牌可能表明用户同意应用程序去了解他的朋友是谁，但是不允许应用程序把这发布到他的Facebook页面上。

令人高兴的是，在Rails应用程序里添加第三方认证相当简单直白。当然，在可以让用户登录之前，需要能够在程序中表示用户！所以在继续之前，按照图5.9中的说明创建一个基本的模型和迁移。

Rails通过三个方面来管理第三方认证。首先就是如何实际上通过第三方来认证用户。我们将会使用优秀的OmniAuth[10] gem，它在图5.8中通过允许开发者为每一个第三方身份验证提供者创建一个 **策略** ，来抽象出整个过程。一个策略处理所有与身份验证提供者之间的交互（图5.8 中的第2∼4步），并且最终在应用程序的对URI /auth/ **provider** /callback执行HTTP POST。和POST 一起包括的这些数据表明认证过程的成功或者失败，并且如果成功，将获得访问令牌，应用程序可以用来获取关于已登录用户的额外信息。截至撰写本文时，策略可以用于Facebook、Twitter、Google Apps、新浪微博以及很多其他的东西，每一个可以作为一个命名为omniauth- **provider** 的gem来用。我们将使用Twitter作为例子，所以把gem 'omniauth-twitter'添加到你的Gemfile，然后像往常一样运行bundle install --without production。图5.10（a）给出了如何添加OmniAuth策略在它完成Twitter的认证时期望使用的路由。

认证处理的第二个方面就是记录当前的用户是否已经被认证了。你可能已经猜到，这个信息可以存储在**session[]**里。但是，我们应该把会话管理和应用程序的其他事情分开，因为如果应用程序使用在面向服务的体系结构中，会话可能不相关。为此，图5.10（b）给出了当用户成功认证时（第3∼9行）如何“创建”一个会话，以及当用户注销时（第11∼15行）如何“摧毁”它。这里有个“双引号”，因为实际上唯一被创建或者摧毁的对象就是**session[:user_id]**的值，在会话期间是已登录用户的主键，而在其他时候就是**nil**。图5.10（c）给出了这个检查是如何被设置了相应的**@current_user**的**ApplicationController**（所有的控制器都会继承它）中

代码文件：ch_advanced_rails/code/auth/routes.rb

```
get  'auth/:provider/callback' => 'sessions#create',:as => 'login'
post 'logout' => 'sessions#destroy'
get  'auth/failure' => 'sessions#failure'
```

代码文件：ch_advanced_rails/code/auth/sessions_controller.rb

```
class SessionsController < ApplicationController
  # user shouldn't have to be logged in!
  skip_before_filter :set_current_user
  def create
    auth=request.env["omniauth.auth"]
    user=Moviegoer.find_by_provider_and_uid(auth["provider"],auth["uid"]) ||
      Moviegoer.create_with_omniauth(auth)
    session[:user_id] = user.id
    redirect_to movies_path
  end
  def destroy
    session.delete(:user_id)
    flash[:notice] = 'Logged out successfully.'
    redirect_to movies_path
  end
end
```

代码文件：ch_advanced_rails/code/auth/login_form.html.haml

```
#login
  - if @current_user
    %p.welcome  Welcome, #{@current_user.name}!
    = link_to 'Log Out', logout_path
  - else
    %p.login= link_to 'Log in with your Twitter account', login_path
```

图 5.10：典型的Rails应用程序认证流程中的可动部分。上图：三条路由遵循了OmniAuth gem的惯例在一个独立的SessionsController 里映射create和destroy行为，加上一个在将来可以处理身份认证失败的路由（例如，用户输入错误的Twitter密码或者拒绝访问应用程序。）为了方便，我们指定想要助手login_path和login_url作为第一条路由里的快捷方式。中图：第3行跳过了在图5.7中添加到ApplicationController里的before_filter，来避免当用户第一次登录时的无限循环。（另外一个策略就是删除图5.7中的第7行，并且让每个控制器指定它自己的控制器怎么做，如果@current_user是nil的话。）一旦给定用户登录成功，create行为会记住会话中用户的主键（ID），直到destroy 行为被调用才会忘记主键。下图：@current_user变量(ApplicationController中第6行设置的，图5.7) 可以被登录部分用来显示合适的消息。用render:partial=>'sessions/login'包含application.html.haml上的这部分

的**before_filter**抽象出来的，这使得控制器方法或者视图仅仅看**@current_user**即可，而不必耦合到用户是如何认证的这些细节。

第三个方面就是把自己的用户身份的代表——也就是说，`moviegoers`中的主键——与认证提供者的代表链接起来，例如Twitter例子中的`uid`。因为我们想扩大客户将来可以用的身份认证提供商，图5.9(a)中的迁移创建了**Moviegoer**模型，指定了`uid`字段和`provider`字段。Alice第一次用她的Twitter ID来登录RottenPotatoes，会发生什么？会话控制器（图5.10(b)）中的第6行里的`find_by_provider_and_uid`查询将会返回**nil**，所以**Moviegoer.create_with_omniauth**（图5.9（b），第5∼10行）将会被调用来为这个用户创建一个新的记录。从我们的观点来说，“被Twitter验证的Alice”和“被Facebook验证的Alice”将会成为不同的用户，因为我们没有方法去了解这些代表的是同一个人。这就是为什么某些支持多个第三方认证提供者的站点提供给用户一种方式来“链接”两个账户，来表明它们识别的是同一个人。

这可能看起来像很多移动的部分，但是和没有一个抽象来完成相同任务例如OmniAuth相比，这是很干净的代码：我们添加了少于24行的代码，并且通过合并更多的OmniAuth策略，本质上不用新的工作就可以支持其他的第三方认证提供者。视频5.2.1 和这段代码相关的用户体验。

视频5.2.1: 用Twitter账户登录RottenPotatoes.
光盘文件：ch_advanced_rails/mov/rottenpotatoes_twitter_login.mp4
这个版本的RottenPotatoes，如正文中描述的一样，被修改为使用OmniAuth gem，允许影迷用他们已有的Twitter ID来登录。

然而，我们刚刚创建了一个安全漏洞。到目前为止，我们已经在ActiveRecord对象上利用了**params[]**里的“集中复制” 的遍历性，比如在**MoviesController#update**里写下**@movie.update_attributes(params[:movie])**。但是如果一个恶意攻击者制作了一个表单提交，尝试修改**params[:moviegoer][:uid]**或者**params[:moviegoer][:provider]** ——应该仅能被认证逻辑修改的字段——通过提交命名为`params[moviegoer][uid]`的 **隐藏表单字段** 等等，这种情况下应该怎么办？图5.9(b)，第3行给出了如何“保护”这样的敏感属性，避免它们同**params**里的其它参数一起赋值。限制性更强的**attr_accessible**[11] **仅** 能指明我们允许**params[]**（或任何哈希值）中哪些键可以在集中赋值的时候使用。这种更加严格的机制是我们推荐的，因为它遵循了计算机安全里的 **最小特权原则** ，12.9节将讨论如何保护客户数据时会再涉及这个话题。

小结：

- 单点用户登录指的是终端用户体验，其中一组证书（例如他们的Google或者Facebook用户名和密码）将会让用户登录进各种各样的不同服务。
- 使用了像 **OAuth** 这样的标准的第三方认证，是实现单点登录的一种方式：发出请求的应用程序可以通过一个认证提供者来验证用户的身份，而不用用户向发出请求的应用程序暴露其密码。
- 在Rails应用程序中最彻底地分析出认证的方式就是抽象出会话的概念。当用户成功认证之后（可能使用OmniAuth[12]）这样的框架，一个会话就通过把被认证用户的`id`（主键）存储在**session[]** 中被创建了。当注销时，会话就会通过从**session[]**里删掉信息而被销毁。
- 使用**attr_protected**和**attr_accessible**来识别那些"敏感的"模型属性，例如用于会话管理和认证的用户ID信息，并且通过散列把它们排除在集中赋值之外。

■ **详细阐述：SSO副作用**

在某些情况下，使用SSO也会启动其他特性；例如，Facebook Connect允许站点利用Facebook的社交网络，这使得（例如）一旦Bob用Facebook 来认证到纽约时报，他就可以看到他的朋友一直在读纽约时报的哪篇文章。虽然这些吸引人的特性进一步加强了使用SSO的理由，而不是"实现你自己的"认证，但是它们和这里讨论的SSO的基本概念是分开的。

自我检查5.2.1. 简要描述RottenPotatoes是怎样让你用Twitter ID登录进去的，而不用向它显示Twitter密码。

◇ RottenPotatoes把你重定向到Twitter支持的页面，在那里你像往常一样登录。重定向包括一个URL，Twitter会向它传回一个信息，确认你已经认证过自己了，并且以你作为Twitter用户的角度指出RottenPotatoes会采取哪些行为。

自我检查5.2.2. 对或错：如果你用Twitter ID登录进了RottenPotatoes，RottenPotatoes就能够用你的Twitter ID发布Twitter内容。

◇ 错误。认证和权限是分开来的。大多数的第三方认证提供者，包括Twitter，允许发出请求的应用程序请求权限去做特定的事情，并且由用户决定是否要允许它这样做。

5.3 关联和外键

在软件体系结构中 **关联** 是两种类型的实体之间的逻辑关系。例如，我们可能会在Rotten Potatoes中添加**Review**和**Moviegoer**来允许个人的用户写下他们对最喜爱的电影的评论，可以通过建立一个评论到电影(每一个评论属于一部电影)以及评论到观众（每一个评论是由一位观众写的）的一对多的关联做到这点。图5.11使用一种类似的 **统一建模语言（UML）** 表来给出这些关联。我们将会在第11章里看到更多UML的例子。

根据Rails的说法，图5.11表明：

图 5.11: 每一个关联的结尾都以它的 基数 来标记，或者是参加到关联的“这一侧”的实体的数目，带星号表示“0或者更多”。在图中，每一个评论属于单一的观众和一部电影，一个没有观众或者没有电影的评论是不被允许的（基数符号“0..1”而不是“1”允许“孤行”评论）

id	title	rating
13	Inception	PG-13
41	Star Wars	PG
43	It's Complicated	R

id	movie_id	moviegoer_id	potatoes
21	41	1	5
22	13	2	3
23	13	1	4

id	username
1	alice
2	bob
3	carol

图 5.12: 在本图中，Alice给了Star Wars 5个土豆点，给了Inception 4个土豆点，而Bob给了Inception 3个土豆点，Carol还没有给出任何评论，而且没有人评论过It's Complicated。为了简洁清晰，`movies`和`reviews`表的其他字段没有显示出来

- 一个观众有很多评论。
- 一部电影有很多评论。
- 一个评论属于一个观众以及一部电影。

在Rails里，模型对象的“永久的家”是数据库，所以需要一种方式来表示存在数据库中的对象之间的关联。幸运的是，关联是如此的普遍以至于关系型数据库提供了一个特殊的机制来支持它们：**外键**。外键是表中的一列，它的工作就是引用另一个表的主键来建立这些表中的对象之间的关联。回想一下，在默认情况下，Rails的迁移创建了表，它们的主键列称作`id`。图5.12给出了一个观众表来记录不同的用户，一个评论表带有外键列`moviegoer_id`和`movie_id`，允许每一个评论查阅它的作者的主键（`ids`）以及它所评论的电影。

例如，为了找出 **Star Wars** 的所有评论，首先要形成`movies`和`reviews`的所有行的**笛卡尔积**，通过把`movies`表的每一行和`reviews`表的每一可能的行连接起来。这会给出一个9行（因为有3部电影和3个评论）、7列（3列来自于`movies`，4列来自于`reviews`）的新表。从这个大表中只挑出这些`movies`表中的`id`等于`reviews`表中的`movie_id`的行，也就是说，仅仅这些电影-评论组合，其中评论就是关于这部电影的。最后，只选出这些行，其中电影的`id`（因此也是评论的`movie_id`）等于41的，也就是 **Star Wars** 的主键ID。这个简单的例子（以关系型数据库的说法，称作**连接**（join））描述了如何对一组有统一数据布局的表使用小部分的操作符（关系代数），来表示和操作复杂的关系。在SQL中几乎所有的关系型数据库都使用的结构化查询语言里，查询方式如下：

代码文件：ch_advanced_rails/code/movies_reviews.sql

```
SELECT reviews.*
  FROM movies JOIN reviews ON movies.id=reviews.movie_id
  WHERE movies.id = 41;
```

如果不使用数据库，我们可能需要提出一个设计，其中每个类的对象都有与它关联的对象的“直接引用”，而不是构建上面的查询计划。一个Moviegoer对象将会保

代码文件：ch_advanced_rails/code/model/association1.rb

```
1  # it would be nice if we could do this:
2  inception = Movie.find_by_title('Inception')
3  alice,bob = Moviegoer.find(alice_id, bob_id)
4  # alice likes Inception, bob hates it
5  alice_review = Review.new(:potatoes => 5)
6  bob_review   = Review.new(:potatoes => 2)
7  # a movie has many reviews:
8  inception.reviews = [alice_review, bob_review]
9  inception.save!
10 # a moviegoer has many reviews:
11 alice.reviews << alice_review
12 bob.reviews << bob_review
13 # can we find out who wrote each review?
14 inception.reviews.map { |r| r.moviegoer.name } # => ['alice','bob']
```

图 5.13：一个简单直白的关联的实现允许直接引用相关的对象，尽管它们存在不同的数据库表中

持一个引用它所写的Review的数组；一个Review对象将会保持一个对它的Moviegoer的引用；等等。这样的一个设计允许写出类似于图5.13中的代码。

Rails的**ActiveRecord::Associations**[13] 模块支持这个设计，我们将通过实践来学习。将图5.14中的代码变化直接应用在标题上，然后就能够启动`rails console`，并且成功地执行图5.13中的例子。

这是如何工作的呢？因为Ruby中的所有事情都是一个方法调用，所以我们知道图5.13中第8行实际上就是调用**Movie** 对象上的实例方法**reviews=**。这个实例方法在内存里记住了它被分配的值（一个Alice和Bob的影评的数组）。回想一下，因为一个Review“属于”关联的一侧（Review属于一个Movie），为了把评论和电影关联起来必须设置评论的`movie_id`字段。**我们并不必修改`movies` 表。**所以在这个简单的例子中，**inception.save!**实际上并没有为 **Inception** 更新它的电影记录：它只是设置了Alice和Bob的评论的`movie_id`字段，将它们“连接”到 **Inception** 。当然了，如果已经修改了 **Inception** 的属性，那么**inception.save!** 就会尝试保存它们；但是因为**save!**是事务处理性的——也就是说，它要么全有要么全无——如果**save!**失败，那么它的 **每一个** 步骤都会失败，所以无论是 **Inception** 的更改还是它关联的Review的更改都不会被保存。

图5.15列出了一些添加到**movie**对象里的最有用的方法，通过声明这个对象**has_many** 评论。特别有趣的是，因为**has_many**暗示了一个有多个对象的（Review）的 **聚合类型** ，所以**reviews**方法和聚合类型类似。也就是说，对它使用图3.7中的聚合习语——用**each**来遍历它的所有元素，使用功能性习语像**sort**、**search**和**map**，等等，就像图5.13中的第8、11和14行那样。

has_one和**has_many**密切相关，它也单数化了关联方法的名称并且在单个已有对象上操作，而不是在聚合类型上。

那么`review.rb`中的**belongs_to**方法调用又会怎样呢？你可能会猜测，**belongs_to :movie**给了**Review**对象一个**movie**实例方法，它查找并返回这个评论属于的那部电影。因为一个评论至多属于一部电影，所以方法名称是单数而不是复数，并且返回单个对象而不是一个枚举。

代码文件：ch_advanced_rails/code/associations_migration.rb

```
# Run 'rails generate migration create_reviews' and then
#   edit db/migrate/*_create_reviews.rb to look like this:
class CreateReviews < ActiveRecord::Migration
  def up
    create_table 'reviews' do |t|
      t.integer    'potatoes'
      t.text       'comments'
      t.references 'moviegoer'
      t.references 'movie'
    end
  end
  def down ; drop_table 'reviews' ; end
end
```

代码文件：ch_advanced_rails/code/model/review.rb

```
class Review < ActiveRecord::Base
  belongs_to :movie
  belongs_to :moviegoer
  attr_protected :moviegoer_id # see text
end
```

代码文件：ch_advanced_rails/code/model/movie_with_associations.rb

```
# place a copy of the following line anywhere inside the Movie class
#  AND inside the Moviegoer class (idiomatically, it should go right
#  after 'class Movie' or 'class Moviegoer'):
  has_many :reviews
```

图 5.14: 上图：创建并应用这个迁移来创建Reviews表。按照惯例优先配置原则，新的模型的外键和已存在的movies和moviegoers 表是相关的。中图：把这个新的Review模型放到app/models/review.rb里。下图：对已存在的文件movie.rb和moviegoer.rb都做这个单行的更改

m.reviews	返回一个所有已有的评论的枚举
m.reviews=[r1,r2]	用**r1**和**r2**集合来替换已有评论的集合，适当的添加或者删除，通过立即在数据库里设置**r1**和**r2**的movie_id字段为**m.id**（**m**的主键）
m.reviews<<r1	通过设置**r1**的movie_id字段为**m.id**来向**m**的评论集合里添加**r1**。变化被立即写到数据库里（你不需要做一个单独的**save**）
r = m.reviews.build(:potatoes=>5)	**r**为一个未保存的 **Review**对象，它的movie_id之前就被设定为表示它属于**m**。参数和给**Review.new**的一样
r = m.reviews.create(:potatoes=>5)	像**build**一样但是立即保存对象（类似于**new**和**save**之间的区别）
注意：如果父对象**m**从未被保存，也就是说，**m.new_record?**是真，然后子对象直到父对象保存之后才保存	
m = r.movie	返回与这个评论关联的**Movie**实例
r.movie = m	设置**m**为与评论**r**关联的电影

图 5.15: 由**movie has_many :reviews**和**review belongs_to :movie**创建的关联方法的一个子集，假设**m**是一个已经存在的Movie 对象，以及**r1**、**r2**都是Review对象。查看**ActiveRecord::Associationsdocumentation**[15]获取一个完整的列表。关联方法的方法名称，遵循基于关联模型的名称的惯例优先原则

小结：

- 关联是应用程序实体中一对一，一对多，或者多对多的关系。
- 关系型数据库（RDBMS）使用外键来表示这些关系。
- ActiveRecord的关联模块使用了Ruby元编程创建新的方法来"遍历"关联，通过构建合适的数据库查询。你还必须自己用迁移来添加必要的外键字段。

■ **详细阐述：ActiveRecord中的关联和数据映射器**

关联的概念是一个软件架构层面的问题；使用外键来代表它们是一个实现方面的选择，而Rails的ActiveRecord库实现的Active Record架构格局，通过在使用关系型数据库时提供自动遍历的方法可以桥接这两者。但是基于外键的关联可以变得很复杂，以至于管理它们的开销限制了关系型数据库的可扩展性，这已经是图2.7中三层体系结构中的第一个瓶颈。一种避免这种缺陷的方式就是使用数据映射器架构模式（见图5.16），其中一个映射器类定义了每个模型和它的关联在存储系统中是如何表述的，并且提供了它自己的代码来遍历它们。因为DataMapper不依赖于底层存储系统中的外键支持，所以它可以使用所谓的"NoSQL" 存储系统，例如Cassandra[16]、MongoDB[17]和CouchDB[18]，它们省略了除了最简单的外键支持之外的支持，以达到远超于大多数的RDBMS的超水平的可扩展性。实际上，你可以在Google AppEngine[19]上部署基于Ruby的SaaS应用，如果它们使用了Google提供的DataMapper库，而不是ActiveRecord的话。

自我检查5.3.1. 在图5.14（a）中，为什么只在`reviews`表中添加外键（**references**），而不在`moviegoers`或者`movies`表中添加呢？

◇ 因为需要把很多评论与单个的电影或者观众关联在一起，所以外键必须是关联中"被拥有的"一侧模型的一部分，在这里就是Review。

自我检查5.3.2. 图5.15中，关联访问器和setter（例如**m.reviews**和**r.movie**）是实例方法还是类方法呢？

◇ 实例方法，因为评论的一个聚合类型和一部特定的电影关联，而不是和一般的电影。

5.4 间接关联

回顾图5.11，Moviegoers和Reviews之间有直接的关联，Movies和Reviews之间也是。但是因为任何给定的Reiew和Moviegoer以及Movie都关联，所以可以说Moviegoer和Movie之间 **间接** 关联。例如，我们可能会问"Alice评论过的所有电影都是哪些？"或者"哪些观众已经评论过 **Inception**？"实际上，图5.13 中的第14行实际上回答了第二个问题。

这种类型的间接关联如此常见，以至于Rails和其他的架构提供了一个描述来简化它的使用。有时也称作 **间接关联**，因为Moviegoers和Movies是 **通过** 它们的评论关联的，反之亦然。图5.17给出了如何对Rails的**has_many**使用**:through**选项，来表述这种间接的关联。你同样可以添加**has_many :moviegoers, :through=>:reviews**到**Movie**模型，然后通过**movie.moviegoers**来询问哪个观众是和一个给定的电影关联的（写评论）。

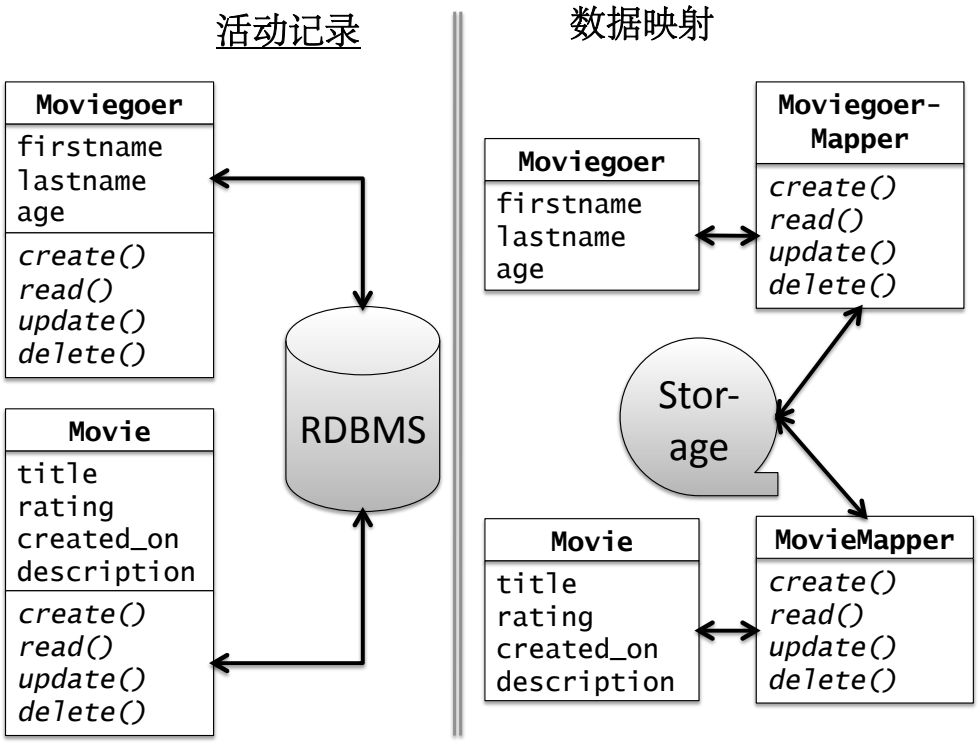


图 5.16: 在Rails使用的Active Record设计模式里（左边），模型对象本身知道它是如何存储在持久层的，并且它和其他类型的模型的关系在那里是如何描述的。在Google AppEngine、PHP和Sinatra使用的Data Mapper模式里（右边），一个单独的类把模型对象和底层存储层孤立开来。每种方式都有它的优缺点。这个 类图表 是Unified Modeling Language (UML)表的一种形式，我们将会在第11章里学到更多相关内容

代码文件：ch_advanced_rails/code/model/has_many_through_example.rb

```
1  # in moviegoer.rb:
2  class Moviegoer
3    has_many :reviews
4    has_many :movies, :through => :reviews
5    # ...other moviegoer model code
6  end
7  alice = Moviegoer.find_by_name('Alice')
8  alice_movies = alice.movies
9  # MAY work, but a bad idea - see caption:
10 alice.movies << Movie.find_by_name('Inception') # Don't do this!
```

图 5.17: 在Rails里使用间接关联。像之前一样，第8行返回的alice.movies对象表现的像一个聚合类型。但是，注意，因为Movie和Moviegoer之间的关联是 通过 属于二者的Review发生的，所以第9行和第10行的语法会导致创建一个Review对象来“连接”这个关联，并且默认情况下它的所有属性都将是nil。几乎可以肯定者并不是你想要的，并且如果验证Review对象（例如，土豆点数目必须为整数），那么新创建的Review对象就会验证失败，并且导致整个操作取消

间接关联在数据库里是如何“遍历”的呢？再看一下图5.12，找出所有Alice评论过的电影首先需要形成 **3** 个表（`movies`、`reviews`、`moviegoers`）的笛卡尔积，在我们的例子里概念上是有27行9列的。从这个表里我们随后选出电影的ID和评论的`movie_id`匹配 **以及** 观众的ID和评论的`moviegoer_id`匹配的这些行。把5.3节的解释扩展一下，SQL查询可能就是这样的：

代码文件：ch_advanced_rails/code/movies_reviews_moviegoers.sql

```
SELECT movies.*
  FROM movies JOIN reviews ON movies.id = reviews.movie_id
  JOIN moviegoers ON moviegoers.id = reviews.moviegoer_id
  WHERE moviegoers.id = 1;
```

为了提高效率，中间的笛卡尔积表通常并没有实体化，也就是说，并没有被数据库显式地构建出来。实际上，Rails 3中有一个复杂的关系代数引擎，为遍历关联构建并执行优化的SQL查询。

这两节的意义并不仅仅是解释如何使用关联，还帮助你理解如何通过使用动态类型和元编程让这些关联变得可能。在图5.14（c）中你把**has_many :reviews**添加到了**Movie**类中。**has_many**方法通过执行某种元编程来定义我们在图5.13中使用的新的实例方法**reviews=**。你肯定已经猜到，惯例优先配置原则决定了新方法的名称以及它将会在数据库中使用的表，等等。就像**attr_accessor**一样，**has_many**并不是一个声明，而是一个在运行时做了这个工作全部的实际方法调用，它添加了大量的新的模型实例方法来帮助管理这个关联。

关联是Rails特性最丰富的方面之一，所以让我们仔细看看这整个文档[20]。特别是：

- 就像ActiveRecord的生命周期挂钩一样，关联提供了额外的钩子，当从一个关联里添加或者移除掉对象时（例如当电影的新Review添加时）就会触发它，这与Movie或者Review本身的生命周期挂钩是不同的。

- 可以在关联的模型上声明验证，例如，为了保证评论仅仅在它已经和某部电影关联之后再保存：

  代码文件：ch_advanced_rails/code/model/validating_associations.rb

  ```
  class Review < ActiveRecord::Base
    # review is valid only if it's associated with a movie:
    validates :movie_id, :presence => true
    # can ALSO require that the referenced movie itself be valid
    #  in order for the review to be valid:
    validates_associated :movie
  end
  ```

- 因为在使用关联的对象上调用**save**或者**save!**也会影响到关联的对象，所以各种各样的警告适用于任何保存失败出现的状况。例如，如果你刚刚创建一个新的Movie以及两个新的Review来链接到它，并且现在你试着保存这个Movie，如果对象无效的话（可能也有其他原因），这三个保存中的任何一个都可能失败。

- 关联方法控制的额外选项，可以设置当一个“拥有的”对象被破坏时，“被拥有的对象”会发生什么。例如，**has_many :reviews,:dependent=>:destroy**指出，如果一部电影被删除了，那么属于这部电影的评论应该从数据库里删除掉。

间接关联小结：

- 当两个模型A和B和共有的第三个模型C有has-one或者has-many关系时，A和B之间就可以通过C建立起多对多的关联。
- **has_many**的**:through**选项允许你操作间接关联的任何一方，就像它是一个直接关联一样。但是，如果你直接修改了间接关联，那么中间模型对象必须是自动创建的，这可能并不是你想要的。

■ **详细阐述：Has and belongs to many**

鉴于`has_many :through`在两个外部实体之间创建了"多对多"的关联（我们运行例子里是Movie和Review），那么我们可以直接创建这样的多对多的关系，而不需要经过一个"中间"表么？ActiveRecord为纯多对多关联，也就是你不需要保持这个关系的任何信息，除了它存在这个事实，提供了之前没有讨论过的另外一种关联，**has_and_belongs_to_many**（HABTM）。例如，在Facebook上，一个给定的用户可能"喜欢"很多留言板，而一个给定的留言板可能被很多用户"喜欢"；因此"喜欢" 是用户和留言板之间的一个多对多的关系。但是，即使在简单的例子里，记录**当** 某个人喜欢或者不喜欢一个留言板时，"喜欢"的概念可能需要它自己的模型来记录这些额外的属性。因此，在大多数情况下，**has_many :through**更合适，因为它允许自己的关系（我们的例子里是电影评论）作为一个独立的模型表述出来。在Rails里，HABTM关联是通过 **连接表** 表示出来的，按照惯例这个表没有主键并且由一个特殊的迁移语法来创建。

自我检查5.4.1. 用自然语言描述一下，找出所有已经评论了某个给定`id`（主键）的电影的观众所需的步骤。

◇ 找出所有`movie_id`字段包含该电影的`id`的评论。对每个评论，找出`id`和评论的`moviegoer_id`字段匹配的观众。

5.5 关联的RESTful路由

我们应该如何用RESTful风格指定与电影评论相关联的行为呢？尤其是，至少在创建或者更新一个评论时，我们需要一种方式来把它链接到一个观众和一部电影上。可以假定观众是我们在5.2节里设置的**@current_user**。但是电影呢？

让我们从BDD的观点来思考这个问题。因为当你在头脑里有一部电影时，创建一个评论才有意义，所以最有可能的"Create Review" 功能会通过一个按钮或者一部特定电影的Show Movie Details页面上的链接来访问到。因此，在我们部署这个控件时，我们知道这个评论将要和什么电影关联。问题是如何为**ReviewsController**中的**new**或者**create**方法获取这个信息。

我们可能使用的一种方法是，当用户访问电影的Detail页面时，我们可以使用在请求期间一直持续的**session[]**，来记住这部刚刚变为"当前的电影"的ID。当**ReviewsController#new**被调用时，我们就会从**session[]**中检索这个ID，然后通过填充评论里的一个隐藏字段把它和评论关联起来，这反过来对**ReviewsController#create**也是可用的。但是，这种方式并不是RESTful 的，因为电影的ID——创建评论的信息的关键部分——在会话中是"隐藏的"。

一个更加RESTful的替代，即让电影的ID变成显式的，就是让RESTful路由本身反映Movie中Review的逻辑"嵌套"：

代码文件：ch_advanced_rails/code/nested_routes.rb

```
# in routes.rb, change the line 'resources :movies' to:
resources :movies do
  resources :reviews
end
```

因为**Movie**是关联的“拥有”一方，所以它是嵌套的外层资源。正如原始的**resources :movies**为电影上的CRUD行为提供了一组RESTful URI助手，这个 **嵌套资源** 为 **属于电影的评论** 上的CRUD行为提供了一组RESTful URI助手。对`routes.rb`做出如上的更改，并且尝试`rake routes`，将该输出与第4章中介绍的简单路由的输出作比较。图5.18 总结了新的路由，它们被 **额外** 提供给我们一直沿用的Movie上的基本RESTful路由。注意按照惯例优先配置原则，URI 通配符**:id**和资源本身的ID——也就是说，评论的ID——相匹配，并且Rails选择“外部”资源名称来让**:movie_id**捕获到“自己拥有的”资源的ID。因此ID值在控制器行为，例如**params[:id]**（评论）和**params[:movie_id]**（将要和评论关联的电影）里是可用的。

助手方法	RESTful路由和行为	
movie_reviews_path(m)	`GET /movies/:movie_id/reviews`	index
movie_review_path(m)	`POST /movies/:movie_id/reviews`	create
new_movie_review_path(m)	`GET /movies/:movie_id/reviews/new`	new
edit_movie_review_path(m,r)	`GET /movies/:movie_id/reviews/:id/edit`	edit
movie_review_path(m,r)	`GET /movies/:movie_id/reviews/:id`	show
movie_review_path(m,r)	`PUT /movies/:movie_id/reviews/:id`	update
movie_review_path(m,r)	`DELETE /movies/:movie_id/reviews/:id`	destroy

图 5.18：指定`routes.rb`里的嵌套路由也提供了嵌套URI助手，类似于提供给常规资源的更简单的路由。把这个表和第4章中的图4.7做比较

图5.19给出了使用这样的嵌套路由来创建和关联新评论的视图和行为的一个简化例子。特别需要注意的是，**ReviewsController**中使用前过滤器来保证在评论被创建之前就设置了**@current_user**（也就是说，某个人已经登录了并且将要“拥有”这个新的评论），并且从路由（参见图5.18）里捕获的电影**params[:movie_id]**存在于数据库内。如果任何一个条件不满足，那么用户就会被定向到一个正确地包含解释发生了什么的错误信息的页面。如果两个条件都满足，那么控制器实例变量**@current_user**和**@movie**就可以被控制器行为和视图访问。

视图使用**@movie**变量来为使用**movie_review_path**助手的表单创建一个提交路径（参见图5.18）。当表单提交之后，`movie_id`再次从路由解析，并且在调用**create**行为之前被前过滤器检查。最后，因为在图5.14（b）中我们声明**moviegoer_id**为一个受保护的属性，所以它不能通过**params**（**create**行为的第23行）中的集中赋值被赋给新的评论，所以我们通过从评论的其他“所有者”**@current_user**建立评论，从而设置它。

我们可以使用类似于以下的电影的Details页面上的方法，通过链接到页面来创建一个新的评论：

代码文件：ch_advanced_rails/code/view/link_to_new.html.haml

```
= link_to 'Add Review', new_movie_review_path(@movie)
```

代码文件：ch_advanced_rails/code/ctrl/reviews_controller.rb

```
class ReviewsController < ApplicationController
  before_filter :has_moviegoer_and_movie, :only => [:new, :create]
  protected
  def has_moviegoer_and_movie
    unless @current_user
      flash[:warning] = 'You must be logged in to create a review.'
      redirect_to login_path
    end
    unless (@movie = Movie.find_by_id(params[:movie_id]))
      flash[:warning] = 'Review must be for an existing movie.'
      redirect_to movies_path
    end
  end
  public
  def new
    @review = @movie.reviews.build
  end
  def create
    # since moviegoer_id is a protected attribute that won't get
    # assigned by the mass-assignment from params[:review], we set it
    # by using the << method on the association.  We could also
    # set it manually with review.moviegoer = @current_user.
    @current_user.reviews << @movie.reviews.build(params[:review])
    redirect_to movie_path(@movie)
  end
end
```

代码文件：ch_advanced_rails/code/view/new_review.html.haml

```
%h1 New Review for #{@movie.title}

= form_tag movie_reviews_path(@movie) do
  %label How many potatoes:
  = select_tag 'review[potatoes]', options_for_select(1..5)
  = submit_tag 'Create Review'
```

图 5.19: 上图：操作**Movie**和**Moviegoer**都“所有”的**Review**的控制器，使用前过滤器来保证“自身的”资源在路由**URI** 中正确识别。下图：一个可能的创建新的评论的**Haml**视图模板，也就是，`app/views/reviews/new.html.haml`

代码文件：ch_advanced_rails/code/scopes_antipattern.rb

```
# BAD: details of computing review goodness is exposed to controller
class MoviesController < ApplicationController
  def movies_with_good_reviews
    @movies = Movie.joins(:reviews).group(:movie_id).
      having('AVG(reviews.potatoes) > 3')
  end
  def movies_for_kids
    @movies = Movie.where('rating in ?', %w(G PG))
  end
end
```

图 5.20: 决定什么使电影成为"良好的"或者一部电影是否适合孩子才真正应该是Movie模型的工作，但是在这个不好的例子里，这些细节已经被根植到两个不同的控制器方法里（我们使用ActiveRecord的group方法来按照电影ID给评论分组，然后运用SQL的AVERAGE聚合器，只保留这些评论平均超过3个土豆点的电影ID）

控制器和视图对关联的支持小结：

- 为关联创建路由的一种RESTful方式是捕获"嵌套"路由URI中资源本身和它关联的项目的ID。
- 当操作有父类的"被拥有的"资源，例如Movie"所有的"的Review，前过滤器可以用来捕获和验证嵌套在RESTful嵌套路由里的ID的有效性。

■ 详细阐述：SOA、RESTful关联路由和会话

RESTful风格的SOA设计指南建议每一个请求都是独立的，所以没有一个会话的概念（也不需要）。在我们的例子里，我们使用嵌套的RESTful资源路由来把电影和评论的ID放到一起，并且依赖认证框架来设置**@current_user** 为拥有评论的观众。对一个纯SOA API来说，我们需要捕获观众ID **以及** 和电影ID一起的评论ID。Rails 的路由子系统足够灵活，允许我们用多个通配符组件来定义路由。一般来说，无论什么时候你需要创建一个有多个"所有者"的对象时，这个设计问题都会出现。如果不需要所有的拥有对象来使已有的对象有效——例如，如果可以让Review变为"匿名的"——那么另一个解决方案就是单独创建评论，并把它分配给一个观众来完成不同RESTful行为。

自我检查5.5.1. 为什么我们必须提供评论的**movie_id**和**moviegoer_id**值给**ReviewsController**中的**new**和**create**行为，而不是给**edit**和**update**行为？

◇ 一旦评论被创建，那么它的**movie_id**和**moviegoer_id**字段存储的值就告诉我们关联的电影和观众。

5.6 组合可重用作用域的查询

我们已经说过很多次，在模型-视图-控制器中要保持独立的关注，应用程序的模型实现细节不应该暴露给控制器，如图5.20所示。一个简单的解决办法就是创建类方法**Movie.with_good_reviews**（可能会带一个参数来指定"好的"评论的平均阈值）和**Movie.for_kids**，但是如果你想允许用户通过 **这两种** 属性来过滤，该怎么办呢——例如只显示适合孩子的好评电影？

可组合作用域是ActiveRelation的一个强大的特性（相当于ActiveRecord后台的"关系代数"），能帮助你做这件事。如图5.21所示，命名作用域是一个在运行时得到

代码文件：ch_advanced_rails/code/scopes_example.rb

```
1  # BETTER: move filter logic into Movie class using composable scopes
2  class Movie < ActiveRecord::Base
3    scope :with_good_reviews, lambda { |threshold|
4      Movie.joins(:reviews).group(:movie_id).
5        having(['AVG(reviews.potatoes) > ?', threshold])
6    }
7    scope :for_kids, lambda {
8      Movie.where('rating in ?', %w(G PG))
9    }
10 end
11 # in the controller, a single method can now dispatch:
12 class MoviesController < ApplicationController
13   def movies_with_filters
14     @movies = Movie.with_good_reviews(params[:threshold])
15     @movies = @movies.for_kids          if params[:for_kids]
16     @movies = @movies.with_many_fans    if params[:with_many_fans]
17     @movies = @movies.recently_reviewed if params[:recently_reviewed]
18   end
19   # or even DRYer:
20   def movies_with_filters_2
21     @movies = Movie.with_good_reviews(params[:threshold])
22     %w(for_kids with_many_fans recently_reviewed).each do |filter|
23       @movies = @movies.send(filter) if params[filter]
24     end
25   end
26 end
27 # in the view:
28 - @movies.each do |movie|
29   -# ...code to display the movie here...
```

图 5.21: 我们使用作用域来封装各种各样的过滤标准，它们可选择带一个或者多个参数。在运行时间内作用域可以灵活的组合（第14～17行），例如，在应对复选框,例如for_kids、with_many_fans等等的出现时。另一种实现方式movies_with_filters_2使用元编程以更少的代码完成了相同的事，并且很容易延伸到更多的作用域上

计算的lambda表达式。但是作用域有两个简洁的功能，使它们在定义显式的方法例如**Movie.with_good_reviews**时会表现得更好。第一，它们是 **可组合的**：如图5.21中的第15~17行所示，调用一个作用域的返回值本身是一个**ActiveRelation** 对象，其他的作用域可以应用在它上面。这允许模型逻辑块，例如这些将要彻底地重用到不同地方的过滤器。

第二，作用域是 **延迟计算的**：作用域链接建立了一个能够创建执行相应SQL查询的关系，但是这个执行直到第一次匹配的结果被请求时才发生。在我们的例子里，这发生在视图里，在图5.21 中第28~29行中的**each**循环里。延迟计算是函数式编程里的一项强大的技术，我们在第12章里会再次碰到。

> **作用域小结：**
> 作用域让你通过声明以组合的方式指定模型逻辑，使彻底地重用各种各样的模型逻辑块成为可能。因为作用域是延迟计算的——直到第一次匹配的结果被请求了才会发生数据库查询——它们可以以任何的顺序组合，而不引起性能损失。

自我检查5.6.1. 为最近n天里被评论的电影写一个作用域表达式，其中n是作用域的一个参数。

◇

代码文件：ch_advanced_rails/code/scopes_recently_reviewed.rb

```
1 class Movie < ActiveRecord::Base
2   scope :recently_reviewed, lambda { |n|
3     Movie.joins(:reviews).where(['reviews.created_at >= ?', n.days.ago]).uniq
4   }
5 end
```

自我检查5.6.2. 为什么作用域逻辑必须是块的一部分或者是lambda表达式？例如，为什么Rails的设计者不是用这个语法：

代码文件：ch_advanced_rails/code/scopes_no_lambda.rb

```
1 class Movie < ActiveRecord::Base
2   scope :for_kids, Movie.where('rating in ?', %w(G PG))
3 end
```

◇ 这个语法里，**where**语句会被立即计算（当这个代码文件加载时），而不是在每次查询前计算。换句话说，只有这些 **文件被加载时**（也就是说，当应用程序启动时）已存在的电影才会被包含在查询里。

5.7 谬论和陷阱

陷阱：太多的过滤器或者模型生命周期回调，或者在过滤器或回调里存在使用过度复杂的逻辑。

过滤器和回调提供了简便又良好定义的方式来消除代码的重复性，但是它们中的很多会让读懂应用程序的流程变得困难。例如，当有很多的前过滤器，后过滤器和“包围”过滤器触发不同集合的控制器行为时，很难分析出为什么控制器行为没有按照预期执行，或者哪个过滤器“阻止了显示”。如果过滤器中的某些操作不是在控制器里被定义的，而是在它继承的控制器里定义的，事情可能变得更糟。例

如**ApplicationController**，过滤器和回调应该在你真正的想集中代码时使用，这些代码如果不集中就会重复。

陷阱：当保存关联的时候不检查错误。

保存一个有关联的对象意味着潜在地修改了多个表。如果这些修改中的任何一个失败的话，也许因为对象或者关联的对象上的验证，那么保存的其他部分也可能会失败。确保检查了**save**的返回值，或者使用**save!**并弥补任何的异常情况。

陷阱：超过1级的嵌套资源。

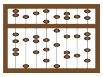
尽管从技术上来说可能会有多层嵌套资源，但是路由和行为会迅速地变得笨重，这可能意味者你的设计没有合理构造。例如，可能有一个额外的需要建模的实体关系，它使用快捷方式，如**has_many :through**来表述最终的关联。

5.8 结束语：语言、生产力和美观

本章给出了两个例子，它们使用语言特性来支持高效地创建漂亮简洁的代码。第一个是使用元编程、闭包和高阶函数来允许模型验证和控制器过滤器在单个地方简洁的声明，但是在代码中的多个点被调用。验证和过滤器是 **面向方面编程**（AOP）的一个例子。AOP因为混淆了控制流所以有时会被批评，但是如果我们能有限制地使用它，就能够增强代码的简洁性。

AOP 一直被拿来和 **来自** 编程语言构造的虚拟做比较，它开始是对Edsger Dijkstra的信 **Go To Statement Considered Harmful**（Dijkstra 1968）的一个幽默回复，这封信促进了结构化编程。

第二个例子是反映在关联助手方法里的设计选择。例如，你可能已经注意到了当与评论关联的**Movie**对象的外键字段被称作`movie_id`时，关联助手方法允许我们引用**review.movie**，允许代码关注到Movie和Review之间的 **架构上** 的关联，而不是外键名称的 **实现细节** 。你当然可以直接操作数据库里的**movie_id**或者**review_id**字段，因为基于不太强大的框架的Web应用程序经常被迫这样做，或者在Rails应用程序里这样做，就像在**review.movie_id=some_movie.id**里一样。但是除了难以阅读，这个代码还固定了一个假设，也就是外键字段以`movie_id` 命名，这可能是不正确的，如果模型使用了高级的Rails特性，例如多态关联，或者如果ActiveRecord被配置为与遵循不同命名原则的遗留数据库互相操作。在这些情况下，**review.movie**和**review.movie=**仍然会工作，但是引用**review.movie_id**会失败。因为某一天 **你的** 代码将会变成遗留的代码，所以你要帮助你的继任者变得高效——使你的实体的逻辑结构与数据库表述尽可能地分离开。



我们可能会问：现在已知道关联是如何存储在关系数据库里的，那么当我们在添加一个评论到电影之后保存这个电影，为什么**movie.save**实际上也会引起`reviews`表的改变呢？实际上，调用新的评论对象上的**save**也能工作，但是我们已经说了Movie有很多的Review，只有当更新了Movie有的Review之后再想保存这个Movie才会有意义。换句话说，它就是以这种方式设计的，以使得对编程者有意义，以及让代码变得更加漂亮。

所有的一切，验证、过滤器和关联助手方法都是值得学习的，它们是利用编程语言特性来增强代码美感和效率的成功例子。

5.9 深入学习

- Rails的ActiveRelation部分，操作了ActiveRecord关联并生成了SQL查询，它是为Rails 3完全重新设计的，非常强大。这个指南[21]除了本章介绍的之外还有很

多例子，将会帮助你有效的使用数据库，我们将会在第12章里看到它对成功操作与否是至关重要的。

- Guides section of the Rails website[22]包含了各种各样的Rails话题和指南，包括调试，管理应用程序的配置，以及更多的东西。
- Rails网站的Guides章节有一个简洁的关联基础知识的回顾[23]。
- **Rails 3 Way** （Fernandez 2010）是对Rails 3的所有方面的一个百科全书式的引用，包括非常强大的支持关联的机制。

E. Dijkstra. Go to statement considered harmful. **Communications of the ACM** , 11(3): 147–148, March 1968. URL `https://dl.acm.org/purchase.cfm?id=362947&CFID=100260848&CFTOKEN=27241581`.

O. Fernandez. **Rails 3 Way, The (2nd Edition) (Addison-Wesley Professional Ruby Series)** . Addison-Wesley Professional, 2010. ISBN 0321601661. URL `http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321601661`.

注释

[1] http://api.rubyonrails.org/classes/ActionDispatch/Routing.html
[2] http://api.rubyonrails.org
[3] http://api.rubyonrails.org/classes/ActiveModel/Validations/ClassMethods.html#method-i-validates
[4] http://api.rubyonrails.org/classes/ActiveModel/Validations/ClassMethods.html#method-i-validates
[5] http://api.rubyonrails.org/classes/ActiveModel/Errors.html
[6] http://guides.rubyonrails.org/active_record_validations_callbacks.html
[7] http://guides.rubyonrails.org/active_record_validations_callbacks.html
[8] http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html
[9] http://en.wikipedia.org/wiki/Iframe#Frames
[10] http://www.omniauth.org
[11] http://api.rubyonrails.org/classes/ActiveModel/MassAssignmentSecurity/ClassMethods.html#method-i-attr_accessible
[12] http://www.omniauth.org
[13] http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html
[14] http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html
[15] http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html
[16] http://cassandra.apache.org
[17] http://mongodb.org
[18] http://couchdb.org
[19] http://appspot.com
[20] http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html
[21] http://edgeguides.rubyonrails.org/active_record_querying.html
[22] http://guides.rubyonrails.org
[23] http://guides.rubyonrails.org/association_basics.html

5.10 建议的项目

过滤器和认证：

练习5.1. 扩展5.2节里的例子来允许通过Facebook Connect来认证。

练习5.2. 拓展你对练习5.1的解决方法，来允许一个通过身份验证的用户“连接”两个账户。也就是说，如果Alice已经用Twitter登录了，随后用Facebook登录，那么她应该能够“连接”这两个账户，使得在将来，以任何一个方式来登录都会认证一致。
提示： 考虑创建一个另外的模型**Identity**，它对**Moviegoer**有多对一的关系。

练习5.3. 在OmniAuth插件的README中，作者给出如下的例子代码，显示如何把OmniAuth集成到一个Rails应用程序里：

代码文件：ch_advanced_rails/code/ex/omniauth_example.rb

```
1  class SessionsController < ApplicationController
2    def create
3      @user = User.find_or_create_from_auth_hash(auth_hash)
4      self.current_user = @user
5      redirect_to '/'
6    end
7    protected
8    def auth_hash
9      request.env['omniauth.auth']
10   end
11 end
```

auth_hash方法（第8～10行）有一个小任务，返回OmniAuth返回的任何内容，作为尝试认证用户的结果。那么你认为为什么作者把这个功能放在它自己的方法里，而不是直接在第3行引用**request.env['omniauth.auth']**呢？

关联和RESTful应用架构：

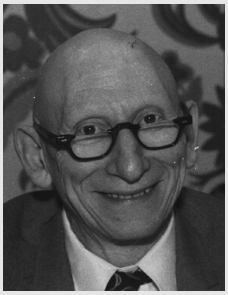
练习5.4. 用评论的**edit**和**update**方法扩展图5.19中的控制器代码。使用控制器过滤器保证用户仅可以编辑或者更新自己的评论。

6 SaaS的客户端框架：JavaScript简介

不能影响你思考方式的编程语言不值得学习。

——Alan Perlis

Alan Perlis（1922——1990）是首位图灵奖获得者（1966年），他由于在高级编程语言和编译器方面的影响而获此殊荣。1958年在他的帮助下设计的ALGOL语言，几乎影响了包括C和Java在内的每一种命令式的编程语言，。为了规避FORTRAN语言的语法和语义问题，ALGOL是第一门拥有正规文法的编程语言，与 **Backus-Naur form** 齐名（是以图灵奖获得者Jim Backus跟他的同事Peter Naur的名字命名的）



6.1 JavaScript：大图景 156
6.2 面向Ruby程序员的客户端JavaScript介绍 158
6.3 函数和构造器 163
6.4 文档对象模型和jQuery 166
6.5 事件与回调 169
6.6 AJAX：异步的JavaScript和XML 174
6.7 测试和调试JavaScript与AJAX 178
6.8 单页面应用和Json API 186
6.9 谬论和陷阱 190
6.10 结束语：JavaScript的昨天、今天以及明天 193
6.11 深入学习 195
6.12 建议的项目 197

重要概念

JavaScript 是一个应用在现代浏览器中动态的解释性脚本语言。本章介绍其主要功能，包括一些我们需要避免使用的功能，因为这些功能的设计选择很值得商榷。我们还介绍了 **JavaScript** 是如何扩展了可以作为SaaS发行的内容和应用程序的类型。

- 浏览器将网页显示成一种叫作 **文件对象模型（DOM）** 的数据结构。在浏览器中运行的JavaScript代码可以检查和修改该数据结构，从而使得浏览器显示修改后的页面元素。
- 当用户与浏览器进行交互（例如敲击按键，点击或移动鼠标），或者浏览器与服务器进行交互时，浏览器将生成一个 **事件** （event）说明发生了什么事情。当这样的事件发生时，JavaScript代码可以采取应用特定的行为来修改DOM。
- 采取 **AJAX** 或者异步的JavaScript和XML，JavaScript代码可以向Web服务器发送HTTP请求而 **不需要** 重新加载页面。响应信息可以用来修改页面元素，从而提供比传统Web页面更丰富的用户体验。Rails片段（partial）和控制器动作（controller action）可以直接用于处理AJAX交互。
- 正如我们在服务器端的SaaS代码中使用高生产率的Rails框架和RSpec TDD工具一样，在这里使用的高生产率 **jQuery** 框架和Jasmine[1] TDD工具来开发客户端代码。
- 我们遵循创建"优雅降级"（有时被称为"渐进增强"）的最佳初践：在缺少JavaScript支持的遗留浏览器中仍能为用户提供一个良好的体验，同时支持JavaScript的浏览器会提供更好的体验。

6.1 JavaScript：大图景

> JavaScript必须“看起来像Java”，而不是Java——或者成为Java的一个哑巴弟弟或劫持小孩的帮手。另外，我不得不在十天完成它，否则比JavaScript更糟糕的东西就会出现。
>
> ——Brendan Eich，JavaScript的发明者

Brendan Eich提出了把Scheme语言嵌入在浏览器中的构想（Seibel 2009）。尽管创建一个类似Java的语法的提议很盛行，许多Scheme的创意还是在JavaScript实现了。

尽管JavaScript跟Java名字相似但是其实没有什么关联：它原先的名字LiveScript是由Netscape通信公司命名的，改成JavaScript是为了利用Java的知名度。事实上，作为一种语言，JavaScript除了它表面上的语法外几乎没有继承Java的任何内容。它有来自一种Lisp的Scheme写法高阶函数功能，这种写法在AJAX编程和Jasmine TDD工具占有重要的地位。它的动态类型系统与Ruby十分相似，并且在语言使用中扮演着同样重要的角色。如果你扎实地掌握了第3章和第8章的一些概念，那么你就可以舒服地使用CSS选择器，就像你在第2章和第7章所做的那样，那么学习和使用JavaScript是一件很容易的事情。

在当今的SaaS生态系统中，JavaScript有四个主要的用途，按照从“最小JavaScript-密度”到“最大JavaScript-密度”排列如下：

(1) 使用JavaScript是为了在以服务器为中心的SaaS应用中提升用户体验，其SaaS应用遵循模型–视图–控制器架构模式。在这种情况下，JavaScript结合HTML与CSS，构成客户端SaaS编程的三个基石。这种情况被称为 **客户端JavaScript**，顾名思义，就是将JavaScript以使其与Web页非进行交互的方式“嵌入”到浏览器中。在这种情景下，服务器通常会发送预渲染的HTML块来响应JavaScript请求，而浏览器会使用这些HTML块来取代现有的页面元素。

(2) 创建适用于单一网页 **单页的应用程序**（SPA），并视情况需要通过增强与服务器的连接来进行扩展。用户体验是指当用户加载初始页面之后，那么就不会发生页面重新加载或页面重绘，虽然页面上的元素会不断更新来响应与服务器的交互。在这种情况下，服务器展现给应用程序一个或几个面向服务的架构（SOA），将XML或 **JSON**（JavaScript Object Notation）数据编码返回到客户端的JavaScript代码中，由这些代码来解析数据并根据这些数据更新各种页面元素。

(3) 创建谷歌文档之类的客户端应用程序，这可以与复杂的桌面应用程序相媲美，并且在没有互联网连接的情况下还有可能继续操作。和所有复杂的软件一样，这样的应用程序必须建立在一些底层架构上。举例来说，JavaScript的Angular[2]框架支持模型—视图—控制器模式。

(4) 建立完整的服务器端应用。这些应用与我们之前用Rails建立的十分类似，区别在于使用了像Node.js这样的JavaScript框架。

我们专注于1和2这两种用途，请把下面的最佳实践牢记于心：

- **优雅降级**：在用法1中，即使缺乏JavaScript，网站的用户体验应该也是可以接受的（显示消息“需要JavaScript才能显示此网页”的情况不算可以接受）。优雅降级一种更正面的说法是 **渐进增强**，即添加JavaScript会有更佳的体验，而不是标明缺少它就会得到惩罚。我们为什么要在乎这一点呢？原因之一就是兼容性：根据Microsoft[3]统计，24%的中国互联网用户在2013年——超过130万人——仍然使用Internet Explorer 6，它有着严重的JavaScript兼容性问题。另一

个原因是JavaScript可能出于安全原因被禁用，尤其是在企业环境中用户无法设置自己的浏览器安全配置的时候。我们把这种不支持JavaScript的情况统称为遗留浏览器。我们坚持认为我们的应用程序在缺乏JavaScript的情况下仍然可以正常使用。当然，这条准则不适用于用途2的情况，因为单页面应用（SPA）一定需要JavaScript的支持。

- **无干扰：** JavaScript代码应该保持完全独立于页面标记。这在两种情况下都助于单独地进行关注，与我们在模型——视图——控制器中所做的一样。在情景1中，它简化了支持遗留浏览器和渐进增强的实现。

图6.1比较了我们所对于服务器端和客户端编程的阐述。视频6.1.1演示了我们将在本章中添加到RottenPotatoes中的两个JavaScript功能。

	服务器	客户端
语言	Ruby	JavaScript
框架	Rails	jQuery
HTTP的服务器客户端体系结构	控制器收到请求，与模型进行交互，渲染新的页面（视图）	控制器收到请求，与模型进行交互，渲染一个片段对象，或者一个XML对象，或者一个JSON编码的对象。该对象是浏览器中运行的JavaScript代码用来更新当前页面的适当位置的。
调试	Ruby debugger，`rails` 控制台	Firebug、JavaScript 浏览器控制台
测试	带有`rspec-rails`的RSpec，用ActiveRecord模型中的fixtures和factories在数据库中进行隔离测试	Jasmine，`jasmine-jquery`，用HTML和JSON fixtures在服务器中进行隔离测试

图 6.1: 使用Ruby和Rails进行服务器端编程，与使用JavaScript进行客户端编程相似性之间的联系，使我们仍然继续关注在高效地写出无重复的（DRY）、简洁明了的并且被测试良好覆盖的代码

视频6.1.1: 带有JavaScript功能的RottenPotatoes.
光盘文件：ch_javascript/mov/js_features.mp4
首先添加一个带有过滤功能的RottenPotatoes电影列表复选框，排除那些不适宜儿童观看的电影。这种行为完全可以使用6.4节和6.5节中描述的技术，在客户端的JavaScript实现。下一步我们将给每一个电影更改“更多信息”的链接，用来在一个“浮动”的窗口中显示额外的信息，而不用加载一个新的页面。由于获取电影信息需要与服务器进行通信，所以这里我们需要用到AJAX。6.6节介绍了AJAX编程。这两种行为将用于优雅降级的实现，以使遗留浏览器的用户仍然有一个很好的体验。

---

quirksmode.org 告诉你更多有关JSAPI浏览器不兼容的内容，这会远远超出你所预期的。

JavaScript的声誉不怎么好，但这不完全是它的错。它起初允许Web浏览器运行简单的客户端代码，用来验证表单输入，或者页面上添加动画元素，或者与Java applet通信。没有经验的程序员开始复制粘贴简单JavaScript的例子，用来在他们的网页里实现好看的视觉效果，这固然带来了糟糕的编程体验，也给语言本身带来了坏名声。事实上，JavaScript是一个强大并且富于表现力的语言，它聚合了许多的伟大思想，例如支持重用和无重复的闭包和高阶函数，但没有编程经验的人很少能正确使用这些工具。

虽然这样说，但实际上因为JavaScript在动荡时期中诞生，它在句法和与语义也有很多奇怪的地方：有些可以认为是习语，但有些确实是令人遗憾的设计。这一语言中的特例几乎和正常的语法规则一样多。此外，不同版本的JavaScript解释器，以及跨浏览器的JavaScript应用程序编程接口（JSAPI，是浏览器的功能，用来允许操作

当前HTML页面的内容）都存在兼容性问题。我们将通过以下两个主要方法来避免这些问题：

(1) 限定使用所有浏览器都支持的ECMAScript 3标准的语言。

(2) 使用强大的jQuery库与HTML文档进行交互，而不是通过某个浏览器JSAPI。

jQuery 可以看作是许多种浏览器JSAPI的一个增强适配器（11.6节）。

6.2节介绍了JavaScript语言以及代码是如何嵌入到网页中，6.3节描述了其函数是如何工作的，理解它是编写干净和无干扰的JavaScript代码的基础。6.4节介绍了jQuery[4]，它用一个在所有浏览器中通用的单一API重写了某些浏览器中不兼容的JSAPI，随后的6.5节介绍了在jQuery的功能如何让编写页面元素和JavaScript代码之间的交互行为变得简单。

具有讽刺意味的是，正如我们将看到的，现代AJAX编程包含的XML比原先的要少的多。

6.6节介绍AJAX编程。在1998年，IE 5浏览器引入了一个新的机制允许JavaScript代码可以在页面加载 **之后** 与SaaS服务器进行通信，并使用服务器的信息在"原地"进行页面更新，让用户不必重新加载一个新的页面。其它浏览器很快复制了这一技术。开发者Jesse James Garrett发明[5]了 **AJAX** 一词代指异步的JavaScript和XML，并用它来描述如何把这些技术结合起来，对诸如谷歌地图这样令人印象深刻的"Web 2.0"应用程序提供支持。

测试和调试客户端的JavaScript是具有挑战性的，因为浏览器在出现错误的时候会无声的失败，而不是把JavaScript错误信息显示给没有没有怀疑的用户。幸运的是，调试器和TDD工具可以帮助你改正代码，正如6.7节中介绍的那样。

最后，6.8节描述了开发和测试基于浏览器的单页应用程序（SPA）中的两种机制，而SPA正在变得越来越流行。

JavaScript背景小结：

- JavaScript只是在名字和语法上与Java类似；尽管它有重大缺陷，它还是实现了Scheme和Ruby语言中包含的伟大思想。
- 我们专注于客户端JavaScript，也就是使用这一语言来提升以服务器为中心的SaaS应用页面的用户体验。我们力求优雅降级（在最坏的不支持JavaScript的情况下，用户体验也应该是适合使用的）和无干扰的（JavaScript代码应该是完全独立于页面标记）。

自我检查6.1.1. 对或错：一个JavaScript的早期的优势在于其表单确认（防止用户在表单中提交无效的数据），可以让我们从服务器删除砍认代码，并把这些代码移到客户端中。

◇ 错误；我们很难保证提交的请求全部来自于该页面（任何人都可以使用命令行工具构建一个HTTP请求），即使确定来自于该页面也可能出现用户使用遗留浏览器的情况。正如我们多次指出，在SaaS应用中，服务器不能相信任何人并且必须始终确认得到的输入。

6.2 面向Ruby程序员的客户端JavaScript介绍

如果你认为你已经听说过，就打断我。

——很多人都说过

除了它的名字和语法，相对于Java与JavaScript，Ruby与JavaScript反而有更多的共同点：

- 几乎所有的东西都是对象。基本的JavaScript对象除了它的键（属性名称）必须是字符串外，与Ruby中的哈希很相似。
- 变量类型都是动态的：变量没有类型，但是它们指向的对象有类型。
- 类和类型的问题甚至比在Ruby上的还要少——因为虽然我们习惯于使用编码约定来实现类的效果，但JavaScript没有类。
- 函数都是能够将上下文环境包含到其中的闭包，这使得函数在任何时刻任何地方都能够正常运行。正如匿名块（**do...end**）在Ruby中无处不在，匿名函数（**function() {...}**）在JavaScript中也无处不在。
- JavaScript解释型语言，并大量使用了元编程。

图6.2显示了JavaScript基本的语法和结构，对Java和Ruby的程序员来说，它们看起来应该很熟悉。谬论和陷阱部分介绍了几种图片相关的JavaScript陷阱。读完本章之后建议读一下这部分，否则你有可能会因为JavaScript的一个愚蠢功能带来的错误而抓狂。JavaScript的机制与Ruby所对应的机制看起来很相似，使用起来也区别不大，但还是有区别。举例来说，Ruby用**nil**同时表示"未定义"（一个从来没有被赋值的变量）和"空值"（一个总为假的值），然而在JavaScript中**null**不同于**undefined**，它代表你所得到的变量的"值"没有进行初始化。

图6.2的第一行所示，JavaScript的基本类型是对象**object**，即一个键/值对的无序集合，或者在JavaScript中被称为 **属性（properties）** 和 **槽位（slots）** 。一个属性的名称可以是任何字符串，包括空字符串。属性的值可以用任何的JavaScript表达式，可以包括另一个对象，但不能是**undefined**。

JSON.org 定义了JSON的精确语法，包含了一个可用于各种其它语言的JSON语法分析库。

如图6.3所示，JavaScript允许通过直接指定其属性和值来传递 **对象文本常量（object literals）** 。这个简单的对象文本常量是 **JSON** 的基础。虽然JSON的全称是JavaScript对象表示法，但它实际上是一个独立的数据表达方式，其表示的数据可以在SaaS服务器之间或者在SaaS客户端与服务器之间进行交换。事实上，如图6.3中第2~11行所示（第11行的结尾处的分号除外）是一个合法的JSON表示。正式地说，每一个JSON对象中的属性值可以是`Number`，统一字符编码标准的`String`，`Boolean`（其值只能是`true` 或`false`），`null`（空值），或者一个递归定义的嵌套的`Object`。不同于完整的JavaScript，在JSON中所表示的一个对象的所有字符串 **必须** 有引号，所以在图6.2中最上面一排的**title**根据JSON的语法需要一对引号。图6.4总结了一系列工具，包括可以用于检查JavaScript代码和JavaScript相关的数据结构的语法和代码规范，以及我们将在这章的剩余部分学习的JavaScript相关的数据结构和协议。

JavaScript对象允许拥有函数值的属性。精心设计的函数库利用这了个语言特性，把所有的函数和值收集到一个单一的 **命名空间** 中。例如我们将在6.4节中所看到的，jQuery定义了一个单一的全局变量**jQuery**来存取所有jQuery库中的功能，而不是把全局命名空间同许多对象杂乱地堆在库中。我们将遵循类似的做法，通过定义少量的全局变量来封装我们所有的JavaScript代码。

客户端JavaScript 一词专门指HTML页面相关并因此在浏览器中运行的JavaScript代码。在应用程序中的每一页中，使用的JavaScript函数或者变量必须包括JavaScript代码本身。我们建议通过无干扰的方式，使用一个**脚本**标记来引用包含代码的文件，如图6.5所示。生成上述标记的Rails的视图

对象	**movie={title: 'The Godfather', 'releaseInfo': {'year': 1972, rating: 'PG'}}** 在变量名合法的情况下，其属性名周围的引号是可有可无的；对象可以进行嵌套。 如果对象的属性名不合法或者直到运行之前还不知道，那么使用**movie.title**或者**movie['title']**来访问对象的属性。 **for (var in obj) {...}** 以任意顺序遍历 **obj** 的属性名。
类型	**typeof x** 返回一个字符串表示 **x** 的原始类型：其中有 **"object", "string", "array", "number", "boolean", "function","undefined"**. 所有的数字都是双倍的。
字符串& 正则表达式	**"string", 'also a string', 'joining'+'strings'** **'mad, mad world'.split(/[,]+/) == ["mad","mad","world"]** **'mad, mad world'.slice(3,4)==", mad"** ; **'mad, mad world'.slice(-3)=="rld"** **'mad'.indexOf('d')==2**, **'mad'.charAt(2)=='d'**, **'mad'.charCodeAt(4)==100** **'mad'.replace(/(\w)$/,'$1$1er')=="madder"** **/ regexp /.exec(string)** 如果没有得到匹配，就返回**null**, 如果匹配成功，则返回一个数组，该数组中的第零个元素与字符串完全匹配，并且其它的元素是括起来的捕获组。 **string .match(/ regexp /)** 也是这样的 **除非 /g** 存在正则表达式修饰符。**/ regexp /.test(string)** （更快）返回**true** 或者**false** 但是不会返回捕获组 交替构造函数：**new RegExp('[Hh]e(l+)o')**
数组	**var a = [1, {two: 2}, 'three'] ; a[1] == {two: 2}** 从零开始，动态增长；对象的键为数字（详见谬论和陷阱） **arr.sort(function (a,b) {...}) a<b,a==b,a>b** 函数返回-1, 0 or 1
数字	**+ - / %**, also **+=**, etc., **++ --**, **Math.pow(num,exp)** **Math.round(n)**, **Math.ceil(n)**, **Math.floor(n)** 分别取最近，更高，更低的整数 **Math.random()** 在（0，1）之间取随机数
类型转换	**'catch'+22=='catch22'**, **'4'+'11'=='411'** **parseInt('4oneone')==4**, **parseInt('four11')==NaN** **parseInt('0101',10)==101**, **parseInt('0101',2)==5**, **parseInt('0101')==65** （数字以**0**起始的，按八进制来解释） **parseFloat('1.1b23')==1.1**, **parseFloat('1.1e3')==1100**
布尔	**false**，**null**，**undefined** （未定义的值，有别于**null**)，0，空字符串''，并且**NaN**（非数值）是 **falsy** (布尔值为假)的，**true** 其所有它他的值是 **truthy**（布尔值为真）的。
命名	**localVar**, **local_var**, **ConstructorFunction**, **GLOBAL** 这都是约定俗成的，JavaScript没有特殊的大写规定。**var**会出现函数的范围变量，否则则为全局的（从技术上来说，是因为全局对象的属性，我们将在6.3节看到）。变量没有类，但是对象有。
控制流	**while()**，**for(;;)**，**if...else** **if...else**，**?:**（三元操作符），**switch/case**，**try/catch/throw**，**return**，**break** 语句用分号分隔开；解释器会尝试自动填充"默认"值，但这是很危险的（我们将在谬论&陷阱讲到）

图 6.2: 类似于表3.1，本这个表总结了JavaScript的基本结构。请参考正文来了解其中的重要陷阱，Ruby用nil来显示空值和返回不存在的实例变量的值，JavaScript中定义了undefined来表示通过返回未声明或者未赋值的变量的值，并区分于特殊的值null和布尔型常数false。然而，所有这三个在条件表达式中的值都为假

代码文件：ch_javascript/code/json_example.js

```
1  var potatoReview =
2  {
3    "potatoes": 5,
4    "reviewer": "armandofox",
5    "movie": {
6      "title": "Casablanca",
7      "description": "Casablanca is a classic and iconic film starring ...",
8      "rating": "PG",
9      "release_date":  "1942-11-26T07:00:00Z"
10   }
11 };
12 potatoReview['potatoes']  // => 5
13 potatoReview[movie].title    // => "Casablanca"
14 potatoReview.movie.title     // => "Casablanca"
15 potatoReview['movie'][title] // => "Casablanca"
16 potatoReview['blah']         // => undefined
```

图 6.3: JavaScript对象的表示法，即通过明确指定其属性和值来指定对象。尽管当一个对象用JSON格式编码时，所有的字符串都需要使用引号，但如果属性名是一个合法的JavaScript变量名，其引号可省略，也可以使用点标记法这种编程习语（如第13和14行所示）。由于对象可以包含其他的对象，我们可以建立嵌套的数据结构（第5行）并且遍历这种嵌套的结构（第13～15行）

Name	Tool type	Description
JSLint	基于网页的	把你的代码复制粘贴到`jslint.com`的表格中来检查错误和Doug Crockford的 **JavaScript: The Good Parts** 一书中指出的文体陷阱。还可以检查一些合法但并不安全的结构，这令有的开发者觉得太过学究了。
JavaScript Lint	命令行	一个预装在bookware VM中的叫Matthias Miller的命令行工具，根据与火狐浏览器相同的JavaScript解释器来报告错误和警告。若要运行需输入`jsl -process` **file** `.js`。
Closure	命令行	谷歌的资源到资源编译器[6]把JavaScript编译成更好地JavaScript，剔除无用的代码并且缩减它，并且给出错误和警告。它与Linter工具十分相近并且执行谷歌的JavaScript风格。它不是预装在bookware VM中，它需要计算机中有Java。
YUI	命令行	雅虎的YUI 压缩器[7]对JavaScript进行压缩，其CSS比起其它的工具更为积进，并且能够寻找运行过程中的风格问题。它不是预装在bookware VM，需要Java。
JSONlint	基于网页	这个工具jsonlint.com[8]会检查你的JSON数据结构中的语法错误。

图 6.4: 一系列可以用于调试JavaScript代码、相关数据结构以及服务器交互的工具。与C语言一样，一个挑战是，存在许多相互竞争的JavaScript的代码风格指南——谷歌的、雅虎的、Node.js项目定义的，以及其他很多——不同的工具采用并执行不同的代码风格检查

代码文件：ch_javascript/code/including_javascript.html

```
<script src="/public/javascripts/application.js"></script>
```

代码文件：ch_javascript/code/obtrusive_javascript.html

```
<html>
  <head><title>Update Address</title></head>
  <body>
    <!--- BAD: embedding scripts directly in page, esp. in body --->
    <script>
    <!--- // BAD: "hide" script body in HTML comment
          //  (modern browsers may not see script at all)
      function checkValid() {    // BAD: checkValid is global
        if !(fieldsValid(getElementById('addr'))) {
          // BAD: > and < may confuse browser's HTML parser
          alert('>>> Please fix errors & resubmit. <<<');
      }
    // BAD: "hide" end of HTML comment (l.3) in JS comment: --->
    </script>
    <!--- BAD: using HTML attributes for JS event handlers --->
    <form onsubmit="return checkValid()" id="addr" action="/update">
      <input onchange="RP.filter_adult" type="checkbox"/>
      <!--- BAD: URL using 'javascript:' --->
      <a href="javascript:back()">Go Back</a>
    </form>
  </body>
</html>
```

图 6.5: 上图：在你的HTML视图中加载JavaScript代码的无干扰且建议性的方式。下图：把JavaScript嵌入到HTML页面中的三个侵入性的方式。我们不赞成用这些方法，因为这么它会把JavaScript代码和HTML标记混合在一起。可悲的是，在实现得很差的网站上的“街头JavaScript”中，这些方法是非常常见的。通过使用script src=方法，以及我们将在本章后面的部分中使用的无干扰的技术把JavaScript代码与HTML元素连接，可以很简单地避免这些问题

助手**javascript_include_tag 'application'**，可以放置在`app/views/layouts/application.html.haml`或者是作为每个页面的一部分并由应用程序提供服务的布局模板中。如果把代码移动到一个或者多个在`app/assets/javascripts`中单独的`.js`文件，那么在你部署产品的时候Rails将自动执行以下步骤：

(1) 把JavaScript文件内容复制到此目录。

(2) 通过去除空格和执行其他简单的转换（`uglifier`工具包）来压缩最终结果。

(3) 把结果放置在公共的子目录中的一个单独的大文件中，该文件在没有Rails干预的情况下直接由表示层提供服务。

(4) 调整由URL发出的**javascript_include_tag**使用户的浏览器加载的不仅是你自己的JavaScript文件还有jQuery库。

Minifying 一词过去被称为压缩变换，从而将jQuery 1.7.2库的大小从247KB减少到了32KB。

这种自动的行为是由包括Heroku这样的现代化的生产环境所支持的，我们叫它资产管道[9]，我们在后面会看到它还允许我们使用CoffeeScript的这样的语言。我可能觉得用户加载一个巨大的JavaScript文件十分浪费，特别是当你的JavaScript应用程序只有寥寥无几的页面并且每个页面只使用JavaScript代码中很小的一部分时。但是用户的浏览器只需要加载一次大文件，保留缓存直到你改变了`.js`文件并重新部署了应用程序。此外，在开发模式中，资产管道会跳过“预编译”过程并且单独加载每一个JavaScript文件，因为在开发过程中它们很可能频繁变化。

客户端JavaScript和HTML小结：

- 像Ruby那样，JavaScript是解释型语言并支持动态类型。基本的对象类型是一个哈希，包含一个带有字符串和任意类型值（包括其它哈希）的键。
- JavaScript的基本数据类型是一个对象，这是一个属性名称和值的无序集合，类似于一个哈希。而由于可以嵌套，所以对象可以表示分层的数据结构。JavaScript的简单的对象文字标记来源于JSON数据交换格式的灵感。
- 首选的无干扰的把JavaScript与HTML页面关联在一起的方式，是通过在HTML文档中的`head`元素包含一个`src`属性是是脚本本身URL的`script`标签，使得JavaScript代码可以独立于HTML标记。Rails助手**javascript_include_tag**利用了Rails的资产管道优势来产生正确的URL。

■ **详细阐述：结构化数据应该用JSON、XML还是YAML?**

如今我们已经看到了至少三种方式来表示结构化数据：XML（参见2.3节），YAML（参见4.2节和8.5节），还有JSON（图6.3中的第2～11行）。除了这三个标准，还有 **许多其他的** 解决数据 **序列化** 的问题（又称 **编组** 或者 **紧缩** 问题）——程序内部的数据结构转换成一个被随后"复活"的表示法。反序列化（解组、膨胀）往往是由不同的程序，有可能是由不同的语言编写甚至是在网络连接的另一端，所以序列化格式必须是便携的。正如我们将在6.8节看到的，JSON已经成为SaaS客户端与服务器端之间最流行的序列化格式；事实上，Ruby 1.9添加了一个备选哈希符号**{foo: 'bar'}**来模仿JSON，实际上等价于**{:foo=>'bar'}**。

自我检查6.2.1. 在Ruby中，方法调用不带任何参数，空括号的方法调用是可选的。为什么在JavaScript中行不通?

◇ 因为函数在JavaScript中是第一类对象，一个不带括号方法名是一个值为方法本身的表达式，而不是对的方法一个调用。

6.3 函数和构造器

在第3章我们提到面向对象和类继承是不同的语言设计理念，很多人把它们混为一谈，是因为最近流行的编程语言，比如Java同时使用它们两个理念。JavaScript是面向对象的，但是它缺乏类的概念。尽管如此，有一些机制使得它与在外观和行为上与那些带有类的语言十分相似。不幸的是，这些有问题的机制设计让新的程序员在面对JavaScript时处处碰壁，特别是关键字**this**的行为。我们将重点关注**this**的两个常见的用途。本节将介绍其第一个用途，以及一个关联陷阱。在第6.5章中介绍其第二个用途。

在Firebug[10]浏览器内建的JavaScript控制台中试试这些例子。事实上并不存在类似于Ruby的`irb`交互式的JavaScript解释器。

在图6.6的1～8行中显示了一个叫**Movie**的函数。这个定义函数的语法可能我们比较陌生，在第9～11行中的替代语法我们就很熟悉了。尽管如此，我们将使用第一个语法的原因有两个。首先，与Ruby不同的是，JavaScript的函数是一个真实的 **第一类对象** ——你可以让它们在各处传递，将它们赋值给变量等等。在第一行中的语法清楚地表明**Movie**仅仅是一个变量，其值恰好是一个函数。其次，虽然不是很明显，第9行的**Movie**变量是在JavaScript的全局命名空间中声明的——这使得很难显得漂亮。一般来说，我们希望在全局命名空间下尽量减少杂乱的东西，所以我们通

代码文件：ch_javascript/code/prototype_example.js

```
1  var Movie = function(title,year,rating) {
2    this.title = title;
3    this.year = year;
4    this.rating = rating;
5    this.full_title = function() { // "instance method"
6      return(this.title + ' (' + this.year + ')');
7    };
8  };
9  function Movie(title,year,rating) {  // this syntax may look familiar...
10   // ...
11 }
12 // using 'new' makes Movie the new objects' prototype:
13 pianist = new Movie('The Pianist', 2002, 'R');
14 pianist.full_title;   // => function() {...}
15 pianist.full_title(); // => "The Pianist (2002)"
16 // BAD: without 'new', 'this' is bound to global object in Movie call!!
17 juno = Movie('Juno', 2007, 'PG-13'); // DON'T DO THIS!!
18 juno;                // undefined
19 juno.title;          // error: 'undefined' has no properties
20 juno.full_title();   // error: 'undefined' has no properties
```

图 6.6：因为函数是第一类对象，因此对象的属性值可以是函数，就像full_title那样。我们将广泛利用这一特性。请注意14～18行中的陷阱

常在自己的应用程序中创建一个的或者一些全局变量命名的对象（我们在最后将调用**RP**），然后所有的JavaScript函数都将是那个对象的属性值。

如果用JavaScript的**new**关键字（第13行）调用**Movie**函数，函数体中的局部变量**this**将被限制到一个新的对象（哈希），这个对象最终由函数返回，这与Ruby的**initialize**构造函数中的**self**很相似。在这种情况下，返回的对象会有**title**、**year**、**rating**和**full_title**属性，并且最后一个属性的值是一个函数。如果第14行对于你来说看起来像一个函数调用，那么说明你已经陷在Ruby中很长一段时间了。因为在JavaScript中的函数是第一类对象，此行只是返回**full_title**的值，它只是函数本身而不是调用它的结果！如果真的要调用的话就要像第15行那样使用括号。当调用函数时，在**full_title**的内部，**this**将指向属性是该函数的对象，在本例中是**pianist**。

需要注意的是，尽管这些例子看起来很像Ruby中调用类的构造函数和调用实例方法，但是JavaScript没有类或者实例方法的概念。事实上，没有一个特定的办法使一个JavaScript函数变成一个构造器。相反，当调用函数时使用**new**能使其变成一个构造器，导致它创建并且返回一个新的对象。这么做的原因是因为JavaScript的 **原型继承** 机制，在这里我们不进行讨论（你可以去看详细阐述了解更多）。然而，忘记这种微妙的差别，会使你在期望使用像类一样的行为而不能得到时感到困惑。

然而，JavaScript的一个功能不足在这种情况下会影响我们。就像第17行所描述的，它（十分不幸地）用一个 **不用 new**关键字并把**Movie**当作旧的普通函数来调用是完全合法的。如果你这样做，JavaScript的行为将会以两种非常可怕的方式展现。首先，在**Movie**函数体中，**this**指向的不是一个全新的对象而是 **全局对象**，该对象定义了各种特殊的常量如**Infinity**、**NaN**、**null**并且提供JavaScript环境的各种其他部分。当JavaScript在浏览器中运行时，该全局对象正好表示浏览器宣窗口的数据结构。因此，第2～5行将会创建新的全局对象并设置新的属性——这显然不是我们想要的。但不幸的是，**this**必须限定在一个范围内否则它会是未定义的，它指向了全局对象，这是语言中的严重设计缺陷。（如果你想了解关于这个古怪行为的更多的原因，详见谬论和陷阱，学习更多章节。关于这个行为的讨论超出了语言介绍的范围。）其次，由于**Movie**不能明确地返回任何东西，它的返回值

（即**juno**的值）将是**undefined**。Ruby的函数在默认的情况下，会返回在函数体中最后一个值。而JavaScript函数除非有明确的返回语言，否则它会返回**undefined**。（第6行的**return**属于**full_title**函数而不是**Movie**本身。）因此，第19和20行抛出了错误：因为我们在一个甚至不是对象的东西上试图引用一个属性（**title**）。

你可以通过严格执行被广泛使用的JavaScript约定避免这些陷阱，当且仅当一个函数是构造函数并且期望使用**new**来调用时，其函数名称应该大写。函数不是“类似构造函数”的时候应该使用小写字母开头的名字。

函数和构造函数小结：

- JavaScript的函数是第一类对象：它们可以被分配给变量，传递给其他函数，或者从函数中返回。
- 尽管JavaScript没有类，在JavaScript中以有序的方式管理命名空间意味着像对象属性那样存储函数，允许一个单一的对象（哈希）像类一样收一组相关的函数。
- 如果关键字**new**用于调用函数，**this**在函数体将指的是一个新的对象，它的原型匹配其函数自身的原型。这种机制类似于创建一个类的新实例，尽管JavaScript缺乏类。
- 如果函数 不用 关键字**new**进行调用，在函数体中的**this**将指的是全局对象，它是你永远也不想得到的，并且函数会返回**undefined**。为了避免这个陷阱，当且仅当一个函数是“类似构造函数”，并且期望使用**new**来调用时，它的名称要大写。

■ **详细阐述：原型继承**

每一个JavaScript对象继承都来自同一个原型对象——新的字符串继承自**String.prototype**，新的数组继承自**Array.prototype**等等，直到**Object**（空对象）。如果查找一个对象的属性但这个对象不具有该属性，其原型会被检查，然后是其原型的原型，直到其中一个原型响应属性或者返回**undefined**。有了这个背景知识，使用关键字**new**调用函数的效果就是创建一个与该函数的原型相同的新的对象。
原型来自于Self，一个最初在著名的Xerox PARC中设计出来的，并且对 **NewtonScript** 有着很深的影响的语言。 **NewtonScript** 是命运多舛的“掌上电脑”Apple Newton的编程语言。恰当地使用原型继承能提供一种有效的实现重用，有别于类所能提供的。不幸的是，JavaScript中原型继承的实现是三心二意的，并且它采用的语法也十分混乱，或许为了使其更像带有类继承这样的“经典”语言。Doug Crockford给这个情况提供一种很好的解决方案，可以在 《**JavaScript语言精粹**》 （Crockford 2008）中看到。

自我检查6.3.1. 参考下面的JavaScript代码，求**square.area**与**square.area()**的值会有什么区别.

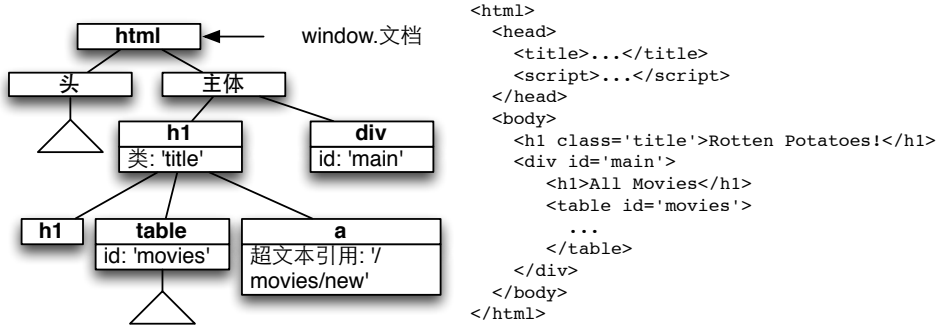


图 6.7: 与RottenPotatoes中相对应的DOM树的简单视图。一个空心的三角形表示我们为了简洁而省略了余下的子树部分。当一个页面被加载时，this.document会设置指向DOM树的根

代码文件：ch_javascript/code/selfcheck_funcall.js

```
var square = {
  side: 3,
  area: function() {
    return this.side*this.side;
  }
};
```

◇ **square.area()**是一个函数调用并且在这种情况下会返回9，而**square.area**是一个未调用的函数对象。

自我检查6.3.2. 给定自我检查6.3.1中的代码，解释一下为什么**s=new square**这句是错误的。

◇ **square**只是一个对象而不是一个函数，所以它根本不能像构造函数那样调用。

DOM在技术上是指标准本身，但是开发人员通常用它来表示特定的DOM树对应的当前页面。

6.4 文档对象模型和jQuery

万维网联盟的文档对象模型（W3C DOM）[11]是一个“平台与语言无关的接口，允许程序与脚本动态访问和更新文件的内容、结构和文档样式”——换句话说，就是一个包含了层次结构的元素的HTML、XML或者XHTML文档的标准表示，如图6.7所示。DOM元素是递归定义的，它的属性之一是一个子元素数组。因此一个DOM节点代表`<html>`HTML页面的元素就足以代表整个页面，因为一个成型页面的每个元素是一个`<html>`的后代。其他的DOM元素属性对应的HTML元素的属性（**href**、**src**等等）。当浏览器加载一个页面，HTML页面解析成一个DOM树，这与图6.7很相似。

在documentation[12]网站中`jquery-rails`工具包解释了在Rails 3.1版本之前怎样把jQuery手动添加到应用程序中。

JavaScript是如何访问DOM的？当JavaScript嵌入到浏览器中时，由全局变量**window**命名的全局对象定义了额外的浏览器特定属性和功能，统称为JSAPI。每当加载一个新的页面时，会创建一个新的全局**window**对象，它不与其字可见页面的全局对象共享数据。**window.document**是全局对象中的一个属性，也是当前文档中DOM树的根元素，还定义了一些函数用来查询、遍历和更新DOM。其中最常见的是**getElementById**，你可能在阅读别人的代码的时候遇到过。

然而，为了避免不同的浏览器在实现JSAPI时的兼容性问题，我们完全绕开这些本地的JSAPI函数，在它们周围使用jQuery中强大的“封装器”。jQuery还在本地JSAPI之外增加了额外的功能，并且其行为不在本地的JSAPI中。例如增加了动画，以及CSS和AJAX更好的支持（参见6.6节）。jQuery 定义了一个全局函

数**jQuery()**（别名**$()**），当通过CSS选择器的时候会（正如我们在图2.5看到的例子）返回当前页面中所有与那个选择器匹配的DOM元素。举例来说，如果页面中存在的话，**jQuery('#movies')**或者**$('#movies')**会返回一个ID为`movies`的单一元素。**$('h1.title')** 会返回`h1`元素中所有CSS类为`title`的元素。这项功能中更一般的一个版本是**.find(selector)**，其目标是搜寻DOM子树根。为了说明其区别，**$('p span')**搜寻 **任何的** 包含在p元素中的`span`元素，如果**elt**已经指向 **特定的** p元素，那么**elt.find('span')**只能找到**elt**后代节点的`span`元素。

不论你用**$()**还是**find**，返回值都是一个匹配选择器的节点集（一个或多个元素的集合），或者**null**代表没有可匹配的。每一个元素被"封装"在jQuery的DOM所表示的元素中，赋予它超过浏览器内置的JSAPI功能。从现在起，这样的元素就叫做"jQuery封装的"元素，来与由浏览器返回的本地JSAPI进行区分。值得一提的是，你可以在节点集中用jQuery封装的元素做各种各样的事情，如图6.8所示：

对**jQuery.noConflict()**的调用"没有定义"别名**$**，如果应用程序使用浏览器内置的**$**（通常名为**document.-getElementById**）或者加载其他的JavaScript库例如原型[13]也会试图定义**$**。

- 为了改变元素的视觉外观，定义能够创建期望的外观的CSS类，并使用jQuery从元素中添加或移除CSS类。
- 为了改变元素的内容，使用jQuery来设置元素的HTML内容或者纯文本内容。
- 为了让元素有动画效果（显示/隐藏、淡入/淡出等等），在特定的元素中调用jQuery函数来操纵DOM以达到预期的效果。

视频6.4.1显示一些关于浏览器JavaScript控制台行为的简单例子。在视频6.1.1中我们将实现这些功能。

视频6.4.1: 使用jQuery操作DOM.
光盘文件：ch_javascript/mov/jquery_dom.mp4
jQuery可以很容易地用JavaScript操作DOM并且在其中提供一个内置的有用的视觉效果库。这些简单的例子表明JavaScript不仅仅可以读取元素和页面上的内容信息，还可以通过修改元素来使浏览器对它们进行重绘。这个行为是客户端JavaScript关键所在。

最后，我们将会看到，**jQuery()**或者**$()**函数是 **重载**：其行为是基于所调用的参数的数量和类型。在本节中，我们介绍选择DOM元素的四种行为中的一种，我们将很快看到其它的行为。

DOM与jQuery小结：

- 万维网联盟的文档对象模型（W3C DOM）是构成HTML文档的层次结构元素的一个语言无关的表示。
- 所有支持JavaScript的浏览器提供JavaScript语言绑定，用以访问和遍历DOM。这组功能加上JavaScript访问浏览器的其他功能，统称为JavaScript应用程序编程接口（JSAPI）。
- 强大的jQuery库为浏览器中不同的JSAPI提供统一的适配器并且添加了许多诸如基于CSS的DOM遍历，动画化和许多其它特殊效果的增强性功能。

自我检查6.4.1. 当**this.document**出现在任何函数的作用域之外时，为什么它相当于**window.document**？

函数的属性, 例子	值/描述
$(dom-element) **$(this)**	返回一组由参数指定的jQuery包裹的DOM元素，这可以是一个CSS3选择器（例如**'span.center'** 或者**'#main'**），通过浏览器的**getElementById**函数返回元素的对象，或者一个事件处理器，元素通过**this**接收事件。就像下方的任何调用目标一样，这个函数返回值是合情合理的。
is(cond)	如果元素是**':checked'**，**':selected'**，**':enabled'**，**':disabled'**进行测试。请注意这些挑选的字符串类似于Ruby的符号，尽管JavaScript中没有符号。
addClass(), **removeClass()**, **hasClass()**	操纵`class`属性的快捷键：从元素中添加或者移除指定的CSS类（字符串），或者测试给定的类是否与当前的元素相关。
insertBefore(), **insertAfter()**	在参数前后插入目标元素。这样**newElt.insertBefore(existingElt)** 在**existingElt**之前插入**newElt**而且必须这么做。
remove()	从DOM中移除目标元素。
replaceWith(new)	用元素提供的 **new** 更换目标元素。
clone()	返回一个与目标元素完全相同的副本，递归克隆及其后代。
html(), **text()**	返回（不带有参数）或者设置（带有一个参数）的元素完成HTML内容或者纯文本的内容。如果元素里面没有嵌套其他元素，你可以用带有嵌套元素的HTML替换它，这不是必需的，但是取代它的文本将会涂抹掉其嵌套元素。
val()	返回（不带有参数）或者设置（带有一个参数）当前元素的值。对于文本框，其值是当前的字符串内容；对于按钮，是按钮的标签；对于选择菜单，是当前选定值的文本。
attr(attr,[newval]) **$('img').attr('src',** **'http://imgur.com/xyz')**	返回或者（第二个参数）设置给定属性元素的值。
hide(duration,callback), **show()**, **toggle()** **slideUp()**, **slideDown()**, **slideToggle()** **fadeOut()**, **fadeIn()**, **fadeTo(duration,target,callback)**	隐藏或者显示目标所选定的元素。可选的 **duration** 是**'fast'**, **'slow'**，或者播放动画所持续的整数或者毫秒数。可选的 **callback** 函数在动画完成时进行调用。其他的动画有着相同的参数包括**slideDown/slideUp/slideToggle** 和**fadeOut/fadeIn**。对于**fadeTo**第二个参数的目标不透明，从0.0（透明）到1.0（不透明）。
evt (func) **$('li').click(function() {** **$(this).hide();** **});**	为了事件 **evt** 把 **func** 设置成一个处理器， **evt** 上的元素是由目标选择的。 **func** 可以使匿名函数或者一个命名函数。请参见图6.9。

图 6.8: 一些在jQuery增强的DOM元素对象中定义的属性和函数。它们应当由适当的元素调用，或者由元素集合作为调用的目标（如Ruby中的接收器）。仅对一个元素有意义的函数，例如`attr`，当使用元素集合调用时仅对第一个元素起作用。函数可以同时进行读取和修改元素属性，缺少最后（或者唯一）参数的时候就像getters那样，当有参数的时候就像setters那样。除非另有说明，所有的函数会返回他们的目标，所以调用可以像elt.insertBerfore(...).hide()那样进行链接。如果想知道更多的功能请参阅jQuery文档[15]

◇ 在这些函数值外，**this**的值是全局对象。当JavaScript运行Web浏览器时，全局对象是window对象。

自我检查6.4.2. 对或错：即使用户关掉了她的浏览器窗口，与窗口相关的JavaScript代码依然可以访问和遍历窗口已显示的HTML文档。
◇ 错误。每个新的HTML文档获取其自身的全局对象和DOM，它们在文档窗口关闭之后会被摧毁。

6.5 事件与回调

动态的HTML是一个不太精确的术语，在过去它有时候指的是基于JavaScript的DOM操作与CSS结合所产生的效果。

到目前为止，我们的DOM操作都是通过直接键入JavaScript命令进行的。毫无疑问，DOM操作的一些有趣的行为可以通过用户操作进行触发。作为DOM的部分JSAPI，浏览器允许连接JavaScript **事件处理器** 到用户界面：当用户执行特定的UI动作，例如点击一个按钮或者移动鼠标进入或者离开特定的HTML元素，你可以委派一个JavaScript函数，它会被调用并有机会对此做出反应。这种能力使得页面行为更像桌面UI，其单独元素可视化地回应用户的交互，与静态的页面不同，其任何交互都会导致一个全新的页面被加载并显示。

任意元素事件	**click**，**dblclick**，**mousedown/mouseup**，**mouseenter/mouseleave**, **keypress** (**event.which**给出按下的键的ASCII码) **focus/blur**（元素获得/失去焦点），**focusin/focusout** (上一级增加/丢失焦点)
用户可编辑控制事件（表格，复选框，单选按钮，文本框，文本字段，菜单）	**change** 当任何的控制状态或者内容改变时触发 **select** （用户选择了文本，字符串**event.which**是被选定的文本） **submit** 当用户以任何形式试图提交表格时触发。

图 6.9: 一些由jQuery API定义的JavaScript事件。为事件设置一个带有 element .bind(' evt ', func)的处理器或者作为一个快捷方式 element . evt (func)。因此，$('h1').bind('click', function() {. . . })等价于$('h1').click(function() {. . . })。回调 func 将会被一个参数传递（可以不去理会）其值是一个jQuery Event对象来描述所触发的事件。记住on和快捷方式将会绑定回调到 所有 与选择器匹配的元素，所以确保传递的选择器是明确的，例如通过其ID识别元素

图6.9总结了由浏览器本地JSAPI和改进后的jQuery所定义的最重要的事件。虽然有些是由DOM元素中用户操作触发的，其它的则与浏览器本身的操作或者与例如表格提交这样的"伪UI"事件相关，可以通过单击"提交"按钮，按下回车键（在某些浏览器）或者其他的JavaScript回调来提交表单。为了给事件附加一个行为，仅仅需要提供一个当事件触发（fire）时会被调用JavaScript函数即可。我们把这样的函数叫做 **回调** 或者 **事件处理器** ，它被 **绑定** 在了DOM元素的事件上。尽管事件是由浏览器自动触发的，但是你也可以手动触发它们：举例来说元素**e.trigger('click')**触发了元素**e**的**click**事件处理器。正如我们将在6.7节所看到的，当你在做测试时这个功能是非常有用的：你可以模拟用户交互，并且确认为了响应UI事件在DOM上产生了正确的变化。浏览器定义了一些事件和元素的内置行为：例如，点击一个链接访问页面。如果这样的元素还有一人程序员提供的**click**处理器，处理器首先运行；如果处理器返回真值（如图6.2所示），其内置行为将被执行，但是如果处理器返回错误值，其内置行为将不会被执行。如果对于用户发起事件中元素 **没有** 处理器，在这种情况下图像是怎样的？在这种情况下，DOM树中的父元素给予机会来响应事件处理器。例如，如果你敲击一个`div`中的`img`元素并且`img`没有点击处理器，`div`将会收到点击事件。这个过程将会持续直到一些元素处理了这个事件或者它一路"冒泡"到顶层的**window**中，可能会有也可能没有一个内置响应，这取决于事件本身。

代码文件：ch_javascript/code/movie_list_filter.js

```
var MovieListFilter = {
  filter_adult: function () {
    // 'this' is *unwrapped* element that received event (checkbox)
    if ($(this).is(':checked')) {
      $('tr.adult').hide();
    } else {
      $('tr.adult').show();
    };
  },
  setup: function() {
    // construct checkbox with label
    var labelAndCheckbox =
      $('<label for="filter">Only movies suitable for children</label>' +
        '<input type="checkbox" id="filter"/>' );
    labelAndCheckbox.insertBefore('#movies');
    $('#filter').change(MovieListFilter.filter_adult);
  }
}
$(MovieListFilter.setup); // run setup function when document ready
```

图 6.10: 使用jQuery在RottenPotatoes的电影页面列表中添加一个"电影过滤器"复选框。文本会仔细地遍历例子，并且在本章剩下部分的图片中概括了这里所展示的技术。例子中我们使用jQuery DOM操作功能而不是浏览器内置的功能，因为jQuery的API在不同的浏览器中的表现比官方的W3C DOM规范更一致

我们讨论事件与事件处理器，引申出了JavaScript中关键词**this**的第三个常见的用途（在6.3节中介绍了前两种用途）。处理事件时，事件处理器函数体中jQuery会安排**this**来参考处理器附加的元素（如果事件是从其子节点"冒泡"那么它可能不是事件最初收到的那个元素）。然而，如果 不用 jQuery写程序，事件处理器中**this**的值是一个全局对象**document.window**，你就需要鉴别处理事件的数据结构（通常情况下作为最后的参数传递给处理器）。由于处理事件是一个如此常见的用法，并且当事件触发时，大部分情况下事件处理器都需要检查或操控角发该事件的元素的状态，因此jQuery被用来明确地设置**this**指向该DOM元素。

把这些碎片拼接一起，图6.10显示了客户端实现一个复选框的JavaScript，当该复选框被选中时，它会自动隐藏除了G和PG分级之外的电影。JavaScript中的总体策略可以概括为：

(1) 鉴别我们需要操作的DOM元素，并且确保有一个便捷且明确的方式使用**$()**选择DOM元素。

(2) 根据需要创造必要的JavaScript函数对元素进行操作。为了举例说明我们把它写下来了，但是正如我们将在图6.7中看到的，对于AJAX或者更复杂的函数，将会用TDD（见第8章）来开发代码。

(3) 定义一个与相应的JavaScript函数结合的设置函数，并执行任何其它必要的DOM操作。

(4) 一旦文件被加载，安排调用设置函数。

第一步，我们将修改我们现有的Rails电影列表视图，将CSS类`adult`连接到表单中分级不是G或PG的电影的所有行。我们仅仅需要做的是按后面的指示修改索引模板中的第13行（参见图4.6），允许我们编写**$('tr.adult')**来选择所需的行：

代码文件：ch_javascript/code/mark_adult_movie_rows.html.haml

```
  %tr{:class => ('adult' unless movie.rating =~ /^G|PG$/)}
```

第二步，我们提供**filter_adult**函数并且在复选框被选中或取消选中的时候安排调用。正如图6.10中的第4～8行所显示的，如果复选框被选中，那么成人电影行会隐藏起来；如果取消选中，就会显示出来。回想一下图6.8，其中**:checked**是jQuery用来检查元素状态的一个内置行为。同时也请记住jQuery选择器中像**$('tr.adult')**这样的通常会返回匹配元素的集合，并且像**hide()**这样的行为会被应用到整个集合。

为什么第5行中用**$(this)**而不是用**this**就好？因为哪个用户交互被派遣到JavaScript函数的机制是浏览器JSAPI的一部分，所以**this**的值是复选框（处理事件的元素）的 **浏览器** 表示。为了使用像**is(':checked')**这样的更强大的jQuery功能，我们不得不对本地元素调用**$**，以赋予其特殊的权力，并把本地的元素"封装"成一个jQuery元素。图6.12的第一行显示了**$**的用法。

第三步，我们提供**setup**函数，它完成两件事。

首先，它创建一个标签和一个复选框（第12～14行），使用图6.12第二行所示的**$**机制并且把他们插在`movies`表之前（第15行）。再次强调，通过创建一个jQuery元素我们能够调用**insertBefore**，它不是内置JSAPI浏览器的一部分。大多数像**insertBefore**这样的jQuery函数返回目标对象本身，所以能够支持像我们在Ruby中看到的"链式"的函数调用。

然后，设置函数将**filter_adult**函数绑定到复选框的**change**处理器中。你可能已经绑定了复选框的**click**处理器，但是**change**更加强大因为它是一个"伪UI"事件的例子：无论通过鼠标点击，还是键盘按键（对于浏览器来说是打开键盘导航，针对不能使用鼠标的残疾用户），甚至是其他的JavaScript代码改变了复选框的状态，该处理器都会被触发。表格的**submit**事件的是十分类似的：比起把**click**处理器绑定到表格提交按钮，把它绑定到提交事件是一个更好的选择，因为这种情况下用户提交表单仅需敲击回车键。

在<表格>之外<输入>?
是的——在HTML 4以及往后的版本中这样做是合法的，正如我们正在做的一样，只要你管理好自己全部输入行为。

为什么我们仅把标签跟复选框添加到Rails视图模板中？原因是我们的优雅降级的设计思想：通过使用JavaScript来创建复选框，遗留浏览器根本不会对复选框进行渲染。如果复选框是视图模板的一部分，遗留浏览器用户仍然可以看到复选框，但是点击它不会有任何反应。

为什么第16行用的是**MovieListFilter.filter_adult**？能不能用**filter_adult**？不可以，因为这将意味着**filter_adult**是一个在**setup**函数范围内可见的变量名，但事实上它并不是一个变量名——仅仅是一个对象为**MovieListFilter**的一个函数值属性，而**MovieListFilter 是** 一个（全局）变量。创建一个或一些全局对象来把函数"封装"成属性，而不是编写一大堆函数然后使得它们的名字和全局命名空间杂乱地混在一起，这是一个很好的JavaScript实践。

最后一步（第四步），安排对**setup**函数的调用。由于历史的原因，与页面相关联的JavaScript代码实体可以在整个页面完成加载并且DOM完全解析之前开始执行。这个功能对于响应那些联网速度较慢的浏览器和互联网是至关重要的。然而，我们常常希望等到页面加载完成并且整个的DOM都解析完成，否则我们可能会尝试把回调与元素绑定在一起，但事实上该元素是不存在的！第19行确实是这么做的，把**MovieListFilter.filter_adult**添加到函数列表中，一旦页面加载完毕就开始执行，就像图6.12最后一行显示的那样。你可以多次调用**$()**让设置函数运行多次，这可以让每个文件里的设置函数与该文件的函数性保持一致，就像我们在这里做的那样。

这是一个密集的例子，但它对你在很多UI改进中会用到的基本jQuary功能进行了阐述。本节图和表的例子中所介绍的技术十分值得花点时间细细品味。图6.12总结了四种不同的使用jQuery的**$**方式，现在所有的四种方式你都有所了解了。

var m = new Movie();	（图6.6，第13行）在Movie的函数体中，this将被绑定到一个新的对象，该对象将从函数中返回。因此（举例来说），你可以使用this.title来设置它的属性。新对象的原型将与该函数的原型相同。
pianist.full_title();	（图6.6，第15行）当full_title执行时，this将被绑定到“拥有”该函数的对象，在本例中是pianist。
$('#filter').change(MovieListFil-ter.filter_adult);	（图6.10，第16行）当filter_adult被调用来处理一个change事件时，this指向的将是该处理器被绑定到的元素，本例中是匹配CSS选择器#filter元素之一。

图 6.11: 在6.3节和6.5节中有三种常见的this用法。详见谬论和陷阱中使用和错用this的例子

使用$()或者jQuery()举例说明	在图6.10中的值/副作用
$(sel) $('.mov span')	返回经过CSS3 sel （第16行）选择器选择的jQuery包装的元素集合。
$(elt) $(this), $(document), $(docu-ment.getElementById('main'))	当一个元素由如getElementById这样的JSAPI调用返回，或者供应给事件处理器进行回调，用这个函数来生成jQuery包装版本的元素，这样你就可以如图6.8（第4行）那样调用操作。
$(HTML[, attribs]) $(' boldwords '), $('<img/>', { src: '/rp.gif', click: handleImgClick })	返回一个与传过来的文本对应的jQuery包装的HTML新元素，这必须包括至少一个带有尖括号的HTML标签（否则jQuery会认为你在处理一个CSS选择器，并且会之前的表行中调用它）。如果一个JavaScript对象是人微言轻 attribs 传递的，它会被用来构造元素的属性。（第13～14行）新的元素不会自动插入到文档中。图6.8显示了一些处理办法，第15行是其中一种方法。
$(func) $(function () {...});	一旦文件加载结束并且DOM也准备好可供操作，就运行提供的函数。这是$(document).ready(func)的快捷方式，而其本身就是一个与浏览器内置JSAPI中onLoad()处理器相关的jQuery包装器。（第19行）

图 6.12: 四种调用重载函数jQuery()或者$()的方式，以及每一种方式的效果。图6.10中展示了这四种方式

jQuery的DOM和事件处理器小结：

- 通过绑定JavaScript处理器或者 **回调** 特定元素的特殊事件，你可以设置或者覆盖各种HTML元素是如何对用户输入做出反应的。jQuery允许你把“物理的”用户事件（例如鼠标点击）和“逻辑的”伪事件（例如提交表单）绑定在一起。图6.9总结了部分的jQuery事件。
- 在事件处理器中，**this**被绑定到接收该事件的元素的浏览器DOM表示。我们通常传递DOM元素给jQuery来得到**$(this)**，给我们“jQuery封装”的元素以支持各种jQuery增强功能，例如**$(this).is(':checked')**。
- jQuery其中的一个高级功能是它的应用转换能力，比如一个元素集合（例如一组由单一的CSS选择器命名的元素）的**show()**和**hide()**以及一个单一的元素。
- 为了达到无重复和优雅降级，事件处理器和元素的绑定行为应该发生在设置函数中，而设置函数会在文档加载并准备好时被调用。这样一来，遗留的无JavaScript浏览器根本不会运行这个函数。传递一个函数到**$()**中会让将函数添加到设置列表中，当文档完成加载时该列表会被执行。

■ **详细阐述：自定义事件**

大多数的jQuery事件是基于浏览器认可的内置事件，但是你也可以定义你自己的事件并且用**trigger**来触发它们。举例来说，你可能在菜单上的月份和日期之外增加在一个单一的外部元素，例如`div`，然后在这个`div`中定义一个**update**事件用来检查月份和日期是不矛盾的。你可以在将检查代码隔离到一个单独的事件处理器中来进行**update**，并单独使用月份和日期菜单的**change**处理器，从其内部通过**trigger**来进行调用。这是一种自定义处理器来使JavaScript无重复的方式。

自我检查6.5.1. 解释一下为什么调用**$(document).find(selector)**相当于调用**$(selector)**。

◇ **document**是一个浏览器内置的全局对象（**window**）的属性之一，该对象指向浏览器中DOM的根的表示。用**$**把文档元素封装，使它能够访问**find**这样的jQuery函数，该函数在目标子树中定位所有与选择器匹配的元素。在本例中，目标是DOM的根，因此它可以找到整个文档中任何匹配的元素。

自我检查6.5.2. 在自我检查6.5.1中，为什么我们需要编写**$(document).find**而不是**document.find**?

◇ **document**也称为**window.document**，是**document**对象的浏览器本地表示。因为**find**是一个jQuery函数，所以我们需要“封装”**document**给予它特殊的jQuery权力。

自我检查6.5.3. 如果省略图6.10中的最后一行会发生什么？那么怎样情况下会调用**setup**函数?

◇ 浏览器会表现得像一个没有JavaScript的遗留浏览器。复选框将不会被渲染（因为这发生在**setup**函数中）。而且即使浏览器有JavaScript，当复选框被点击的时候，什么也不会发生，这是因为**setup**函数将我们的JavaScript处理器绑定到了复选框的**change**事件。

6.6 AJAX：异步的JavaScript和XML

在1998年，微软向Internet Explorer 5定义的JavaScript全局对象添加了一个新的函数。`XmlHttpRequest`（通常缩写成XHR）允许JavaScript代码发起HTTP请求到服务器而 **不需要** 加载新的页面，以及使用当前页面的服务器响应来修改DOM。这个新的函数是AJAX应用程序的关键，它允许创建一个更具交互性的UI，更酷似一个桌面应用程序，谷歌地图就是一个有力的证明。令人高兴的是，你已经知道了“AJAX on Rails”编程需要的所有材料：

(1) 创建新的或修改现有的控制器动作（参见4.4节）来处理由JavaScript代码生成的AJAX请求。无需渲染整个视图，其动作将会渲染一个局部（参见5.1节）来生成一个HTML块插入到页面中。

(2) 在JavaScript中构建RESTful URI并且使用XHR发送HTTP请求到服务器。你可能已经猜到了，jQuery在很多情况下有许多有用的快捷方式。因此我们将使用jQuery的更高层次并且更加强大的功能，而不是直接调用XHR。

(3) JavaScript定义为 **单线程** ——它只能在同一段时间处理一个任务直到该任务完成——当JavaScript等待服务器响应的时候浏览器UI将会被“冻结”。因此XHR反而会立即返回并且让你提供一个事件处理器回调（就像你在6.5节中浏览器编程时所做的）它会在服务器响应或者发生错误时触发。

(4) 当响应到达浏览器时，回调会传递响应内容。它可以使用jQuery的**replaceWith()**来彻底替换一个已经存在的元素，使用**text()**或者**html()**适当的更新元素内容，或者动画化例如使用**hide()**来隐藏或者显示元素。如图6.8所示，因为JavaScript函数是闭包的（就像Ruby块），所以一旦创建XHR调用，回调具有在XHR调用发生时所有的可见变量的访问权限，尽管它会迟一段时间并且在不同的环境中执行。

让我们一起来探讨每一步是如何为视频6.1.1中的AJAX功能工作的，在该视频中电影细节在浮动窗口中出现而不是加载一个单独的页面。步骤1需要我们定义或者创建一个新的控制器行为来处理请求。我们将仅仅用现有的**MoviesController#show**动作，所以我们不需要定义一个新的路线。这个设计决策是十分牢靠的，因为AJAX版本与原始版本执行相同的函数，即为RESTful “show”动作。我们将会修改**show**动作，使得如果它响应了AJAX请求，它渲染了如图6.13（a）描述的部分而不是整个视图。你也可以专门为AJAX定义一个单独的控制器动作，但是如果他们对于现有的动作进行重复操作的话，这可能不是DRY的。

我们的选择器是怎么知道**show**是被JavaScript代码调用还是被用户发起的一个正常HTTP请求调用？幸运的是，每个主要的JavaScript库和大多数浏览器在所有的AJAX HTTP请求中都会设置一个HTTP报头`X-Requested-With: XMLHttpRequest`。控制器实例中有表示进来的HTTP请求的**requiest**对象，而定义在其上的Rails助手的**xhr?**方法会检查此报头是否存在。图6.13（b）展示了控制器动作将会重绘一部分。

劫持 有时候我们用一个幽默的词汇来形容这样的技术。

继续步骤2，我们的JavaScript代码是怎样构造和触发XHR请求的？正如在视频中显示的，当单击上面有电影名字链接的时候，我们希望会出现浮动窗口。像6.5节所解释的，我们可以通过给元素增加一个明确的JavaScript **click**处理器来“劫持”它的内置行为。当然，为了实现优雅降级，我们应该只在JavaScript可用的情况下才劫持链接的行为。遵循相同的决策比如在6.5节中的例子，我们的**setup**函数（在图6.14中的第2~8行）把处理器与创建一个隐藏的`div`绑定在一起来显示浮动窗口。遗留浏览器不会运行那个函数，所以只会得到点击链接的默认行为。

代码文件：ch_javascript/code/ajax_movie_partial.html.haml

```
%p= movie.description

= link_to 'Edit Movie', edit_movie_path(movie)
= link_to 'Close', '', {:id => 'closeLink'}
```

代码文件：ch_javascript/code/ajax_show_action.rb

```
class MoviesController < ApplicationController
  def show
    id = params[:id] # retrieve movie ID from URI route
    @movie = Movie.find(id) # look up movie by unique ID
    render(:partial => 'movie', :object => @movie) if request.xhr?
    # will render app/views/movies/show.<extension> by default
  end
end
```

图 6.13: 上图：一个简单的将会被渲染的部分，它也将被返回给AJAX请求。我们给"关闭"链接一个单独的元素ID，这样我们就可以方便的把隐藏弹出内容的处理器和它绑定在一起。下图：渲染部分的控制器行为，通过简单的修改图4.9得到：如果该请求是一个AJAX请求，第5行执行渲染并且立刻返回。（:object选项使得@movie在该部分作为一个局部变量，其名称和该部分的名称匹配，在本例中是movie。）如果xhr?非真，控制器方法将会执行默认的渲染行为，像往常一样重绘show.html.haml视图

代码文件：ch_javascript/code/ajax_rottenpotatoes.js

```
RP = {};
RP.setup = function() {
    // add invisible 'div' to end of page:
    $('<div id="movieInfo"></div>').
        hide().
        appendTo($('body'));
    $('#movies a').click(RP.getMovieInfo);
};
RP.getMovieInfo = function() {
    $.ajax({type: 'GET',
            url: $(this).attr('href'),
            timeout: 5000,
            success: RP.showMovieInfo,
            error: function() { alert('Error!'); }
           });
    return(false);
};
RP.showMovieInfo = function(data) {
    // center a floater 1/2 as wide and 1/4 as tall as screen
    var oneFourth = Math.ceil($(window).width() / 4);
    $('#movieInfo').
        html(data).
        css({'left': oneFourth,  'width': 2*oneFourth, 'top': 250}).
        show();
    // make the Close link in the hidden element work
    $('#closeLink').click(RP.hideMovieInfo);
    return(false);  // prevent default link action
};
RP.hideMovieInfo = function() {
    $('#movieInfo').hide();
    return(false);
};
$(RP.setup);
```

图 6.14: ajax函数构造并且发送一个具有给定特征的XHR请求。type规定HTTP使用的动词，url是请求的URL或者URI，timeout规定一个等待响应的毫秒数，如果超时就会宣告失败，success规定一个函数来调用返回的数据，如果超时或者其他错误发生就会调用error的函数。ajax函数有更多的选项，特别是为了实现更鲁棒的错误处理

代码文件：ch_javascript/code/floating.css

```
#movieInfo {
  padding: 2ex;
  position: absolute;
  border: 2px double grey;
  background: wheat;
}
```

图 6.15: 将这段代码添加到app/assets/stylesheets/application.css指定的"浮动"窗口会定位其绝对坐标而不与它周围的元素相关。但正如文本所解释，直到运行时我们都不知道坐标是什么，所以当我们准备好显示浮点窗口时，我们使用jQuery来动态修改#movieInfo的CSS样式属性

$.ajax 当然，**$.ajax**仅仅是**jQuery.ajax**的一个别名。

实际的单击处理器**getMovieInfo**必须触发XHR请求并且提供一个将被返回数据调用回调函数。为了达到目的我们使用jQuery的**ajax**函数，它需要一个对象，这个对象的属性具体规定了AJAX请求的特征，如图6.14第10～15行所示。我们的例子显示，你可以在这个对象中指定的一些属性；一个十分重要但并没有展示的属性是**data**，它既可以是追加到URI的一个字符串的参数（如图2.3所示）；也可以是一个JavaScript对象，在这种情况下，对象的属性和值将被序列化为一个字符串，这样就可以追加到URI。和往常一样，这样的参数会出现在**params[]**哈希中，并可用于Rails控制器行为。

视频6.6.1使用Firebug交互式调试器以及Rails调试器逐步调试图6.14中其余的代码。XHR请求的目标URI很容易得到：因为我们所劫持的链接已经连接到了RESTful URI中以显示电影的细节，正如第11行显示我们可以查询其`href`属性。第13和14行提醒我们其函数值属性可以指定一个命名函数，如**success**，或者一个匿名函数，如**error**。为了保持这个例子简单，我们的错误行为是原始的退化的：不管发生什么样的错误，包括一个5000ms（5秒）的超时，我们仅仅弹出一个警告框。而成功时，我们指定**showMovieInfo**进行回调。

视频6.6.1: 通过AJAX进行单步交互.

光盘文件：ch_javascript/mov/javascript_with_breakpoints.mp4

AJAX调试需要结合JavaScript调试器例如Firebug和服务器端的调试器例如`ruby-debug`，我们在第4章碰到过这部分内容。要知道Firefox的"信息"视图（参见视频2.3.2）我们通过修改自身的DOM内容来显示弹出的窗口和工具提示，所以如果你打算用JavaScript控制器"测试一些东西"，当这些功能处于活动状态时，你可能会得到一些意想不到的结果。**提示：** 视频中的JavaScript代码用的是**RP**而不是**MoviePopup**来命名那些存储与这个例子有关的JavaScript函数的全局变量，除此这点不同之外，两份代码是相同的。

在图6.14中第20行和第23行有一些有趣的CSS技巧。我们的目标是"浮动"弹出的窗口，我们可以用CSS通过在图6.15中添加标记来指定它的绝对位置。但是由于不知道浏览器窗口的大小，我们不清楚其浮动窗口应该有多大，应该放置在哪里。**showMovieInfo**会自己计算尺寸和坐标，它有`div`一半宽，是浏览器窗口的四分之一高（第20行）。它使用服务器返回的数据取代HTML的`div`内容（第22行），元素在主窗口中水平居中，并且距离顶部边缘250像素（第23行），然后最终显示到目前为止一直处于隐藏状态的`div`（第24行）。

还有最后一件事要做：浮动`div`应当有一个使它消失掉的"关闭"链接，在第26行把一个非常简单的**click**处理器与它绑定在了一起。最后，**showMovieInfo**返回**false**（第27行）。这是为什么呢？因为作为点击一个链接（<a>）的结果，处理器被调用b，所以我们需要返回**false**来抑制其与该动作相关的默认行为，即打开链接。（出于同样的原因，在第31行中"关闭的"链接的**click**处理器返回**false**。）

有如此多不同的函数来调用相同的例子，它在进行调试时，追溯控制流就会变得非常艰难。虽然你可以用**console.log(string)**给浏览器的JavaScript控制器窗口输出一个简短的信息，但运行时会很容易忘记移除它们，正如在第8章的描述中，“`printf` 调试”会变得很缓慢，低效并且令人沮丧。6.7节会介绍一种用Jasmine创建测试的更好的方法。

最后，尽管它并没有在这个特殊的例子中出现，但是这个警告我们需要提及。当你在使用JavaScript在系统运行时动态地创建新的元素时，这个警告就有可能出现。我们知道就像与CSS类`myClass`相匹配的所有当前的元素的点击处理器那样，**$('.myClass').on('click', func)** 会和 **func** 绑定在一起。但是如果在初始的页面加载并且初次调用**on 之后** 使用JavaScript创建与`myClass`匹配的新元素，这些元素是没有与之绑定的处理器的，因为**on**只能与已经存在的元素进行绑定。

这个问题的一个常见解决方案是利用jQuery的机制，允许委派一个父元素处理一个后裔元素，通过**on**的多态性：**$('body').on('click','.myClass', func)**把HTML的`body`元素（它是一直存在的）与`click`事件进行绑定，但是把事件 **委派** 给任意后代让其与选择器`.myClass`进行匹配。因为每次事件处理时都会进行委托检查，所以与`.myClass`匹配的新元素在被创建时都会“自动地”将 **func** 作为它们的单击处理器。

AJAX小结：

- 要创建一个AJAX交互，弄清楚什么样的元素将获得新的行为，什么样的新元素可能需要被构造以支持交互或者显示响应，等等。
- 一个AJAX交互将通常需设计三段代码：发起请求的处理器、接受相应的回调、还有在**document.ready**函数中的代码（设置函数）来绑定处理器。在单独的命名函数中实现每个功能而不是提供匿名函数，这样更具可读性。
- 就像我们在6.5节的例子中那样，对于优雅降级，任何仅在AJAX交互中使用的HTML元素都应当在被创建，并绑定到设置函数中的回调，而不是包含到HTML页面本身中。
- 像Firebug或者谷歌的Chrome和苹果的Safari中的JavaScript控制台这样的互动调试器，以及使用**console.log()**的“`printf` 调试器”，它们都可以帮助你找到JavaScript的问题。但是找出问题的一个更好的方式是通过测试进行，请参见第6.7节。

■ **详细阐述：事件驱动编程**

操作指定的一个完成回调而不是等待完成发生，这样的编程模型称为 **事件驱动编程** 。你可能从这个简单的例子中的处理器和回调数目中总结出来，事件驱动程序被认为是比Rails应用程序这样的 **任务并行** 程序更难编写和调试的。在Rails应用的服务器中，单独的机器能有效地创建我们应用的副本来处理多个并发用户。当然在幕后，操作系统在这些任务之间进行切换，就像程序员在JavaScript所操作的：当一个用户的应用逻辑"副本"等待数据库的响应而被阻塞，举例来说，另一个用户的副本将被允许运行，并且当数据库响应到达时第一个副本得到了"回叫"信息。在这种情况下，事件驱动编程和任务并行编程是对偶的，并且出现了像WebWorkers[16]这样的标准，通过允许在不同的操作系统线程中同时运行JavaScript程序的副本，使得在JavaScript中任务能够并行。然而，JavaScript本身缺乏像Java的**同步**和线程间通信这样的并发抽象，故并发性必须由应用程序显式管理。

自我检查6.6.1. 在图6.14的第13行中，为什么编写的是**RP.showMovieInfo**而不是**RP.showMovieInfo()**？
◇ 前者是一个实际的函数，这是**ajax**所期望的一个**success**属性，而后者是该函数的**调用** 。

自我检查6.6.2. 在图6.14的第33行中，为什么编写的是**$(RP.setup)**而不是**$('RP.setup')**或者**$(RP.setup())**呢？
◇ 我们需要传递实际的函数给**$()**，而不是它的名字或者它的调用结果。

自我检查6.6.3. 接着上一题6.6.2，如果我们不小心调用**$('RP.setup')**会发生什么？结果将是一个语法错误还是一个合法的，但是不是我们想要的行为？
◇ 回顾**$()**被重载并且用字符串进行调用时，除非它包含任何的尖括号或者CSS选择器否则它会试图将该字符串解释为一个HTML标记。后者适用于这种情况，所以它会返回一个空的集合，因为没有元素是`MoviePopup`标签并且它的CSS类是`setup`。

6.7 测试和调试JavaScript与AJAX

即使是最简单的AJAX例子也有许多可移动的部件。在本节中，我们将展示怎样使用Jasmine进行测试。Jasmine是由Pivotal Labs研发的一个开源的JavaScript TDD框架。Jasmine的设计目的是模仿RSpec并且跟RSpec一样支持TDD实践。本节其余的部分需要你读过第8章或者精通TDD和RSpec。就像图6.16显示的，我们将在Jasmine中重用所有TDD概念。

为了启用Jasmine，添加**gem :jasmine**到Gemfile中并且像往常一样运行`bundle`，然后如图6.17所示，在应用程序的根目录运行命令。由于一些令人费解的原因，我们并不能完全运行一个空的Jasmine测试程序组，所以创建包含如下代码的文件`spec/javascripts/sanity_check_spec.js`：

代码文件：ch_javascript/code/sanity_check_spec.js

```
1 describe('Jasmine sanity check', function() {
2   it('works', function() {  expect(true).toBe(true); });
3 });
```

为了运行测试，输入`rake jasmine`，然后用浏览器打开`http://localhost:8888`我们就可以看到测试结果。从现在开始，当我们在`app/assets/javascripts`更改任

步骤	RSpec/Ruby	Jasmine/JavaScript
函数库	rspec, rspec-rails gems	jasmine gem, jasmine-jquery add-on
安装	rails generate rspec:install	rails generate jasmine:install
测试文件	spec/models/, spec/controllers/, spec/helpers	spec/javascripts/
命名规则	spec/models/movie_spec.rb包含app/models/movie.rb的测试	spec/javascripts/movie_popup_spec.js包含app/assets/javascripts/movie_popup.js的测试；spec/javascripts/moviePopupSpec.js包含app/assets/javascripts/moviePopup.js的测试
配置文件	.rspec	spec/javascripts/support/jasmine.yml
运行所有测试	rake spec	rake jasmine，然后访问http://localhost:8888；或者运行rake jasmine:ci，使用Selenium/Webdriver运行一次并捕获输出；或者使用jasmine-headless-webkit[17]来从命令行运行。

图 6.16: 比较Jasmine跟RSpec中的设置和使用。所有的路径都与应用的根目录相关，并且所有的命令有需要在应用程序的根目录中运行。正如你所见，它们之间的主要区别在于在Ruby用lower_snake_case对文件和方法命名，而在JavaScript使用的是lowerCamelCase

代码文件：ch_javascript/code/jasmine_jquery_setup.sh

```
1 rails generate jasmine:install
2 mkdir spec/javascripts/fixtures
3 curl https://raw.github.com/velesin/jasmine-jquery/master/lib/jasmine-jquery.
      js > spec/javascripts/helpers/jasmine-jquery.js
4 git add spec/javascripts
```

图 6.17: 在应用程序中创建与Jasmine相关的目录。在第1行中创建的spec/javascripts目录是你的测试路径，其子目录support和helper类似于RSpec设置（参见8.2节）。在第2行为夹具添加一个子目录（参见8.5节）。第3行安装了一个Jasmine附加组件，用以测试基于jQuery的代码以及使用夹具。第4行在项目添加了一些新的JavaScript TDD文件

自检?
`rake jasmine:ci`在运行Jasmine程序组的时候只会使用一次Webdriver并收集其输出，然后（快的）`jasmine-headlesswebkit`[18]或者Jasmine-Rails[19]工具在不需要运行浏览器的经常性开销的情况下运行测试。这两种方法都是在自动化持续集成（CI）的环境（参见12.3节）下运行的（这样的话就不需要人为在浏览器页面观察其测试结果。）

何的代码或`spec/javascripts`中的测试时，只需要重新加载浏览器页面以重新运行所有测试。

测试AJAX代码必须解决两个问题，如果你已经读过第8章中关于TDD的内容，你就已经对两者的解决方案足够熟悉了。第一，就像我们在8.7节所做的那样，我们必须通过拦截AJAX调用来“隔离互联网”，这样就可以返回“包装好的”AJAX响应并且在与服务器隔离的情况下测试JavaScript代码。我们将用桩解决这个问题。第二，我们的JavaScript代码希望在已渲染的页面上找到特定的元素，但是正如我们所看到的，当我们运行Jasmine测试时，浏览器会查看Jasmine报告页面而不是应用程序。幸运的是，我们可以用夹具来测试依赖于页面现有的某些DOM元素的JavaScript代码，和8.5节中测试Rails应用程序代码一样，它基于现有数据库中的某些项目。

测试用例的结构

- **it("does something", function() {...})**
  通过给一个描述性的名称和一个函数来执行测试制定了一个简单的测试（规则）。
- **describe("behaviors", function(){...})**
  收集一组相关的规则，由**it**、**beforeEach**和**afterEach**。**describe**的调用组成的函数体可以进行嵌套。
- **beforeEach** 和**afterEach**
  在拥有相同的**describe**的**it**块之前运行设置/清除函数。和RSpec一起，如果**describe**是嵌套的，所有的**beforeEach**都是由外到内运行的，并且所有的**afterEach**是由内到外的。

期望

期望是采用**expect(object). expectation** 或者**expect(object).not. expectation** 的规范。以下是Jasmine中内置的常用期望：

- **toEqual(val), toBeTruthy(), toBeFalsy()**
  用==测试等式，或者一个值为布尔真或布尔假表达式。

是由Jasmine jQuery插件提供的常用的期望——在这种情况下，参数**expect**应该是jQuery封装的元素或者元素的集合。

- **toBeSelected(), toBeChecked(), toBeDisabled(), toHaveValue(stringValue)**
  表单中输入元素的期望。
- **toBeVisible(), toBeHidden()**
  如果一个元素宽度和高度为0，或者它是一个`type="hidden"`的表单，或者它的元素或者它的祖先的CSS属性是`display: none`，那么隐藏就为真。
- **toExist(), toHaveClass(class), toHaveId(id), toHaveAttr(attrName,attrValue)**
  测试一个元素的各种属性和特征。
- **toHaveText(stringOrRegexp), toContainText(string)**
  测试元素的文本是否完全匹配给定的字符串或正则表达式，或者包含给定的子串。

图 6.18: small subset 中总结了Jasmine和Jasmine-jQuery的部分常用功能，其结构如图8.20和图8.21所示，还可以从网站Jasmine documentation[22]和Jasmine jQuery add-on[23]看到完整文档

图6.18为RSpec用户概括了Jasmine。我们将通过5条Jasmine规则的快乐路径来开发6.6节中的弹出窗口功能。虽然对愉快路径也几乎不能进行详尽的测试，但是我们的目标是用一般的例子说明Jasmine测试技术，并明确讲述Jasmine桩和夹具在AJAX测试中的使用。

如图6.20所示，Jasmine测试用例的基础架构是显而易见的：像RSpec一

桩（间谍）

- **spyOn(obj , 'func')**
  创建并返回现有函数的一个监视者（模拟），该间谍必须是由`func`命名的 **obj** 的函数值属性。监视者 **替代** 现有函数。
- **mostRecentCall**是一个可以跟踪其状态的监视者属性，包括最近调用的参数数组**args[]**在内。

监视者可以调用下面的修饰符来控制其行为：

- **andReturn(value)**
- **andThrow(exception)**
- **andCallThrough()**
- **andCallFake(func)**
  func 必须是一个没有参数的函数，尽管它已经访问了监视者通过 **spy .mostRecentCall.args[]**调用的参数，并且可以用这些参数调用其它的函数。

夹具和工厂（需要`jasmine-jquery`）

- **sandbox({class: 'myClass', id: 'myId'})**
  给定HTML属性创建一个空的`div`，那么它默认就是一个不带CSS类并且ID是`sandbox`的空`div`。这是另一种创建**setFixtures**参数的方法，它可以避免把HTML字符串带入测试代码中。
- **loadFixtures(" file .html")**
  从`spec/javascripts/fixtures/` **file** `.html`加载HTML内容并且并且把它放进一个ID为`jasmine-fixtures`的`div`中，该夹具在测试用便之间会被清除。
- **setFixtures(HTMLcontent)**
  创建一个夹具而不是从文档中加载它。 **HTMLcontent** 可以是如** text **这样的文字形式的HTML字符串，或者是一个**$(' text ')**这样的jQuery封装元素例如。
- **getJSONFixture(" file .json")**
  在`spec/javascripts/fixtures/` **file** `.json`.中返回JSON对象。这对于存储模拟数据来模拟AJAX调用的结果非常有用，而且无需把JSON对象文字放进测试代码中。

图 6.19: 图6.18续表，描述了桩（Jasmine中的 间谍 ）和夹具

代码文件：ch_javascript/code/movie_popup_spec.js

```
1  describe('MoviePopup', function() {
2    describe('setup', function() {
3      it('adds popup Div to main page', function() {
4        expect($('#movieInfo')).toExist();
5      });
6      it('hides the popup Div', function() {
7        expect($('#movieInfo')).toBeHidden();
8      });
9    });
10   describe('clicking on movie link', function() {
11     beforeEach(function() { loadFixtures('movie_row.html'); });
12     it('calls correct URL', function() {
13       spyOn($, 'ajax');
14       $('#movies a').trigger('click');
15       expect($.ajax.mostRecentCall.args[0]['url']).toEqual('/movies/1');
16     });
17     describe('when successful server call', function() {
18       beforeEach(function() {
19         var htmlResponse = readFixtures('movie_info.html');
20         spyOn($, 'ajax').andCallFake(function(ajaxArgs) {
21           ajaxArgs.success(htmlResponse, '200');
22         });
23         $('#movies a').trigger('click');
24       });
25       it('makes #movieInfo visible', function() {
26         expect($('#movieInfo')).toBeVisible();
27       });
28       it('places movie title in #movieInfo', function() {
29         expect($('#movieInfo').text()).toContain('Casablanca');
30       });
31     });
32   });
33 });
```

图 6.20: AJAX代码中开发的五条Jasmine specs的愉快路径（参见6.6节）。第2～9行测试MoviePopup.setup函数是否正确建立了用来显示电影信息的浮动div。第10～32行检查AJAX代码的行为，并通过在调用附近建立桩而不用实际调用RottenPotatoes服务器

代码文件：ch_javascript/code/movie_row.html

```
<table id="movies">
  <tbody>
    <tr class="adult">
      <td>Casablanca</td>
      <td>PG</td>
      <td><a href="/movies/1">More about Casablanca</a></td>
    </tr>
  </tbody>
</table>
```

图 6.21：这个HTML夹具模仿由RottenPotatoes的电影列表视图所产生的`#movies`列表的一行（如图4.6）。我们可以通过从浏览器的"视图资源"中复制粘贴HTML代码而生成这样的夹具，或者通过检查JavaScript控制台中的$('#movieInfo').html()来生成由JavaScript动态生成的资源（例如"隐藏成人电影"复选框）。夹具可能由于你改变了应用的视图而不同步，谬论和陷阱中描述了一种防止这种情况的方法。

样，Jasmine用**it**来指定一个单一的例子和可嵌套的**describe**块，把例子的相关集合分组。和RSpec中一样，**describe**和**it**把代码块看成一组参数，而在Ruby中代码块是用**do...end**分隔开来的，在JavaScript中，它们是没有参数的匿名函数（没有名字的函数）。标点符号序列**});**也是十分普遍的，因为**describe**和**it**是有两个参数的JavaScript函数，后一个参数正是一个没有参数的函数。

describe('setup')检查**MoviePopup.setup**函数是否正确地创建了`#movieInfo`容器但并不让它显示出来。**toExist**和**toBeHidden**是由Jasmine-jQuery的附加组件提供的期望匹配器。由于Jasmine在你运行任何例子之前会加载所有的JavaScript文件，因此在运行测试之前调用**setup**（如图6.14的第34行），这样测试函数是否各司其职是明智的。

describe('AJAX call to server')的例子十分有趣，因为它们用桩和夹具来将客户端的AJAX代码从与之通信的服务器间隔离开来。图6.19总结了在Jasmine和Jasmine-jQuery中可用的桩和夹具。就像RSpec一样，Jasmine允许我们运行测试装置并且用**beforeEach** 和**afterEach**清除代码。在这组例子中，我们的设置代码加载了图6.21所示的HTML夹具，如果它在电影列表显示出来后被调用了，就可以用来模仿**getMovieInfo**控制器将会看到的环境。夹具的功能是由Jasmine-jQuery提供的，每一个夹具是在`div#jasmine-fixtures`中被加载的，而`div#jasmine-fixtures`是在Jasmine主页的`div#jasmine_content`之内的，并且所有的夹具在每个规格之后都清除，以保持测试的独立性。

第一个例子（图6.20的第12行）检查AJAX调用是否使用了从表中得到的正确的电影URL。为了做到这一点，它用Jasmine的**spyOn**来隔离掉**$.ajax**函数。就像RSpec的**桩**，其调用会 **代替** 任何已有的同名函数，所以当我们通过点击鼠标手动触发`#movies`表中的（唯一）`a`元素时，如果一切工作正常，我们应该期待我们的间谍函数被调用了。因为在JavaScript中函数是对象属性值是很常见的，**spyOn**需要两个参数，一个对象（**$**）和用以监视**'ajax'**的对象的函数值属性名。

第15行看起来非常复杂，但其实它十分简单明了。每一个Jasmine间谍牢记传递给它的**mostRecentCall**的参数，就像在6.6节中解释的那样回调，**真正的** 调用AJAX函数接收一个单一的对象（如图6.14的第9～15行所示），其**url**属性是AJAX调用应该访问的URL，第15行的规则是仅检查这个URL的值。事实上，它测试**$(this).attr('href')**是否是从表单中提取AJAX URL的正确的JavaScript代码。

图6.22展示了为互联网建立桩来测试AJAX和测试面向服务的架构中代码的相似点。（参见8.7节）。正如你所见，在这两种场景下，在哪里建立桩是根据在我们的测试中需要多少个堆栈来决定的。

这个简明的介绍，以及本节中的汇总表，应该能够使你开始在JavaScript代码

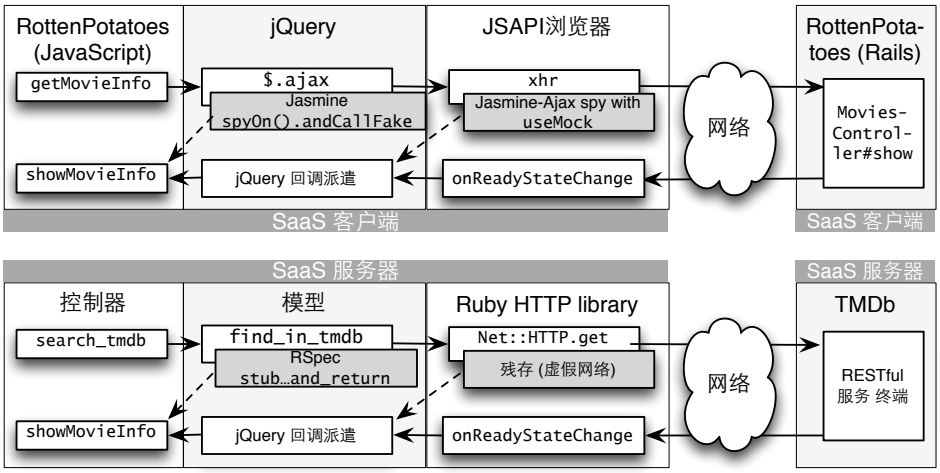


图 6.22: 上图：正常情况下，我们的**getMovieInfo**函数调用**jQuery**的**ajax**，这会在浏览器的**JSAPI**中调用**xhr**，发送请求到服务器。然后服务器的回复会触发浏览器的**JSAPI**中的回调逻辑，这会调用一个内部的**jQuery**方法，并最终调用**showMovieInfo**回调。如果我们为**ajax**函数建立桩，就会导致**showMovieInfo**立即被调用；我们也可以"更进一步的"为**xhr**建立桩（用**Jasmine-Ajax**插件），这会导致**jQuery**内部的调度器立即被调用。下图：**8.7**节关于图**8.19**的讨论的图形化表示

代码文件：ch_javascript/code/jquery_factory.js

```
1 describe('element sanitizer', function() {
2   it('removes IMG tags from evil HTML', function() {
3     setFixtures(sandbox({class: 'myTestClass'}));
4     $('.myTestClass').text("Evil HTML! <img src='http://evil.com/xss'>");
5     $('.myTestClass').sanitize();
6     expect($('.myTestClass').text()).not.toContain('<img');
7   });
8 });
```

图 6.23: Jasmine-jQuery的**sandbox**方法用给定的属性创建一个新的**HTML**`div`。如果没有给定的属性其`id`默认为`sandbox`。第**4**和**5**行用**sandbox**创建的元素。**sandbox**可以用来暂时性的容纳用类似工厂的方法创建的元素而会"污染"用**HTML**标记的测试代码

用使用BDD。这些工具的完整文档最好的来源是Jasmine documentation[24]和Jasmine jQuery add-on[25]。

JavaScript中的Jasmine BDD小结：

- 就像RSpec一样，Jasmine规则是伴随着描述性字符串的匿名函数。它们是由Jasmine函数的**it**引入的，可以用与**describe**与**beforeEach**和**afterEach**有关联的调用（建立测试和拆除测试）的（嵌套的）**describe**块来分组。
- **spyOn**可以用来通过用间谍替代的方式为现有的方法建立桩。如图6.19所示，该间谍的行为可以通过CandCallThrough、**andReturn**等函数控制。
- Jasmine-jQuery的HTML夹具不仅可以提供"之前的"用来触发AJAX请求的内容，还可以提供"之后的"用来测试AJAX请求结果是否成功的内容。

■ **详细阐述：为什么不在Jasmine规则上使用只在客户端使用的代码?**

在6.5节中我们没有包含只在客户端使用的规则的例子，由于同样的原因第8章也不写视图规则：一种普遍的做法是用集成或者验收等级测试来检测客户端视图的行为，例如使用Webdriver的Cucumber脚本（参见7.5节）。

■ **详细阐述：测试客户端表单确认**

在表单可以提交之前，一个常见的JavaScript使用实例是在用户输入的时候来确认表单中的条目。可以通过这样的方式来测试这样的自确认的表单：创建一个表示一个表单或表单一部分的HTML夹具，用 **元素** 的**.val()**来设置一个或多个表单输入的值，然后触发 **元素** 的**.blur()**来让元素失去焦点，模拟用户按下Tab键或者用鼠标来移到不同的表单字段。然后你可以检查另一个表单字段是否用新的值正常更新了（通过检查它们的 **元素 .val()**），或者侦查确认函数，用**.andCallThrough()**来确保元素失去了焦点，即它被调用了。。

■ **详细阐述：是夹具还是工厂?**

正如8.5节所解释的，在Rails应用程序中人们经常喜欢用工厂在"恰当的位置"创建必要的测试替身而不是指定夹具。为什么我们用夹具而不是工厂来描述AJAX测试？其中一个原因是，它们在JavaScript中的代价是不同的。在Rails应用程序中，夹具是在运行测试之前被加载到数据库中的，并且像**find**这样的各种各样的ActiveRecord方法在夹具不同时可能表现出不同的行为。因此夹具可能会破坏测试的独立性（**I**ndependence）。在Rails中工厂是有吸引力的代替品，因为像FactoryGirl这样的工具包可以很容易地在每一个测试需要的"恰当的时间"将测试替身实例化。在Jasmine中，我们将使用HTML"工厂"取代HTML夹具，我们将使用**$("**)来创建内嵌的HTML元素，但是许多开发者认为这是令人不快的，因为把HTML标记与JavaScript测试代码混在一起会使JavaScript代码难以阅读。如图6.23所示，Jasmine-jQuery可能在没有过度用HTML标记污染你的测试代码的情况下使用工厂提供一些简单的支持，但总体上AJAX的测试夹具避免了Rails的夹具中的一些陷阱，它们确实把自己引入了另一个陷阱中——应用程序视图"不同步"的可能。我们将在谬论和陷阱节讨论这个陷阱及其解决方案。

自我检查6.7.1. Jasmine-jQuery同样也支持**toContain**和**toContainText**检查是否文本或HTML的字符串出现在元素内。在图6.20的第7行中，为什么

用**.not.toContain(' ')**代替**toBeHidden()**会出错？

◇ 隐藏元素是不可见的，但是它仍然包含与元素相关联的文本或者HTML。因此**toContain**样式的匹配器可以被用来测试元素的 **内容** 而不能测试其 **可见性** 。此外，有许多种方法把元素隐藏起来——它的CSS可以包括`display:none`，它宽度和长度都可以为零，或者它的祖先是隐藏的——而且**toBeHidden()**会匹配检查所有上面提到的内容。

自我检查6.7.2. 跟RSpec一样，Jasmine支持**andReturn()**从桩中返回一个包装的值。在图6.20中，为什么我们必需编写**andCallFake**，当作为**ajax**的结果，来将**ajaxArgs**传递给一个函数，而不是简单地编写**andReturn(ajaxArgs)**？

◇ 要记住AJAX调用是异步的。它 **不是 $.ajax**调用会从服务器返回数据的情况：正常情况下，它会立刻返回，有时会过一段时间返回，你的回调是用从服务器返回的数据调用的。**andCallFake**模拟了这个行为。

6.8 单页面应用和Json API

谷歌地图是新兴的客户端单页应用程序（SPA）的早期例子。在一个SPA中，从服务器加载初始页面之后，所有的交互都会在不出现任何页面重新加载的情况下显示给用户。虽然在本节中我们不会开发一个完整的SPA，但是我们将展示这样的开发过程中必要的技术。

到目前为止，我们已经集中精力在使用JavaScript来提高以服务器为中心的SaaS应用。因为HTML长期以来是由这些应用程序服务的内容的 **通用语** ，渲染一部分并且在DOM内用JavaScript插入一些"已经做好"的部分是一种明智的继续下去的方式。但是在SPA中，通常客户端的代码会在服务器中请求一些"原始"数据，并且使用那些数据来构建或者修改DOM元素。Rails应用程序如何才能返回原始数据而不是HTML标记的JavaScript客户端代码？

一个简单的机制是让控制器行为使用**render :text**返回一个普通的字符串。但是如果我们需要发送结构化数据到客户端，同样会面临用关系型数据库解决过的问题——怎样才能对数据进行"冷冻——干燥"来让其结构在客户端中可以正确地"再造"，以及如何 **序列化** 和随后 **反序列化** 数据。

在SPA发展早期，XML似乎对序列化格式是一个有前途的选择。AJAX里面的X字母代表XML，3.6节展示了一个从服务器返回的XML格式数据的简单例子。尽管XML看起来简单，但是完整的XML规范有很多怪癖，使得完全兼容的解析器很编写起来十分复杂并且很有挑战性。虽然大多数主流浏览器都有内置的XML解析器，但它们的JSAPI是不兼容的，而且jQuery不给它们提供任何的façade（外观），就像它在DOM所做的那样。即使是至是轻量级的XML解析器例如Sax-JS[26]也在你的JavaScript程序中添加了大约1300行代码（LOC），并且在访问DOM时不提供任何便捷方法。

一个有吸引力的替代方案是JSON，它是我们在6.3节中看到的JavaScript对象符号。它比XML更简单但是足以代表许多应用程序的数据结构，而且十分受欢迎，因为许多RESTful的API可以服务于JSON或者XML：你可以为每种格式调用一个不同的端点（URL），或者通过在对REST API的调用中传递一个参数指定哪一个是你想要的。由于JSON格式内置了JavaScript内置对象符号的合适的一个子集，我们可以原则上只编写**var e=eval(j)**来反序列化一个JSON编码的字符串**j**到一个"活动的"JavaScript对象**e**中去。在实践中，现代化的浏览器的JSAPI包括一个**JSON.parse**函数，它不如**eval**快但是更安全：而**eval**将执行任意的（不可信甚

代码文件：ch_javascript/code/render_json.rb

```
Review.first.to_json
#  => "{\"created_at\":\"2012-10-01T20:44:42Z\", \"id\":1, \"movie_id\":1,
  \"moviegoer_id\":2,\"potatoes\":3,\"updated_at\":\"2013-07-28T18:01:35Z\"}"
```

图 6.24: Rails内置的to_json可以通过在模型的每一个属性上递归地调用自身来序列化简单的ActiveRecord对象。正如你所见，它不会遍关联信息——用户的movie_id跟moviegoer_id信息会序列化成整数，而不是变成Movie跟Moviegoer这样由整数外键指向的对象。你可以通过在ActiveRecord模型中重写to_json来进行更复杂的序列化

至有可能是邪恶的）JavaScript代码，如果你要求解析有效的JSON数据结构之外的任何东西，那么**JSON.parse**将会引发一个错误。（JSONLint，在图6.4中列出，验证JSON表达式的语法。）

为了在客户端代码中使用JSON，我们必须解决三个问题：

(1) 我们是如何让服务器应用程序生成JSON来响应AJAX请求，而不是渲染HTML视图模板或者其中的一部分？

(2) 客户端是如何指定它期望的JSON响应的，以及他是如何使用JSON响应数据来更新DOM的？

(3) 当测试期望得到JSON响应的AJAX请求时，我们怎么才能用夹具来“为服务器建立桩”并且在隔离的情况下测试这些行为，就像我们在6.7节所做的那样？

第一个问题很简单。如果你已经控制了服务器代码，Rails控制器动作可以发出JSON而不是XML或者一个用**render :json=> 对象** 的Haml模板，这会发送一个JSON所表示的对象回到客户端，就像控制器行为的单一的响应那样。就像渲染一个模板，在每一次**渲染**动作中你只能允许进行一次调用，所以一个给定的控制器行为的所有响应数据必须打包成一个JSON对象。

render :json通过在 **对象** 中调用**to_json**来创建要发回给客户端的字符串。默认的**to_json**实现可以序列化简单的ActiveRecord对象，如图6.24所示。

当然，我们必须安排服务器返回一个JSON对象，如上所述。

为了让AJAX调用期望得到一个JSON编码的响应，我们仅需要确保传递给**$.ajax**的参数对象包含一个值为`json`的**dataType**属性，如图6.25所示。这个已存在的属性告诉jQuery让它在返回的数据中自动调用**JSON.parse**，所以你就不必亲自这么做了。

如何不是每次都调用服务器，我们该如何测试我们测试代码？令人高兴的是，Jasmine-jQuery的夹具机制允许指定JSON的夹具以及HTML的夹具，如图6.26所示。

代码文件：ch_javascript/code/movie_popup_json.js

```
1  var MoviePopupJson = {
2    // 'setup' function omitted for brevity
3    getMovieInfo: function() {
4      $.ajax({type: 'GET',
5              dataType: 'json',
6              url: $(this).attr('href'),
7              success: MoviePopup.showMovieInfo
8              // 'timeout' and 'error' functions omitted for brevity
9            });
10     return(false);
11   }
12   ,showMovieInfo: function(jsonData, requestStatus, xhrObject) {
13     // center a floater 1/2 as wide and 1/4 as tall as screen
14     var oneFourth = Math.ceil($(window).width() / 4);
15     $('#movieInfo').
16       css({'left': oneFourth,  'width': 2*oneFourth, 'top': 250}).
17       append($('<p>' + jsonData.description + '</p>'),
18              $('<a id="closeLink" href="#"></a>')).
19       show();
20     // make the Close link in the hidden element work
21     $('#closeLink').click(MoviePopup.hideMovieInfo);
22     return(false);  // prevent default link action
23   }
24   // hideMovieInfo omitted for brevity
25 };
```

图 6.25: 这个版本的MoviePopup期望的是一个JSON而不是HTML响应（第5行），所以success函数用返回的JSON数据结构在弹出的div中创建新的HTML元素（第17∼19行；观察jQuery的DOM操作函数，像append这样的可以有多个想要创建的HTML片段的参数）。为了简洁起见就像图6.14里面一样做了省略

代码文件：ch_javascript/code/movie_popup_json_spec.js

```
describe('MoviePopupJson', function() {
  describe('successful AJAX call', function() {
    beforeEach(function() {
      loadFixtures('movie_row.html');
      var jsonResponse = getJSONFixture('movie_info.json');
      spyOn($, 'ajax').andCallFake(function(ajaxArgs) {
        ajaxArgs.success(jsonResponse, '200');
      });
      $('#movies a').trigger('click');
    });
    // 'it' clauses are same as in movie_popup_spec.js
  });
});
```

图 6.26: Jasmine-jQuery希望在spec/javascripts/fixtures/json中找到夹具包含.json数据的文件。执行到第5行之后，jsonResponse将会包含实际的JavaScript对象（不是原生的JSON字符串！）这将会传递给success处理器

单页程序小结：

- 由于用JavaScript增强的传统SaaS应用通常会创建完整的HTML块（例如使用HTML的部分）这样客户端会简单地将其“插入到”当前的HTML页面中。SPA通常会从一个或者多个服务器收到结构化的数据并且用那些数据来合成新的内容，或者在页面中修改已有的内容。
- JOSN的简单性和它与JavaScript完美的天然匹配很快使它成了SPA中交换的结构化数据的令人更喜欢的格式。Rails可以用**render :json=>对象**把简单的ActiveRecord模型序列化为JSON，但是你可以覆盖ActiveRecord的**to_json**方法来序列化任意复杂的数据结构。
- 在**$.ajax**调用中设置**dataType**属性为“`json`”可以告诉jQuery让它自动把服务器的响应数据反序列化为JSON对象。
- 一个返回JSON夹具的间谍可以用来模拟测试一个测试SPA时服务器的响应，这使得Jasmine测试能够被从SPA所依赖的远端服务器中隔离出来。

■ 详细阐述：其他为AJAX将互联网隔离掉的方法

8.7节讨论了在怎样在“靠近客户”或者“远离客户”的地方为互联网建立桩以使得测试和外部的服务相隔离。6.7节通过为**$.ajax**建立桩和强制它立即调用**success**函数，而不是允许它和外部HTTP请求一起处理，我们“靠近客户”的地方建立了桩。这个技巧和8.7节中用**find_in_tmdb**建立桩来立即得到一个值，而不是允许它进行一个真实的请求相类似。另一种选择，更彻底的锻炼处理实际AJAX服务器响应的代码，就是在网络层建立桩，正如FakeWeb对Rails应用程序所做的。如FakeWeb让你提供一个基于XHR调用的参数的“包装的”的XML或者HTML响应。`jasmine-ajax`[27]——Pivotal Labs提供的一个Jasmine扩展，让你可以给AJAX XHR调用时提供“包装的”XML、HTML、JSON响应，而不是允许XHR调用被处理。然后你可以监视**success, failure, timeout**等等函数，以及传递给**$.ajax**来确保依赖于服务器响应的正确的处理器被调用了。

■ 详细阐述：利用自己的API实现以用户为中心的设计

XML: **$(data).find('elt_id').children().each(function() ...)**
主持人模式和利用你自己的API来实现用户为中心的设计[28]

自我检查6.8.1. 图6.26给出了JSON夹具的使用，为什么 **仍然** 需要在第4行中加载HTML夹具？

◇ 第9行尝试为一个匹配了`#movies a`的元素而触发了单击控制器，如果我们不加载HTML夹具表示电影表单中的一排的话，就不会有这样的元素存在。（实际上，**MoviePopupJson.setup**函数试图在这个元素上绑定一个单击处理器，所以这也会失败。）这是一个例子，既使用了HTML夹具来模拟用户点击页面元素，也使用了JSON 夹具来模拟从服务器来的对那个点击的成功的响应。

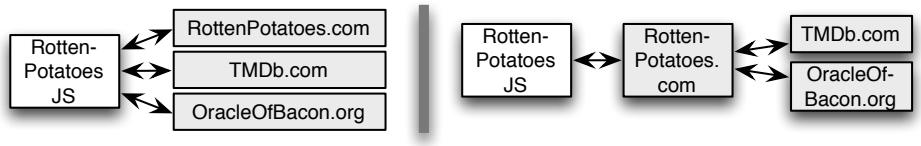


图 6.27: 从多个不同的服务里检索资产的一个内置浏览器的SPA的架构。左图：如果JavaScript代码是从RottenPotatoes.com中加载的，那么默认的浏览器为JavaScript实现的 同源策略 将会禁止代码对其他的域中的服务器调用AJAX。 跨资源共享 （CORS）规范放松了这个约束，但只有非常新的浏览器才支持。右图：在传统的SPA架构中，单个服务器服务于JavaScript代码并且和其他远程服务交互。这个安排遵守了同源策略，也允许主要的服务器在需要的情况下为客户做一些额外的工作

■ **详细阐述：同源策略**

你也可以安排SPA来和一个RESTful的服务器façade（下面）通信（参见11.6节），如图6.27所示。如果SPA依赖于多个站点的内容，那么你可能可以这样做：为了安全，JavaScript浏览器应用程序受 **同源策略** 约束，这表示一个JavaScript应用程序只能让AJAX向应用程序本身被服务的相同源（2.2节中描述到的方案、主机名称和端口号）发出请求。

6.9 谬论和陷阱

谬论：AJAX当然会提升应用程序的响应能力，因为更多的执行是在浏览器中发生的。

对于一个精心设计的应用程序，AJAX可能有很大的 **潜力** 改善某些交互的响应能力。然而，AJAX的许多因素也会阻止完成这一目标。在任何的AJAX动作发生之前，JavaScript代码必须从服务器获取，它依赖于某些库或者框架，如jQuery。在像手机平台这样的平台上，这可能会引起前期延迟使得后面的存储无效。JavaScript面对不同的浏览器和设备，其性能会有很大的变化，网络连接速度从1Mb（智能手机）到1000Mb（高速有线网络），并且其他的因素远远超越了你的控制范围，所有的这一切让AJAX对性能的影响很难进行预测。在某些清况下，AJAX会减缓程序的速度。就像所有的强大的工具一样，AJAX应该充分理解为什么以及如何让它能提高响应能力，而不是因为应用程序感觉很慢，就增加了这个工具会有所帮助的模糊的希望上。在第12章中提到的技巧将会帮助你识别和解决一些常见的性能问题。

陷阱：创建一个网站因为没有JavaScript而失败的网站，而不要用它进行改进。

为了对安全性要求较高和使用不同浏览器的人能更好地访问，一个设计良好的网站如果能用JavaScript就会更好，但是没有它也是 **可以接受的** 。举例来说，GitHub的浏览器页面代码在没有JavaScript的情况下会运行得很好，但是在有JavaScript的情况下它可以更平稳和快速地工作。用这两种方法访问的情况说明Github网站是一个渐近增强的好例子。测试在没有JavaScript的时候会运行得更快：网站的JavaScript是可选的意味着你可以在一个更快的例如像Cucumber跟Capybara这样的“无头浏览器”模式下做大部分的集成测试。

陷阱：产品代码中无声的JavaScript故障。

当在Rails代码中出现未预期的异常，你会立即知道，正如我们看到的：应用程序会显示一个丑陋的错误页面。或者如果你已经足够小心了，像Hoptoad这样的服务会立即联络你并报告错误，这部分我们将在第12章看到。但是JavaScript的问题在于它无声的故障——用户点击一个控件或加载一个页面，然后什么也没发生。这样的问题是特别有害的，因为如果一个AJAX请求发生时，**success**回调将不会被调用。所以我们提出警告：jQuery提供快捷键**$.get(url,data,callback)**、**$.post(url,data,callback)**和**$.load(url_and_selector)**，就像我们在6.6节所介绍的封装的**$.ajax()**方法，但是所有的这些东西发生错误时会产生无声的故障，而**$.ajax()**允许你在发生指定错误的情况下调用额外的回调。

陷阱：测试时无声的JavaScript故障。

当你在使用Jasmine的时候也会出现"无声的失败"陷阱：当在浏览器页面重新加载Jasmine规格时，如果JavaScript文档或者规范中有语法错误，你可能看到一个没有任何错误的空白页面。我们建议使用Doug Crockford的JSLint[29]工具，它不仅可以发现语法错误，还可以指出你的坏习惯，还有Crockford跟其他人认为不合理的JavaScript设计。同样，你可能会意外地加载HTML夹具结果出现非法的HTML。举例来说，你可能意外地创建了一个ID复制了一个已有的元素的对象，或者包含了一个不适合嵌套的元素，或有HTML语法错误。由于夹具在测试运行时会被加载到一个实际页面，不规范的页面的结果可能是不可预知的，或者会出现无声故障的情况。

陷阱：只提供昂贵的服务器操作并且依靠JavaScript做其余的事情。

如果JavaScript是如此强大，为什么不充分地用它编写所有的应用程序逻辑，而使用服务器仅仅作为单薄的API数据库？我们将在第12章中看到，成功的缩放需要**减少**数据库加载，并且经过仔细考虑后才能把API暴露给JavaScript客户端代码，编写没有必要的复杂数据库查询指令是有风险的，所以客户端JavaScript代码可以为每一个视图挑选需要的数据。第二，既然你已经几乎能够完全控制服务器的性能（还有用户体验），那么说明你的客户端几乎什么都没有。因为面对多种多样浏览器类型，网络连接速度和其他因素远远超出你的可控范围，每个用户浏览器的JavaScript性能是庞大且不可控的，使得提供一致的用户体验性能非常困难。

陷阱：允许HTML或者JavaScript夹具之间相互同步或者与应用程序代码同步。

使用HTML夹具测试你的AJAX功能的风险在于，夹具是基于应用程序生成的HTML运行的，如果改变了应用程序视图模板而没有对夹具进行修改的话，运行的测试与HTML相悖，因为此HTML与应用程序真正的输出不匹配。

有一个自动的解决方案：Pivotal Labs工作流[30]使用RSpec（参见第8章）自动地为Jasmine测试从应用程序视图中创建一些夹具。这个办法同样避免了另一个微妙的问题：在一个小的夹具上成功执行的测试，在整个页面的DOM却失败了。举例来说，两个事件处理器尝试回应同一事件可能会在生产中做错事，但是如果它们仅仅在该时间使用独立的夹具测试其中一个事件，单元测试发现不了这个问题。运行规则使用全页面的"夹具"而不是一个页面中不同片段的夹具来解决这个问题，Pivotal公司的自动工作流使用有效的方法能够解决这个问题。

陷阱：在JavaScript函数中this的错误用法。

在JavaScript函数体中**this**的值是程序员面对这个语言时悲痛困惑的源头。特别地，在你看到如下的两个例子，新的程序员不会发现某个函数的**this**值与该函数怎样编写的无关，而是与它是怎么样被调用的有关。所以同一函数的不同调用会导致绑定不一样的**this**。详细讨论为什么**this**是这样工作的已经远远超出本节的介绍范围，“学习更多”这节是给那些对此感兴趣想要了解深层次信息的人提供一些指导，这将会把你带入一个新的领域并告诉你JavaScript如何受到其祖先Scheme和Self的影响的。

直到你更深刻地理解这个问题之前，你可以通过我们描述的常见情况，将你的代码编写得更安全，这部分在图6.11中有总结。

陷阱：JavaScript——不良的部分。

> 为了指针运算[Ken] Thompson发明了一种++运算符。我们现在知道其实指针运算并不好，并且我们也再不这么做了。它已经牵连到缓冲区溢出攻击以及其他恶意的东西上。C++是最后一种包括++运算符的流行语言，一门如此糟糕的语言，连它的名字都是以此运算符命名的。
>
> ——Douglas Crockford，《**编程风格和思维观念**》，在第12届USENIX网页应用程序会议如是说。

在创业热潮中，JavaScript是在荒诞的进度压力下诞生的：LiveScript仅仅用了10天进行设计、实现和发布。结果导致该语言有一些公认的性能不足和陷阱，正如在C语言中有一些“陷阱（gotchas）”一样。所以我们强烈推荐使用例如JSLint[31]这样的高质量代码工具，它们能在你碰到潜在的陷阱时和在有机会的情况下美化你的JavaScript代码的情况下警告你。这个工具在bookware提供的虚拟机中已经预装好了，你可以通过在命令行中键入`jsl -process` **filename** 来运行它。它有无数据的禾土工大，你可以在JSLint website[32]上阅读它。

需要避免的特定陷阱如下：：

(1) 解释器在觉得你忘记插入分号的时候好心地帮你添上，但有时它会猜错，结果是致命的并且其代码行为会引起一个意想不到的改变，像下面的例子：

代码文件：ch_javascript/code/semicolon_misfeature.js

```
// good: returns new object
return {
  ok: true;
};
// bad: returns undefined, because JavaScript
//  inserts "missing semicolon" after return
return
{
  ok: true;
};
```

一个好的工作区需要采取一个一致的代码样式，能够使得“精确的错误”及时被发现，就像推荐给Node.js包开发者的一样。

(2) 尽管语法能建议出显示块的作用域——举例来说，在循环函数体内部的for循环有自己的一组花括号，在其中附加的**var**可以出现声明——在函数中的 **所有** 用**var**声明变量其整个函数的 **任何地方** 都是可见的，包括在任何的嵌套函数。因此，在像**for (var m in movieList)**这样的通用结构中，**m**的范围是整个函数，不仅仅是循环体本身。在循环体内部的带有**var**声明变量也是如此。这种行

为叫做 **函数范围**，是在Algol 60发明的。让函数保持简短（还记得9.5节部分中的SOFA么？）有助于避免阻塞陷阱以及函数范围。

(3) **Array**仅仅是一个键为非负整数的对象，在一些JavaScript实现中，通过线性阵列检索项目稍稍比通过散列检索项目要快一点，但是在大多数情况下是没有必要的。其缺陷是如果你试着用负数或者非整数索引一个阵列，将会创建一个字符串值键。即**a[2.1]**变成**a["2.1"]**。

(4) 比较运算符**==**跟**!=**会自动执行类型转换，所以**'5'==5.0**是对的。运算符**===**跟**!==**比较后没有执行任何转换。这可能会令人困惑，因为Ruby也有一个**==**操作符（"三长度"）做一些完全不同的事情。

(5) 数组和哈希相等是基于标识符而不是值本身的大小，所以**[1,2,3]==[1,2,3]**是错误的。不像Ruby，**Array**类可以定义其自己的**==**运算符，在JavaScript中，你必须解决其内置的行为，因为**==**是语言的一部分。

(6) 字符串是不可改变的，所以像**toUpperCase()**这样的方法经常返回一个新的对象。如果你想要更换现有的变量的值需编写**s=s.toUpperCase()**。

(7) 如果你调用一个函数时其参数要比它本身定义指定的要多，额外的参数会被忽略；如果你调用较少的参数，未被分配的参数是**未定义的**。在这两种状况中，数组的**arguments[]**（在函数范围内）可以访问所有实际传递的参数。

(8) 如果你在字符串中创建新的属性，则字符串常量与通过**new String**创建的字符串的行为不同，就像下面的代码所显示的。因为JavaScript在**fake**中创建了一个临时的"包装对象"来回应**fake.newprop=1**，执行分配然后立即销毁包装对象，留下没有任何**newprop**属性的一个"真正的"**fake**。如果你明确地用**new**创建它们那么你就可以设置一个额外的字符串属性。不过还好的是，你无需在内置类型中设置属性：定义你自己的原型对象并且用成分分析而不是集成（参见第11章）来创建字符串的一个的属性，然后设置你认为合适的其他属性。（此限制同样适用于数字和布尔值，正如我们先前提到的它不适用于数组，因为它们只是一个哈希的特例。）

代码文件：ch_javascript/code/wrapper.js

```
real = new String("foo");
fake = "foo";
real.newprop = 1;
real.newprop      // => 1
fake.newprop = 1; // BAD: silently fails since 'fake' isn't true object
fake.newprop      // => undefined
```

6.10 结束语：JavaScript的昨天、今天以及明天

JavaScript作为Web客户端语言的特殊地位耗费了人们大量的精力。大多数智能机和平板电脑现在可以运行JavaScript，可便携的移动设备可以使用HTML、CSS和JavaScript，可以建立丰富轻便的移动设备端应用程序，而不是分别创建单独的移动平台，如iOS和Android。像PhoneGap[33]框架使得JavaScript能够编写移动端应用程序，特别是与jQuery Mobile[34]或者Sencha Touch这样灵活的UI框架相结合。事实上，如今在移动应用程序中 **不使用** JavaScript的主要原因是其性能不

代码文件：ch_javascript/code/coffee_example.coffee

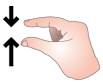
```
RP =
  setup: ->
    # construct new DOM elements
    $('<label for="filter" class="explanation">' +
      'Restrict to movies suitable for children' +
      '</label>' +
      '<input type="checkbox" id="filter"/>'
      ).insertBefore('#movies').click(RP.filter_adult)
  filter_adult: ->
    # 'this' is element that received event (checkbox)
    if $(this).is(':checked')
      $('#movies tbody tr').each(RP.hide_if_adult_row)
    else
      $('#movies tbody tr').show()
  hide_if_adult_row: ->
    $(this).hide() unless /^G|PG$/i.test($(this).find('td:nth-child(2)').text
        ())
$(RP.setup)       # when document ready, run setup code
```

图 6.28: 图6.10的CoffeeScript版本。在众多不同的语言中，CoffeeScript为匿名函数提供了类似于 Haskell语言 的简洁的语法，使用类似Haml的缩进而不是用花括号表示其结构，不但允许类似Ruby省略大多数括号而且借用@实例变量符号引用this属性。生成的代码更易阅读并且比JavaScript文字版少1/3行

足，但是由于“Web 2.0”网站和像谷歌文档（Google Docs）这样复杂的SPA越来越依赖JavaScript，开发者已经专注于JavaScript的性能和生产力。

1、**性能。**即时编译（JIT）其技术和其他高级语言工程功能一样是为语言带来压力，使用其他的解释器甚至一些编译语言来缩短了性能差异。超过半打的**JavaScript引擎** 实现和一个编译器（Google的Closure）就能够编写大多数开源的代码，例如微软、苹果、谷歌和其他厂商在其各自的JavaScript浏览器的翻译器性能上进行较量。评估解释型语言的性能是十分棘手的，因为其结果是取决于解释器的实现以及具体的应用，但Box2D的物理引擎的基准[35]发现JavaScript要比Java要慢上5倍，比C语言要慢上10∼12倍，并且发现性能差异的一个因素是使用三个不同的JavaScript解释器。尽管如此，JavaScript比起2011年5月已经快了不少，惠普用它重写了其Palm webOS操作系统的大段部分。我们可以预期这一趋势将继续下去，因为当新硬件可用时JavaScript是第一门受到关注的语言之一，它对于面向用户的应用程序十分实用：例如WebCL提出把JavaScript跟OpenCL语言绑定在一起给图形处理器（GPUs）编程。

1、**生产力。**我们在学习Ruby和Rails时一遍又一遍看到它们的效率与简洁性同步改善。JavaScript的语法是很难达到简洁的，部分原因是因为与Java十分类似市场驱动的要求。CoffeeScript[36]尝试着恢复一些简洁和漂亮的语法更好的适应JavaScript本质。一个源到源的编译器编译CoffeeScript（`.coffee`）文件到`.js`文件中包含正规的JavaScript，并能提供给浏览器。Rails资产管道是自动编译的，所以你不需要手动生成`.js`文件或者把它们包含在你的源代码树中。因为CoffeeScript是由JavaScript编译的，它不能做任何JavaScript先前没有做过的事情，但它为通用的结构提供了许多更简洁的语法符号。举例来说，图6.28展示了图6.20中的Jasmine规则的较少干扰的版本。



SPA开发者使用工具创建以JSON为中心的应用程序来提升性能。举例来说，Yahoo的开源Mojito[37]框架允许同样的JavaScript代码，不论是从JSON客户端还是从基于Node的服务器对HTML进行描绘。然而，按照这个路径进行的应用程序会有一个潜在的巨大缺点：其内容的索引是不可变的并且其搜索引擎也无法搜索到

它，没有了它网络就会失去很大的可用性。对于这个问题的技术解决方案是有的，但这里我们就不多讨论了。除此之外，一些人觉得JavaScript的单线程执行模式会阻碍生产力，因为它需要事件驱动编程，似乎不太可能很快有所改变。有些人对采用例如Node这样的基于JavaScript的服务器框架而感到惋惜，JavaScript库提供的事件驱动版本与POSIX操作系统（类Unix的操作系统）中任务并行代码所使用的工具相同。Rails核心委托Yehuda Katz来对许多有经验的程序员的观点进行汇总：当一些事情是按照顺序发生，例如服务器端代码在SaaS应用中处理控制器行为，其连续模型和阻塞模型更容易进行编程；当发生一些事件时，其顺序是不可预知的，像用户启动的用户界面事件这样的外界刺激反应，异步模型更有意义。你们的作者坚信，未来的软件是“云端+客户端”的应用程序，我们认为对每个工作来说，选择正确的语言或框架比起痴迷于单一的语言或者框架是否将统治客户端跟云端部分的应用程序要重要得多。

最后，早期的Web页面常常是由HTML和CSS创建的（可能会用到WYSIWYG制作工具），如今绝大多数的HTML是通过像Rails这样的框架生成的。与此类似的，CoffeeScript的发展表明尽管JavaScript将会依然是浏览器编程的 **通用语言** ，但是它可能会越来越多地成为一种目标语言，而不是由大多数人直接编写代码的语言。

6.11 深入学习

我们只涵盖了使用JavaScript API的一小部分语言无关的DOM。DOM表示本身还有一组丰富的数据结构和遍历方法，并且其API可用于许多主流的语言，例如`dom4j`[38]是Java库，Nokogiri[39]是Ruby工具包。

这里有一些额外的掌握JavaScript跟jQuery的优秀资源：

- 谷歌JavaScript的专家Miško Hevery的演示： **JavaScript是怎样工作的，一个关于JavaScript和浏览器DOM的介绍** [40]。
- Yehuda Katz[41]是Rails和jQuery的一个活跃的核心提交者。他的面向程序员的博客文章讨论讲述了Ruby跟JavaScript从实用到深奥的技巧和技术。特别的是，他在贴在他的博客上的Ruby块和JavaScript匿名函数微妙的差别[42]和另一个**this**在JavaScript函数的运作方式的原因[43]值得一读。
- jQuery是一个非常强大的并且其潜力未被发掘的库。 **jQuery:Novice to Ninja** （Castledine and Sharkie 2012）是一个十分成功的文献，它有许多远远超出我们介绍能力的好例子。
- **JavaScript语言精粹（The Good Parts）** （Crockford 2008）作者是JSLint[44]的创作者。该书非常固执严谨地阐述的JavaScript，一门心思放在对其良好功能的严格使用上，同时坦率地揭露其设计上的缺陷。如果你有打算编写一套完整的能够比得上Google Docs的JavaScript程序，就“必须”通读这本书。
- ProgrammableWeb[45]的网站列出了成百上千条API，包括可重复编译的和不可重复编译的代码，它们提供XML和JSON数据服务，这在对SPA和混搭网站可能很有用。有一些是完全开源并且无需认证，剩下的需要开发者秘钥，有些是免费的而有些不是。

E. Castledine and C. Sharkie. **jQuery: Novice to Ninja, 2nd Edition - New Kicks and Tricks** . SitePoint Books, 2012.

D. Crockford. **JavaScript: The Good Parts** . O'Reilly Media, 2008.

P. Seibel. **Coders at Work: Reflections on the Craft of Programming** . Apress, 2009. ISBN 1430219483. URL http://www.amazon.com/Coders-Work-Reflections-Craft-Programming/dp/1430219483.

注释

[1]http://pivotal.github.com/jasmine
[2]http://angularjs.org
[3]http://www.ie6countdown.com
[4]http://jquery.org
[5]http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications
[6]http://developers.google.com/closure
[7]http://yui.github.io/yuicompressor
[8]http://jsonlint.com
[9]http://guides.rubyonrails.org/asset_pipeline.html
[10]http://getfirebug.org
[11]http://www.w3.org/DOM
[12]https://github.com/rails/jquery-rails
[13]http://prototypejs.org
[14]http://api.jquery.com
[15]http://api.jquery.com
[16]http://en.wikipedia.org/wiki/Web_Workers
[17]http://johnbintz.github.com/jasmine-headless-webkit/
[18]http://johnbintz.github.com/jasmine-headless-webkit/
[19]https://github.com/searls/jasmine-rails
[20]http://pivotal.github.com/jasmine
[21]http://github.com/velesin/jasmine-jquery
[22]http://pivotal.github.com/jasmine
[23]http://github.com/velesin/jasmine-jquery
[24]http://pivotal.github.com/jasmine
[25]http://github.com/velesin/jasmine-jquery
[26]http://github.com/isaacs/sax-js
[27]https://github.com/pivotal/jasmine-ajax
[28]http://pivotallabs.com/users/msofaer/blog/articles/1423-presenters-and-logical-apis/
[29]http://jslint.com
[30]http://pivotallabs.com/javascriptspecs-bind-reality/
[31]http://jslint.com
[32]http://jslint.com
[33]http://phonegap.com
[34]http://jquerymobile.org
[35]http://blog.j15r.com/2011/12/for-those-unfamiliar-with-it-box2d-is.html
[36]http://coffeescript.org
[37]http://developer.yahoo.com/blogs/ydn/posts/2012/04/
[38]http://dom4j.sourceforge.net
[39]http://nokogiri.org
[40]http://misko.hevery.com/2010/07/14/how-javascript-works/
[41]http://yehudakatz.com
[42]http://yehudakatz.com/2012/01/10/javascript-needs-blocks/
[43]http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this
[44]http://jslint.com
[45]http://programmableweb.com

6.12 建议的项目

练习6.1. 原型继承的一个缺点就是所有的对象属性都是公开的。（回顾一下，在Ruby中 **没有** 属性是公开的：显式地或者使用**attr_accessor**定义的getter和setter方法，是唯一的从类的外部访问属性的方法。）但是，我们可以利用闭包来获取私有属性。为User对象创建一个简单的构造器，可以接收用户名和密码，并且提供**checkPassword**方法来判别提供的密码是否是正确的，但是它不允许检查实际的密码。这个"accessors only"习惯用法贯穿jQuery。
提示：构造器应该返回一个对象，其属性之一是一个函数，利用JavaScript闭包来"记住"初始提供给构造器的密码。

练习6.2. 在6.5节中使用的示例中，假设你不能修改服务器的代码来在电影表的行中添加`adult` CSS类。仅仅使用客户端JavaScript，应该如何识别隐藏的行呢?

练习6.3. 编写JavaScript创建日、月、年的层叠菜单，只允许有效日期的条目。例如，如果2月被选作月，那么Day菜单应该从1～28，除非Year菜单显示为闰年，这时Day菜单才应该从1～29，等等。

作为奖励，把你的JavaScript包装在一个Rails助手里，结果与日期菜单有相同菜单名称和选项标记作为Rails的内置助手，这会让JavaScript成为一个替代品。**注意**：菜单也适用于非JavaScript驱动的浏览器，这很重要；在这种情况下，菜单应该静态显示一个月中的1～31日。

练习6.4. 创建AJAX代码必须创建基于**has_many**关联的层叠菜单。也就是说，给定Rails模型A和B，其中A有**has_many** B，这一对里的第一个菜单应该列出A选项，当其中一个被选中之后，检索相应的B选项并且填充B菜单。

练习6.5. 在ActiveModel（我们在第5章里讲到过）里增加验证功能来自动生成JavaScript代码，在表单提交之前确认表单的输入。例如，假如RottenPotatoe的**Movie**模型断言一部电影一定会有一个非空的标题，那么如果验证不满足的话，JavaScript代码应该防止"Add New Movie"表单提交，给用户显示一个有用的信息，并且突出显示验证有问题的字段。至少要处理内置验证，例如非空、最小/最大字符串长度、数值范围约束以及获得积分，基于正则表达式的验证。

练习6.6. 按照6.5节介绍的jQuery事例方法，使用JavaScript实现一组电影页面列表的复选框，一个用于评级的复选框（G、PG、等等），当检查时允许有这个评级的电影留在列表中。当页第一次加载页面时，所有的都应该被检查；取消选中其中任何一个都会隐藏电影与评级。

练习6.7. 扩展6.6节介绍的示例使得如果用户不断扩大并且在电影表中的同一行崩溃的话，只会发一次请求服务器获取那部电影的信息。换句话说，实现客户端JavaScript缓存每个AJAX调用上检索的电影信息。

练习6.8. 如果你访问`twitter.com`，而加载页面超过几秒的时间，那么会出现一个弹出为出现的延迟道歉，并且建议你再次加载页面。解释一下你将如何使用JavaScript来实现这个行为。
（**提示**：记住JavaScript一旦被加载就可以开始执行，而**document.ready**函数直到文档完全加载和解析才会运行。）

练习6.9. 在本章使用JSON和jQuery技术来使用BDD开发以下的与RottenPotatoes相对应的单页面应用程序（SPA），也就是我们所说的LocalPotatoes。当用户进入到她的

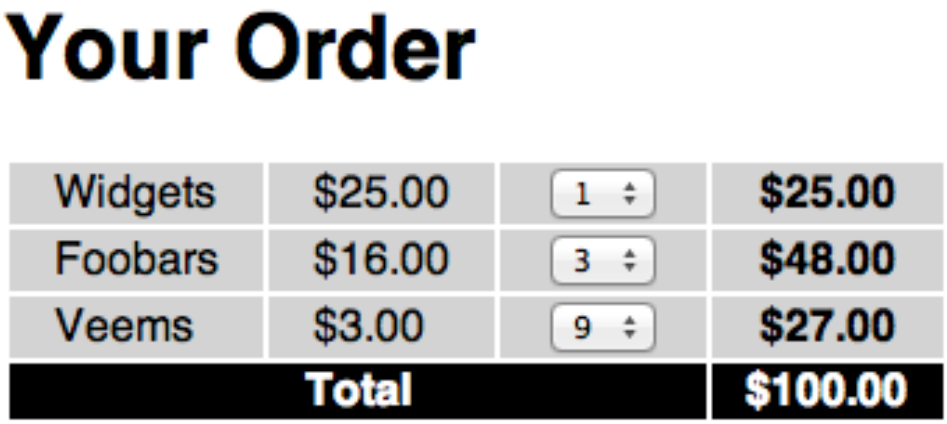


图 6.29: 一个简单的有下拉菜单选择每个物品卖多少的购物车

美国邮政代码（在美国称作“邮政编码”），LocalPotatoes使用Fandango[1]电影粉丝网站免费提供的RSS（真正简易聚合）来检索附近的电影院的名称和位置。这个数据返回XML格式的，所以你将会需要在你的JavaScript代码里做一些最小的XML解析来提取电影院的名称和电影名字。在客户端页面会显示一个电影院的列表；当用户点击电影院的名称时，Google Maps JavaScript API[2]用于把地图集中到电影院的位置，并且电影院上显示的电影列在Movie盒子上。点击电影的名字，使用免费的Open Movie Database API[3]查找关于它的信息，这将会以JSON或者XML的形式返回基本的结果，并且显示电影的宣传艺术品和Internet Movie Database[4]上用户的所有的评级。

练习6.10. 考虑一个网站，卖固定小数目的物品，并且用户只通过与物品名称边的下拉菜单里选择数量来表示每个物品她想买多少。写出简单的JavaScript代码来观察这些下拉菜单，并且每一次他们变化时，都更新Total域中的订单的总价值，也就是把每个物品的数量乘以合适的物品价格然后把这些结果相加。Total域应该是只读的（也就是用户不可改变的）。

练习6.11. 图6.20使用AJAX存根仅仅测试了快乐路径的规则**描述为(“成功”)**。为悲伤路径添加规则**“服务器错误”**和**“超时”**。

第II部分

软件开发：敏捷开发与计划文档模式

7 需求：行为驱动设计用户故事

Niklaus Wirth
（1934——）1984年因开发了一系列创新性的程序语言获图灵奖，这些语言包括Euler、Modula和Pascal。

显然地，程序设计课程应该教授设计与实现的技巧和方法，应该选择可以很好地演示的、循序渐进的 **样例**

——Niklaus Wirth，“Program Development by Stepwise Refinement” **CACM** 14(5)，1971年5月

7.1 行为驱动设计与用户故事简介 **202**
7.2 **SMART**用户故事 **204**
7.3 任务点、速率和**Pivotal Tracker** **206**
7.4 敏捷开发的成本估计 **208**
7.5 **Cucumber**和**Capybara**介绍 **209**
7.6 运行**Cucumber**和**Capybara** **211**
7.7 低精度用户界面模型和故事板 **213**
7.8 优化**Rotten Potatoes** **216**
7.9 显式的和隐式的、命令式的和声明式的情景 **220**
7.10 计划文档的视角 **223**
7.11 谬论和陷阱 **229**
7.12 结束语：**BDD**的利与弊 **230**
7.13 深入学习 **232**
7.14 建议的项目 **234**

重要概念

本章重要概念包括需求获取、成本估算、项目进度安排和进展程度监测。你将看到我们在敏捷开发模式和计划文档生命周期模式下实现这些需求。

这些源于 **行为驱动开发** （ **BDD** ）的敏捷生命周期的概念内容有以下几点：

- **用户故事** 用来引出功能需求。
- **低精度模仿** （Low-fidelity，Lo-Fi） 用户界面和 **故事板** 用以引出用户界面需求。
- **任务点** 用以将用户故事转换为成本估算。
- **效率** 用来测试和估计进度表。
- 使用 **Cucumber** 工具来将用户故事转换成验收测试。
- 使用 **Pivotal Tracker** 工具来跟踪项目进度、计算工作效率以及估计到里程碑的时间。

与计划文档的生命周期相比较，你要熟悉用不同形式表现的完全相同的概念。

- 通过 **访谈** ， **情景** 和 **用例** 进行需求获取，通过 **软件需求说明书** （ **Software Requirements Specification** ）得到需求文档，以及通过 **需求追踪** 做到需求实现。
- 基于项目管理者经验或公式的成本估算方式，例如 **COCOMO** （ **建筑成本模型** ），利用 **PERT**（**Program Evaluation and Review Technique**）**图** 的日程安排与进度监测方法，以及利用 **版本控制系统** 方法来管理文档、日程以及代码的变更管理。
- **风险评估** 和管理用来增加项目成功的机率。

生命周期阐明了 **功能性** 与 **非功能性需求** 之间的不同，以及 **显式的** 和 **隐式的需求** 之间的不同。

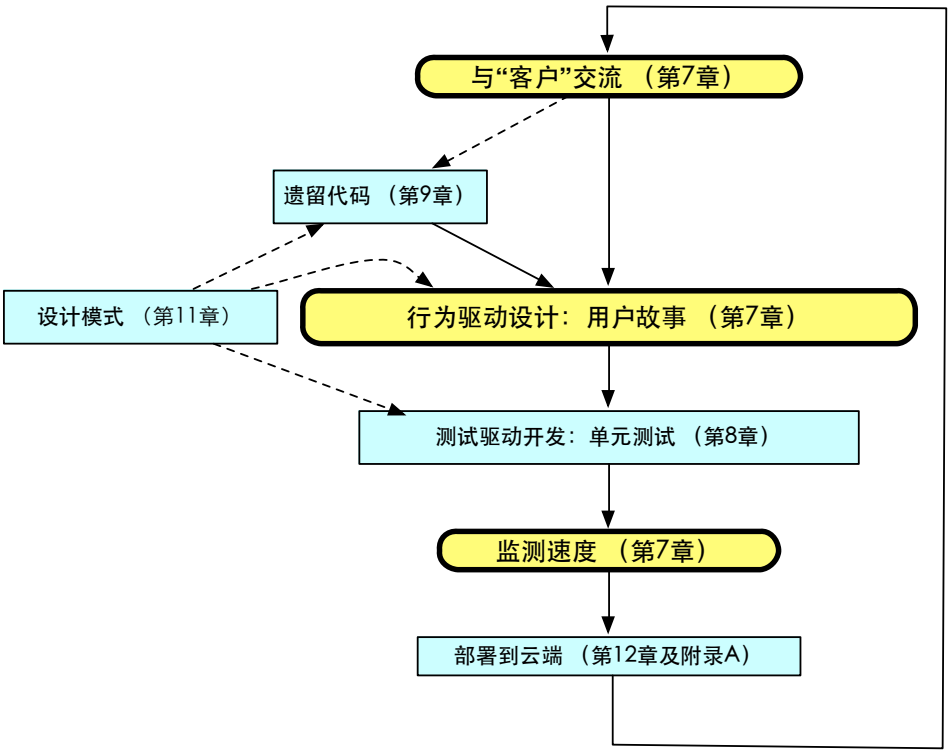


图 7.1：图为敏捷软件生命周期及其关系的迭代过程。本章将重点讨论行为驱动设计的一部分——同用户交谈

7.1 行为驱动设计与用户故事简介

行为驱动设计是正确执行的测试驱动开发。

——匿名

软件工程的失败源于开发者没有做出用户真正需要的东西，或者完成得太迟，或者超出预算，或者他们不能做好维护更新，或者兼而有之。

敏捷生命周期被用来处理许多常见类型的软件中出现的问题。图7.1说明了一种第1章提到的敏捷生命周期的迭代模式，高亮标出的是本章中的部分。我们可以在第1章中看到，敏捷生命周期包含：

敏捷开发涉众 包括用户、客户、开发者、维护程序员、操作员、项目经理……

- 与涉众持续密切合作来实现需求和测试。
- 在部署新功能时，通常每两个星期保持一个工作模式——所谓的 **迭代** ——并且与涉众一起决定下一步该做什么，确认现在的系统是他们真的想要的。保持一个可用原型以及优先开发那些能减小工程未按期完成或超过预算的可能性的功能，以及增加涉众在超出预算的情况下仍然对当前的系统满意的可能性。

与第1章计划文档生命周期不同，敏捷开发并不区分开发模式与维护模式的时间段，也不改变参与人员。在敏捷开发模式下，只要实现了第一组功能，你将基本上一直处于维护模式，这种方法使工程更易于维护更新。

我们从 **行为驱动设计（BDD）** 来介绍敏捷生命周期。BDD在 **开发前和开发中** 询问关于应用行为的问题，以便涉众以及开发者较少可能地传达错误的意图。需求是通过在计划文档模式使用的方法书写的，但与之不同的是，需求是被持续地改进来确保产生的软件符合涉众的期望。换种说法，使用第1章的术语，BDD需求的目标是 **确认**（构建了正确的东西），而不仅仅是 **验证**（正确地构建了东西）

BDD版本的需求就是 **用户故事**，用来描述应用将如何被期望使用。这些轻量级的需求更适合敏捷开发。用户故事帮助软件涉众计划并规划开发的优先级。用户故事能够帮助涉众计划开发过程并做好优先排序。因此，与计划文档模式类似地你也会从需求开始，但在BDD模式中，用户故事替代了计划文档模式中的设计文档。

认真分析应用的 **行为** 以及应用的 **实现**，我们就更容易减小应用涉众之间的误解。同时，我们将在下一章看到，行为驱动设计与强调 **做** 测试实现的测试驱动设计模式（TDD）有很紧密的联系。实践中它们在一起密切合作，但出于教学目的，我们将按顺序介绍它们。

用户故事来源于人机交互（HCI）社区。他们发展了一种3英寸宽、5英寸长（76毫米×127毫米）的索引卡片，即“3–5卡片”。（我们很快将看到其他来自人机界面社区的纸张与笔的技术。）这些卡片包括由客户和开发者写的1∼3个日常的非技术性语言的句子。它的基本理论是纸卡片看起来不可怕并且易于整理，因此可以帮助开展头脑风暴和方便安排优先级。这样，对于用户故事自身来说，它们必须是可测试的、足够小以便在一个迭代周期里实现，而且还要具有商业价值。7.2节将进一步讨论好的用户故事的细节的指导。

需要注意的是个人开发者是一个人工作，没有与客户的交互，因此他们不需要“3–5卡片”。但这些“独行侠”开发者并不符合敏捷开发的哲学中与客户持续密切地合作的规则。

本章以及下一章我们将采用第2章和第4章的Rotten Potatoes应用为例。我们首先来看涉众，对这个简单应用来说这是很简单的：

- Rotten Potatoes的操作员，以及
- Rotten Potatoes的最终用户（电影迷）

我们将在7.8节介绍一个新的功能。但为了帮助理解移动部分，我们从一个Rotten Potatoes已有的功能的用户故事开始，让我们通过一个简单的设置来了解所有部分之间的关系。这个用户故事是用来添加电影到Rotten Potatoes数据库。

本书的配套光盘中包含了可以方便复制粘贴的代码示例。

代码文件：ch_bdd/code/story1.feature

```
Feature: Add a movie to Rotten Potatoes
  As a movie fan
  So that I can share a movie with other movie fans
  I want to add a movie to Rotten Potatoes database
```

这个用户故事格式由创业公司Connextra开发并以它的名字命名的。不幸的是，这个公司已经不存在了。这种格式是：

代码文件：ch_bdd/code/connextra.feature

```
Feature name
  As a [kind of stakeholder],
  So that [I can achieve some goal],
  I want to [do some task]
```

这种格式定义了涉众，不同的涉众可能以不同的方式描述其期望行为。例如，用户可能想要通过链接到信息源从而更方便找到信息，但操作员可能想要链接到预告片以使他们能获得广告收入。所有

三个从句都需要出现在Connextra格式中，但并不需要以固定的顺序。

行为驱动开发与用户故事小结：

- 行为驱动开发强调与涉众合作来定义将要开发的系统的行为。利益相关者包括所有人：客户、开发者、管理者、操作员……
- **用户故事**，一个从人机交互（HCI）社区借用来的方法，使非技术相关人员更易于提出需求。
- **3–5卡片**，每一个都用1∼3个句子描述用户故事。这是一种简易的、没有技术要求的方式，使 **所有** 的涉众都能头脑风暴并能方便安排功能的优先级。
- Connextra格式的用户故事包括涉众、涉众对用户故事的目标，以及当前的任务。

自我检查7.1.1. 对或错：在行为驱动开发中，3–5卡片上的用户故事扮演着计划文档模式中的设计需求的角色。
◇ 正确。

■ **详细阐述：用户故事及用例分析**

用户故事是 **用例分析** 的一种轻量级实现。用例分析是一个在软件工程中传统应用的术语，用来描述相似的流程。一个完整的用例分析包括用例名称、参与者、动作的目的、用例的总结、先决条件（动作之前的全局状态）、行动方案的步骤（包括用户行为以及系统响应）。一个 **用例表** 是一种包括用简笔画小人站起来表示参与者的UML格式的图表（参见第11章），简笔画小人代表参与者。用例表可以用来生成或扩展用例，或包含一些用例作为参考。例如，如果我们有一个用例用来“用户登录”和另一个用例用于“用户登录查看账户小结”，那后一个可以包括前一个作为引用，因为第二个用例的先决条件是用户已经登录。

7.2 SMART用户故事

用户故事的好坏区别在哪里呢？缩略词SMART描述了一个具体且易于记忆的指导：确定性（Specific）、可评估（Measurable）、可实现（Achievable）、相关性（Relevant）以及时间限制（Timeboxed）。

- **确定性。** 这是不确定的功能与其确定性的版本的比较：

代码文件：ch_bdd/code/vaguefeature.rb

```
Feature: User can search for a movie (vague)
Feature: User can search for a movie by title (specific)
```

- **可评估。** 在确定性基础上添加可评估性意味着每个故事都是可以测试的，即每个合理的输入都有确定的预期结果。一个不可评估的功能与可评估的功能比较如下：

代码文件：ch_bdd/code/unmeasurablefeature.rb

```
Feature:  Rotten Potatoes should have good response time (unmeasurable)
Feature:  When adding a movie, 99% of Add Movie pages
          should appear within 3 seconds (measurable)
```

在系统实现需求的情况下，只有第二种情形是可以测试的。

- **可实现。** 从理论上来说，我们在一个敏捷周期内实现一个用户故事。如果在一个周期内实现的故事少于一个，那么这个故事难度太大，你需要把这些故事分割成更小的。第10章描述了名叫 **Pivotal Tracker** 的工具，用来测量 **速率**，即不同困难程度的故事的完成速度。

- **相关性。** 一个用户故事必须对一个或多个涉众有商业价值。为了获取真正的商业价值，一种方法是不停地问"为什么"。我们来看一个地区性电影院售票应用的例子。假设我们的目的是添加一个人人网链接功能，以下是实际操作中的"五个为什么"的递归的问题和答案：

  (1) 为什么要添加人人网链接功能？作为票房经理，我认为更多的人将同朋友一起去看电影，从电影中得到更多享受。

  (2) 为什么要让观众从中得到更多享受？我认为我们会卖更多的票。

  (3) 为什么你想卖更多的票？因为电影院可以赚更多的钱。

  (4) 为什么电影院要赚更多的钱？因为我们想通过赚更多的钱以避免倒闭。

  (5) 为什么电影院明年是否营业很重要？因为如果不这样，我将失业。

  （我们非常确定这项商业价值至少与一个涉众利益相关。）

- **有时间限制。** 时间限制意味着如果你超出时间预算的话，你需要停止开发对应的故事。这时要么放弃，并将用户故事分割成更小的；要么做一个新的评估，重新规划时间，安排剩下的时间和任务。如果分割故事没有帮助，你需要到客户那里寻找用户故事最有价值的部分，并确保这一部分你能很快完成。

  对每一个用户故事都做时间预算的原因是我们极其容易低估软件工程的时间长度。如果不仔细计划每个周期，整个工程将要延期，甚至失败。学会为一个软件工程做预算是项很关键的技能。超过一个用户故事的预算，之后重构这个用户故事，是学习这一技能的一种途径。

SMART用户故事小结：

- **SMART** 缩写涵盖了用户故事令人满意的功能：确定性、可测量、可实现、相关性以及时间限制。
- **五个为什么** 是一种帮助你探索发现用户故事的真正商业相关性的方法。

自我检查7.2.1. 以下项目违反了SMART指导的哪个功能？

代码文件：ch_bdd/code/selfcheckvaguefeature.rb

```
Feature: Rotten Potatoes should have a good User Interface
```

◇ 它不是确定的，不可以被测量，在一个周期内无法实现，并且没有时间限制。它只有商业相关性，只满足了五个中的一个特征。

自我检查7.2.2. 重写以下功能使其符合SMART功能。

代码文件：ch_bdd/code/notSMARTfeature.rb

```
Feature: I want to see a sorted list of movies sold.
```

◇ 以下是该用户故事的SMART版本。

代码文件：ch_bdd/code/SMARTfeature.rb

```
Feature: As a customer, I want to see the top 10 movies sold,
         listed by price, so that I can buy the cheapest ones first.
```

给出用户故事作为捕获用户需求的工作成果，下面介绍一种度量标准及工具来测量其生产力。

7.3 任务点、速率和Pivotal Tracker

一种简单地测量团队生产力的方法是计算每个周期用户故事的完成数，然后计算每星期平均完成故事数。这个平均数将被用于决定每个周期计划要实现的故事数。

这种测量方法的问题在于，一些故事比另一些故事难很多，这会导致错误的预测。一个简单的解决方法是用一个整数来表示每个用户故事的难度。我们推荐使用3个难度级别：1代表简单的故事，2代表中等难度的故事，3代表非常复杂的故事。（随着估计和完成实现经验的增长，你可以用一个更宽的难度范围。）这样平均效率就是每个周期团队完成的 **任务点** 的数目，即 **速率**（velocity）。**待办的工作**（backlog）就是这个周期还没有完成的故事集合。

斐波那契规模 更多地，斐波那契规模一般是：1、2、3、5和8。（每一个数字都是前两个数字的和。）然而，在Pivotal实验室，8已经很稀少了。

注意到速率测量工作效率基于队伍的自我评估。只要团队持续进行用户故事评估，每个周期团队完成了5个还是10个任务点并不重要。速率评估的目的是告诉涉众团队需要多少迭代周期来完成要求的功能。这样可以帮助设定合理的预期，减小他们失望的可能。

Pivotal Tracker（关键跟踪器）是跟踪用户故事和速率的一种服务。图7.2是Tracker的用户界面。一开始需要输入用户故事，并填上难度等级。然后需要按照优先顺序把它们放到当前面板或放入待办面板。每个面板中故事的顺序对应着它们相对的优先级。当故事完成时，它们会被放入一个已完成的面板中，Tracker会按照优先顺序提示待办的工作中的故事。另一个类别是冰箱面板，包含未标注优先级的故事。它们可能无限期地停留在“冻结”状态，但当你准备好开始做它们的时候，把它们拖到当前面板或待办的工作面板。

Tracker允许在用户故事优先列表中插入发布点的标签。由于它基于任务点的完成数来计算效率，所以它支持软件实际发布时的实时评估。这种方法与时间计划管理形成鲜明的对比。在时间计划管理中，经理选择一个发布日期，整个团队努力工作来赶在这个截止日期前完成。

Tracker简介 Pivotal Labs制作了一个很好的3分钟介绍视频[5]给使用Tracker的人。

Tracker最近添加了一项新的概念，它把相关的用户故事分为一种叫 **史诗** 的组。在Tracker中史诗组拥有自己的面板以及进度条，并且它们的用户故事可以在待办的工作中独立地排序。这一功能的目的是给软件工程师一个大的图景，让他们了解到当前应用的开发在大的功能上的进展。

开发者并不需要决定什么时候完成用户故事。开发者按下交付按钮，将它发送给产品经理——与Scrum组织中是完全相同的角色。产品经理尝试使用用户故事，决定点击接受按钮表示用户故事已经完成，或者点击拒绝按钮表示用户故事需要开发者重新来实现完成。

图 7.2: Pivotal Tracker服务的用户界面截屏

团队需要虚拟共享空间来分享信息。Tracker允许开发者给用户故事添加文档，提供一个方便添加界面模型和设计文档的地方。这里有一些其他的信息空间区域，团队可以用它来交流分享包括会议笔记、软件架构等信息：

- 每一个Github库（参见A.7节）都提供了一个Wiki页面，允许团队成员一起编辑文档或添加文件。
- Google Docs[6]允许合作创建和查看图片、报告、电子表格及文本文档。
- Campfire[7]是一个基于Web服务的有密码保护的在线聊天室。

> **小结：**
> 为了让团队管理好每个周期，准确预测团队需要花费多长时间来实现新的功能，团队将任务点作为用户故事的难度，并追踪团队的 **速率** ，或每个周期的平均任务点。Pivotal Tracker提供了一种帮助完成划分优先顺序并记录用户故事及其状态，计算速率，根据团队历史数据预测软件开发时间的服务。

自我检查7.3.1. 对或错：当比较两个团队时，速率更高的团队生产效率也更高。
◇ 错误。因为每个团队对用户故事会指定不同的任务点，你不能用以任务点计的效率来比较不同的团队。但是，你可以通过比较一个团队在不同周期的效率来发现它在不同时间是否有不同生产效率。

自我检查7.3.2. 对或错：当你不知道如何实现一个给定的用户故事时，将它的任务点设置为3点。
◇ 错误。一个用户故事可能非常复杂以至于你无法用一个方法来实现它。如果这样的话，你应该回到涉众那里重构该用户故事，将它们构造成一些简单的你知道如何去做的任务。

7.4 敏捷开发的成本估计

Pivotal Labs 是一个提供敏捷开发培训、咨询并与客户合作开发特定软件产品的公司。

鉴于敏捷宣言认为客户合作的重要性大过合同谈判，我们并不惊讶于它 不 遵从计划文档中的方法。我们将在7.10节中看到，给定一组功能，计划文档模式中用成本估计和日程计划用来帮助他们赢得合同。本节主要讲述Pivotal Labs基于敏捷开发（Burkes 2012）的实现过程。

由于Pivotal使用敏捷开发，它从不保证在日期D前完成功能X、Y和Z。Pivotal保证以最高效的方式在日期D之前提供一定数量的工作所需资源。通过这种方法，Pivotal要求用户与项目团队一起工作来定义优先级，让Tracker服务的速率指导决策，决定在日期D之前发布的版本中实现哪些功能。

当一个潜在的用户第一次与敏捷开发团队接触时，如果这个工作适合敏捷开发团队，那么首先他们会通过30∼60分钟的电话告诉用户合约是什么样子的，与其它的“外包”代理商有什么不同，他们要求用户提供怎样的时间承诺，等等。首次电话商谈确定敏捷开发团队将在给定一个时间和素材基础上工作，而不是在一个固定的出价上。后者是计划文档模式的常见过程。敏捷开发团队让用户给出一个他们想要实现的软件高度抽象的描述，他们当前的开发进程如何，以及他们当前的雇员情况，等等。

如果用户对他们得到的消息感到满意，并且敏捷开发团队也认为这是一个合适的项目，用户将访问一个Pivotal称为“展望”的讨论。展望通常是指一个与潜在客户

的约90分钟的对话，大多是个人的。敏捷开发团队会要求客户派产品负责人，首席开发员（如果有）、设计师（如果有）以及他们预期结果的现有设计，等等。基本上，用户代表会带来任何能帮助描述清楚他们预期结果的资料或人。敏捷开发团队有两人参加展望会议。

在展望会期间，敏捷开发团队将要求用户详细地描述他们预期的结果，并询问一系列问题来确定还不清楚的、有风险的以及外部整合的问题等等。本质上来说，敏捷开发团队希望确定任何有可能给团队需要处理的评估带来不确定性的事物。如果敏捷开发团队遇到了一个非常清楚地定义了他们预期产品，一个完成了设计并且没有外部整合等等的用户，团队就可能做出一个相当精确的估计，例如“20～22周”。另一方面，如果用户没有清晰的产品定义，并且有很多外部整合问题，或其它不确定性，团队的评估结果就可能有很大的跨度，例如“18～26周”。如果像Pivotal Labs一样使用结对编程（参见10.2节），评估结果将是以“两人—周”为单位的。

用户离开展望会后，与项目相关的主要工程师会留下来15～30分钟，讨论并确认项目星期数的估计。之后，他们会将他们的结论，包括评估、风险识别等等交给销售人员，销售人员接下来会将这些结果写成一个完整的计划并用邮件发给用户。

因为敏捷开发团队只处理时间和素材，我们很容易从估计的星期数得到费用的估计范围。

小结：
敏捷宣言强调客户要参与合同，因此一个敏捷开发团队的“成本评估”的概念更多地是关于建议用户选择怎样规模的团队来获得最大的效率。根据布鲁克定律，团队规模的效率有一个收益减小点（参见7.11节）。敏捷团队在展望过程中的目标是确定这个点，然后随时间提高团队的规模达到这个点。敏捷开发公司通过与外部客户的简短的讨论来确定时间和素材要求，并以此决定成本。我们将在7.10节看到，这种方法与计划文档过程的公司有鲜明的对比，后者向客户承诺在商定费用基础上，一组功能会在某个日期前完成。

自我检查7.4.1. 对或错：由于使用了敏捷开发，Pivotal Labs不需要合同。
◇ 错误。Pivotal当然需要提供合同让用户签署，但是它主要是向客户承诺：客户付费用给Pivotal，它会用最好的努力让客户在一段有限的时间内尽量满意。

我们已经知道用户故事在进度测量中扮演着非常重要的角色，下面介绍一个让用户故事扮演另一个重要角色的工具。

7.5 Cucumber和Capybara介绍

神奇的是，**Cucumber** 工具将客户理解的用户故事转化为验收测试和集成测试。验收测试用以确保用户满意。同时 **集成测试** 可以确保不同模块之间接口的假设一致性以及通讯正确性。（第1章描绘了测试的类型）。重要之处在于，Cucumber是客户和开发者之间的桥梁：用户故事并不像代码，因此它们对客户来说非常清楚，可以用来达到协议。但它们并不是完全是自由的。本节将介绍Cucumber如何实现这个小小的奇迹。

在Cucumber环境中，我们将用术语 **用户故事** 来代表包含一个或多个 **情景** 来示范不同方式的 **功能** 是如何被使用的。关键词**功能**和**情景**确定了各自的组件。每一个情景都是由3～8个 **步骤** 序列组成的。

图7.3是用户故事的一个简单例子，显示了只有一个添加电影 **Men In Black** 情景的功能。这个情景有8个步骤。（此处我们只展示了一个单独的情景，但功能通常有

代码文件：ch_bdd/code/AddMovieStory.feature

```
1  Feature: User can manually add movie
2
3  Scenario: Add a movie
4    Given I am on the RottenPotatoes home page
5    When I follow "Add new movie"
6    Then I should be on the Create New Movie page
7    When I fill in "Title" with "Men In Black"
8    And I select "PG-13" from "Rating"
9    And I press "Save Changes"
10   Then I should be on the RottenPotatoes home page
11   And I should see "Men In Black"
```

图 7.3：一个与添加Rotten Potatoes电影功能有关的Cucumber情景

Cucumber关键词
Given（给定）、**When**（当.....时候）、**Then**（然后）、**And**（以及）和**But**（但是）有不同的名字，仅仅是为了方便读者。但是它们都是同一种方法的别名。这样，你就不用记忆许多关键词的语法了。

多个情景。）尽管描述呆板，但Cucumber的这种格式对非技术人员的用户来说仍然易于理解，并能帮助开发工作。这符合敏捷开发和行为驱动开发的基本原则。

每一个情景的每一步都从它的关键词开始。以**Given**（给定）开头的步骤通常设定先决条件，例如导航到某一页。以**When**（当.....时候）开头的步骤一般使用Cucumber内置的网页步骤来模拟用户按下按钮。以**Then**（然后）开头的步骤一般检查一些条件是否为真。连接词**And**（以及）允许更复杂版本的**Given**（给定）、**When**（当.....时候）、**Then**（然后）短语。这种格式中你看到的余下的一个关键词是**But**（但是）。

一组额外的文件定义了测试这些步骤的Ruby代码。这些被称为 **步骤定义**（step definitions）。一般情况下，许多步骤都能映射到单个步骤的定义上。

那么Cucumber是如何将情景中的步骤与 **步骤定义** 对应到一起并进行测试呢？技巧在于Cucumer使用 **正则表达式**（参见第3章）来将情景中的步骤的英文短语映射到测试程序中的步骤定义。

例如，如下字符串是一个Rotten Potatoes的情景中的步骤定义：

Given /^(?:|I)am on (.+)$/

这个正则表达式可以匹配图7.3中第4行的“I am on the Rotten Potatoes home page”。同时也可以捕获字符串“am on”后面直到行末的部分（“the Potten Potatoes home page”）。这个步骤定义的主体包含Ruby代码来测试对应步骤，就像上述捕获字符串一样。

这样，一种考虑步骤定义和步骤之间关系的想法是：步骤定义就像方法定义，而情景的步骤就像方法调用一样。

接下来，我们需要一个模拟用户使用这些功能的工具，在测试不同情景进行测试。在Rails环境下，这种工具被称为 **Capybara**，而Cucumber能和它无缝地拼接起来。Capybara作为一个“模拟的用户”执行Web浏览器中的操作。例如，点击一个链接或按钮。Capybara可以与应用交互、接收网页、解析HTML以及像用户一样提交表单。

■ **详细阐述：为Web建立桩**

可惜的是，Capybara并不像JavaScript一样足够智能，我们在第6章中看到了JavaScript。不过幸运的是，通过适当的选项，Capybara能够和Webdriver进行交互使其真地运行并遥控浏览器，通过“远程控件”来使它像用户故事描述的一样工作。在本章，我们并不介绍这个模式，因为它比“模拟”模式要慢很多，而且“模拟”模式对除了JavaScript测试以外的所有测试都很合适。本章结尾的图7.15展示了这些工具间的关系。

Cucumber小结：

- Cucumber结合了用户想要添加一组 **情景** 的 **功能** 。我们把这种结合称为 **用户故事** 。
- 情景的步骤使用关键词**Given**（给定）来定义当前状态，使用**When**（当.....时候）来定义操作，以及**Then**（然后）来定义操作序列。
- 情景步骤还用关键词**And**（以及）和**But**（但是）作为连接词，实现更复杂的状态、操作以及序列的描述。
- **Cucumber** 用 **正则表达式** 的方法把 **步骤** 映射到 **步骤定义** 。
- **Capybara** 用模拟用户和浏览器执行情景步骤的方法对SaaS应用做一些测试。
- 把 **功能** 以及包含许多步骤的不同 **情景** 的功能存储在文件中，将Ruby代码保存在包含用来测试每种步骤的 **步骤定义** 的独立文件中，Rails工具 **Cucumber** 和 **Capybara** 将自动测试SaaS应用的行为。

自我检查7.5.1. 对或错：Cucumber通过正则表达式将情景步骤映射到步骤定义，Capybara模拟一个用户依照这些情景与SaaS应用进行交互。
◇ 正确。

7.6 运行Cucumber和Capybara

Cucumber中用户故事的一个主要好处是 **"红—黄—绿"分析** 。当用户故事写好时，我们马上就能试着运行它。一开始，步骤传递被高亮标记为"红"（失败）或"黄"（未实现）。我们的目标是通过逐渐地增加代码使它通过测试并使每一个步骤都从黄或者红变成"绿"（通过）。在一些情况下，这个操作很简单。下一章，我们将同样试着在 **单元测试** 上使它从红变成绿。请注意单元测试是对单独的方法来说的，而Cucumber的情景能够测试应用中的整个路径，因此Cucumber更适合用来做验收测试或集成测试。

Cucumber是绿色的 通过测试的被标记绿色，正是这款工具的名称来源cucumber（黄瓜）的颜色。

和其它我们看到的有用工具一样，Cucumber是一个Ruby工具，所以我们要做的第一件事情就是确认我们的应用依赖于这个项目并且使用Bundler来安装它。新建一个`myrottenpotatoes`的应用，如第4章开始的一样，添加以下内容到`Gemfile`。我们提到过，Cucumber和其相关的项目都只在测试和开发环境中被需要，在产品环境中并不需要（4.2节介绍了Rails应用运行的三种环境）。

代码文件：ch_bdd/code/add_cucumber_to_gemfile.rb

```
# add to end of Gemfile
group :test, :development do
  gem 'cucumber-rails'
  gem 'cucumber-rails-training-wheels' # some pre-fabbed step definitions
  gem 'database_cleaner' # to clear Cucumber's test database between runs
  gem 'capybara'         # lets Cucumber pretend to be a web browser
  gem 'launchy'          # a useful debugging aid for user stories
end
```

只要你更新了`Gemfile`，运行`bundle install --without production`。如果一切顺利，你最后会看到“Your bundle is complete.” 我们现在需要建立Cucumber和Capybara需要的目录和“模板”文件。和Rails一样，Cucumber自带一个做这件事的**生成器**。在应用的根目录里，执行以下两条命令（如果有提示问是否覆盖已有的文件例如`cucumber.rake`等，你可以安全地回答是）：

```
rails generate cucumber:install capybara

rails generate cucumber_rails_training_wheels:install
```

执行这两个生成器可以给出作为起始模板的常用的步骤定义，例如与一个web浏览器的交互。在这个应用里，你将在`myrottenpotatoes/features/step_definitions/web_steps.rb`里看到它们。另外，对这些预先定义的步骤，你将需要新建步骤定义来匹配应用的独特功能。你需要学习一下最常见的预先步骤定义，以及如何使用它们使得通过更少的步骤定义来实现你要的功能。

在运行Cucumber之前，还有一步必须要做：你必须通过运行`rake db:test:prepare`来初始化测试数据库。你需要在首次运行测试和每次数据库模式更改时运行这个命令。4.2节给出了相关的详细介绍。

现在，你已经准备好开始使用Cucumber了。你在`features`目录下添加了扩展名为`.feature`的文件作为功能。把图7.3中的用户故事复制到`features`目录下的`AddMovie.feature`文件中。为了查看情景和步骤定义的交互以及它们如何像新英格兰的枫树一样随季节改变色，输入

```
cucumber features/AddMovie.feature
```

请参考下述视频来执行接下来的操作。

视频7.6.1: Cucumber第一部分.
光盘文件：ch_bdd/mov/AddMovieCuke.mp4
这段视频显示了Cucumber如何通过给步骤染色来显示测试是否通过。红色的是失败的步骤，黄色的是未实现的，绿色的是通过测试的步骤。（红色的失败步骤后面的是蓝色的，意味着它们还没有被测试。）第4行的第一个步骤是红色的，所以Cucumber跳过了那个测试。它失败的原因如Cucumber消息显示的一样，是paths.rb中没有路径能匹配“the Rotten Potatoes home page”。这个消息甚至建议了修复这个失败的方法：添加路径到`paths.rb`.这个新的路径将第一个步骤变成像黄瓜一样的绿色，但这时第6行的第3个步骤是红色的。错误信息指出，它失败的原因是没有路径匹配“Create New Movie page”,我们可以同样通过添加路径到paths.rb来修复它。现在所有的步骤都和（绿色的）黄瓜一样酷，添加电影的情景测试就通过了。

小结：
添加功能作为BDD的一部分，首先需要定义接受的标准。Cucumber能够做到从用户故事中捕获需求并且从用户故事中得到验收测试和集成测试。更进一步地，我们能获得的是能自动运行的测试，这样我们就能创建回归测试来帮助我们在更新代码时还能保证这些功能正常工作。（我们将在第9章中的一个比Rotten Potatees更大的例子中看到回归测试的用途。）

自我检查7.6.1. Cucumber标记通过测试的步骤为绿色。那么标记为黄色和红色的步骤有什么区别是?

◇ 黄色步骤表示这个步骤还未实现，红色步骤表示这个步骤已经实现但是没有通过测试。

7.7 低精度用户界面模型和故事板

新功能一般都伴有一个用户界面（UI），因为许多SaaS应用都与使用者有交互。这样，BDD的一部分任务经常是提出一个用户界面来满足用户故事。如果一个用户故事说用户需要登录，那么我们需要展示一个登录的界面的实体模型。可惜的是，编写用户界面的软件原型可能会导致用户不敢提出改进需求，这与我们早期设计的需要是相悖的。

我们想要的是3–5卡片的UI等价物。为了吸引对非技术涉众的参与，鼓励反复试验，这种造价物应该容易被改变甚至舍弃。就像HCI社区为用户故事提倡3–5卡片，他们推荐在UI实体中使用幼儿园会用到的工具：蜡笔、画图用纸以及剪刀。他们称用户界面这种低技术含量的方法为 **低精度UI**，并称纸制原型为 **草图**。例如，图7.4展示了添加电影到Rotten Potatoes的一个低精度的草图。

理想情况下，你为所有与UI有关的用户故事添加草图。这看起来是冗长乏味的，但最终在使用HTML来实现真正的UI时你必须指定所有的UI细节，而且用铅笔和纸来实现正确的UI比用代码要简单很多。

低精度草图展示了UI在某个时间的实例下的样子。然而，我们需要显示草图怎样才能像一个页面的用户界面那样一起工作。电影制作者在场景问题上面临一个类似的挑战。他们的解决办法，即 **故事板**，就像一本漫画书一样贯穿整个电影，每个场景上都画上了图画。与电影的线性次序不同的是，UI的故事板通常是一个树或图，屏幕切换由不同用户的选择决定。

为了实现一个故事板，我们必须考虑一个Web应用的所有用户界面：

- 页面或部分页面
- 表单和按钮
- 弹出菜单

图7.5展示了一系列低精度草图，有用户点击会引起草图之间转换的指示。

在画完草图和故事板后，你已经准备好写HTML了。第2章讲述了Haml如何标记成为HTML，以及HTML元素的**class**和**id**属性能通过层叠样式表（CSS）被用于添加格式信息。低精度方法的关键在于从你的草图中获得一个好的总体结构，并使用最小的CSS（如果有的话）来获得和你的草图看起来相似的视图。提醒一下，页面布局的公共部分——横幅、结构化的`divs`等等——可以加入`views/layouts/application.html.haml`中。

从观察低精度UI草图开始，把它们布局分割成“块”。使用CSS的`divs`来构建图层明显的部分。在所有事情工作正常之前并不需要把它做的很漂亮。添加CSS样式、图片等等是个有趣的部分，但请在它正常工作 **之后** 再开始试图把它弄得好看一些。

7.5节介绍的例子与现有功能有关，并不需要更新Haml或CSS。下一节会给Rotten Potatoes添加新的功能，Haml也会随之改变。

小结：
同样是从HCI社区借鉴，**低精度UI** 是探索用户故事对应的用户界面的低成本方法。纸和笔能让改变或舍弃变得简单，这涉及所有涉众。**故事板** 根据用户的动作来捕获不同页面间的交互。在这种低成本媒介下进行实验比用Haml和CSS创建HTML页面后试验要节省很多功夫。

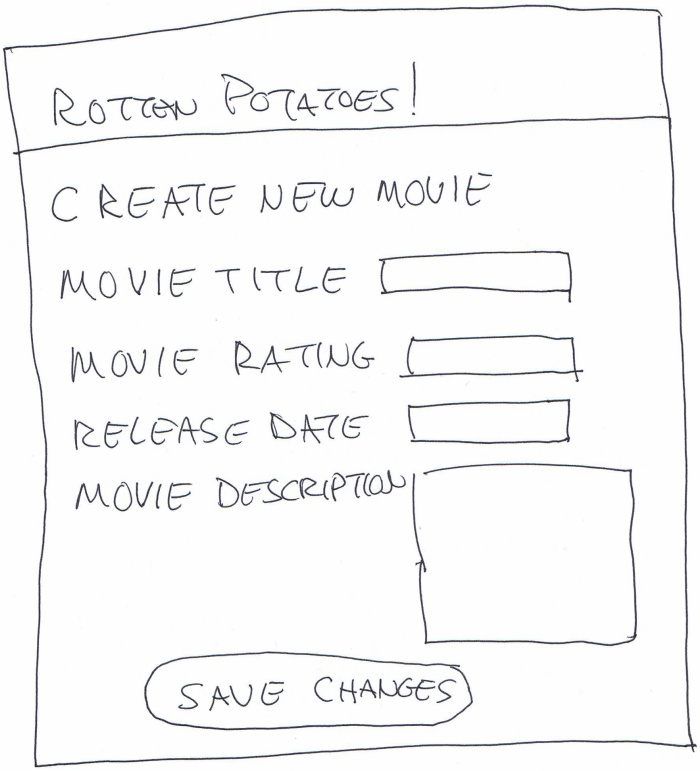


图 7.4: 添加一个电影到Rotten Potatoes时的窗口

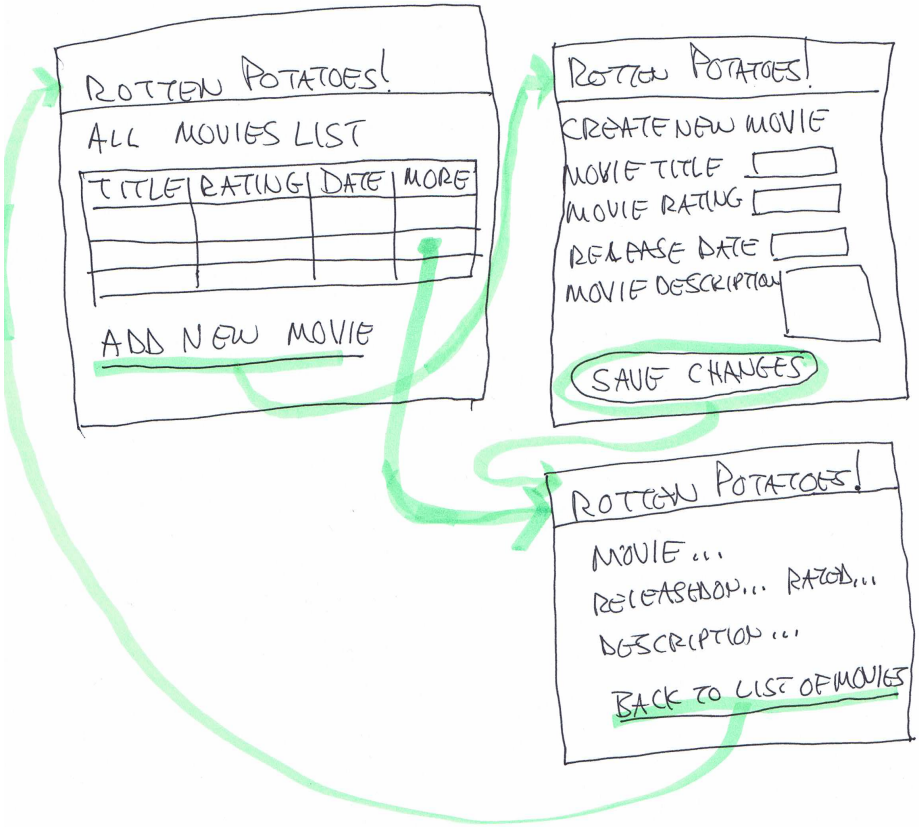


图 7.5: 添加电影到Rotten Potatoes的故事板图片

自我检查7.7.1. 对或错：低精度用户界面和故事板的目的是在运行用户界面之前调试它。
◇ 正确。

7.8 优化Rotten Potatoes

作为用户故事和低精度UI的第二个例子，假设我们想要搜索公开电影数据库（TMDb）来找到一部我们感兴趣添加到Rotten Potatoes的电影的信息。和我们将在第8章中看到的一样，TMDb有自己的API（应用程序接口）允许在面向服务的架构下访问它的信息。

本章使用了Cucumber来设计两个情景，与之相关的低精度UI草图来展示我们想要Rotten Potatoes如何与TMDb一体化，以及通过暂时“欺骗”一些代码来使其中一个情景变成绿色。在第8章中，我们将写一些代码来使另一个情景变成绿色。让第一组情景运行看起来有些繁琐，因为你不得不经常添加许多公共结构，但之后这会变得快很多，事实上你甚至可以重用你之前的步骤定义来构造更高级别的“声明性”步骤，就像我们将在7.9节中看到的那样。

图7.6中的故事板显示了我们预想的功能工作的情况。我们将在Rotten Potatoes的首页默认列出所有电影，并添加一个可以输入一些关键词的搜索框以及一个“搜索”按钮，通过找到那些标题中包含这些关键字的方式搜索TMDb中的电影。如果搜索有了匹配结果，那么所谓的“快乐路径”（happy path）将被执行——第一个匹配的将被“预填充”到第4章中开发的添加新电影页面中。（在真实的应用中，你可能想创建一个独立的页面来展示所有的匹配，让用户来选择一个。但我们这里是故意使例子变得简单。）如果搜索没有匹配到任何电影——“悲伤路径”（sad path）将被执行——我们将回到首页并用一条消息通知用户没有找到。

一般情况下，你应该首先完成快乐路径。当你遇到需要写 **新** 代码的失败步骤或未开发步骤时，你通过测试驱动开发（TDD）写这些代码。我们将在第8章来写一些代码真正调用TMDb并把它整合到现在的情景中。现在，我们要开始用悲伤路径来阐明Cucumber的功能和BDD过程。图7.7展示了新功能的悲伤路径情景，创建一个包含这些代码的文件`features/search_tmdb.feature`。当我们用`cucumber features/search_tmdb.feature`运行这些功能，第二步 **Then I should see “Search TMDb for a movie”** 应当会失败（红色），因为我们还没有添加这些文本到首页`app/views/movies/index.html.haml`中去。所以，首要任务是改写代码来使这个步骤变成绿色。

严格来说，一个“纯粹”的BDD方法能仅仅通过在任何地方添加文本 **Search TMDb for a movie**，然后重新运行这个情景使这个步骤通过测试。但实际上我们知道，下一步的 **When I fill in “Search Terms” with “Movie That Does Not Exist”** 仍然会失败，因为我们还没有添加名为“Search Terms”的表单区域到视图中。所以基于效率的考虑，我们在`index.html.haml`中添加图7.8中的代码。我们将在下面具体解释。

第3行文本允许 **Then I should see “Search TMDb for a movie”** 通过测试。剩下的部分构造了输入表单，我们在第4章中介绍过的，所以你对一些标记应该是熟悉的。以下两点是显而易见的。第一，在视图中与任何用户的交互，我们需要一个控制器动作来处理这个交互。在本例中，包含搜索关键词的交互操作被提交到表单中。第5行中说到，当表单被提交时，控制器动作**search_tmdb**将接收到表单提交。显然这些代码现在还不存在，但是我们需要给这个动作选择一个描述性的名称。

第二件要注意的事情是HTML`label`标签的使用。第2章的图2.14告诉我们，第7和8行将被展开成以下的HTML标记：

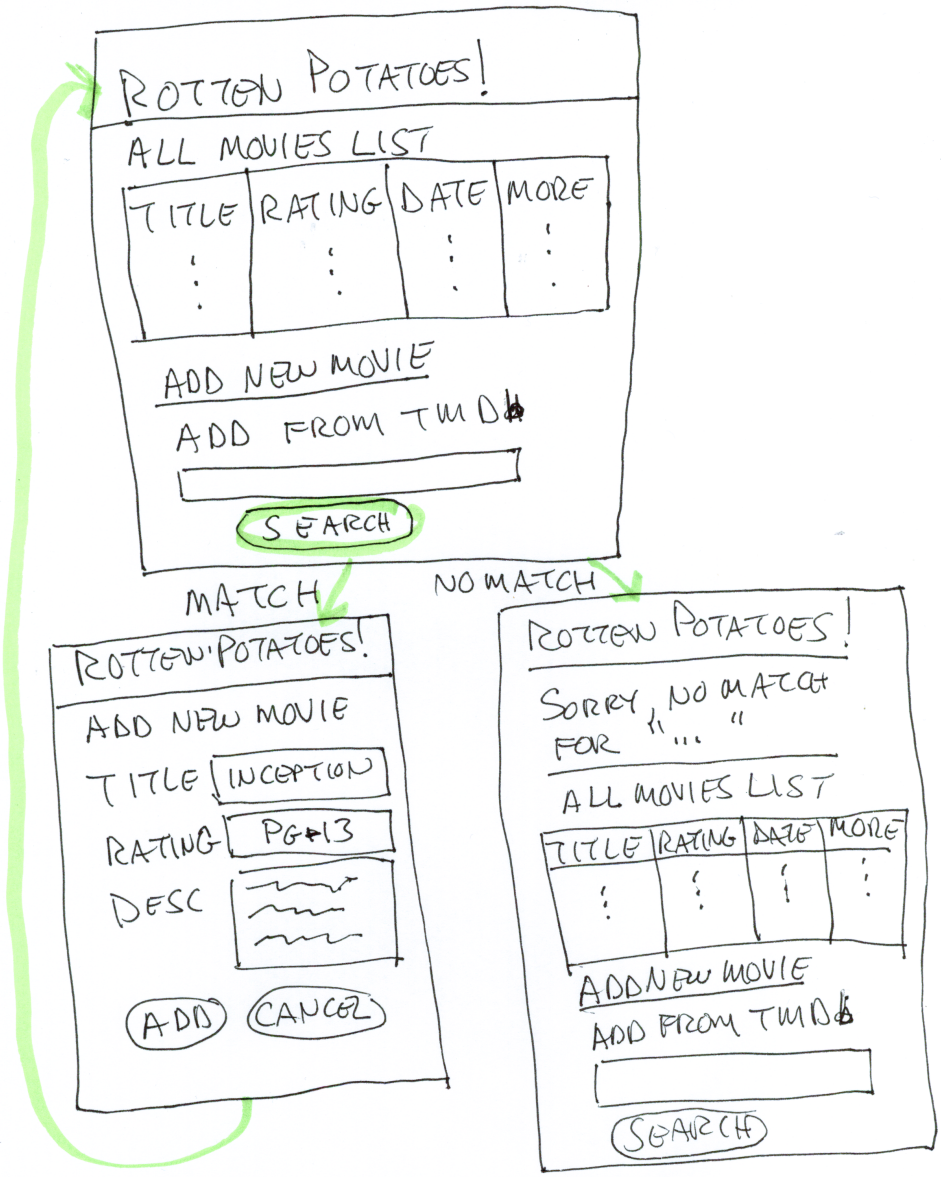


图 7.6: 搜索电影数据库的用户界面故事板

代码文件：ch_bdd/code/AddTMDbSearch.feature

```
Feature: User can add movie by searching for it in The Movie Database (TMDb)

  As a movie fan
  So that I can add new movies without manual tedium
  I want to add movies by looking up their details in TMDb

Scenario: Try to add nonexistent movie (sad path)

  Given I am on the RottenPotatoes home page
  Then I should see "Search TMDb for a movie"
  When I fill in "Search Terms" with "Movie That Does Not Exist"
  And I press "Search TMDb"
  Then I should be on the RottenPotatoes home page
  And I should see "'Movie That Does Not Exist' was not found in TMDb."
```

图 7.7：与添加搜索电影数据库的功能关联的悲伤路径情景

代码文件：ch_bdd/code/add_form_to_homepage.html.haml

```
-# add to end of app/views/movies/index.html.haml:

%h1 Search TMDb for a movie

= form_tag :action => 'search_tmdb' do

  %label{:for => 'search_terms'} Search Terms
  = text_field_tag 'search_terms'
  = submit_tag 'Search TMDb'
```

图 7.8：搜索TMDb页面中的Haml代码

反复这样做？ `rake cucumber`运行你所有的功能，或更精确地，运行这些由Cucumber的配置文件`cucumber.yml`中的 **默认的配置文件** 选择的。[8]
在下一章中我们将看到一个叫做`autotest`的工具，能够在你更新文件的时候自动重新运行测试。

代码文件：ch_bdd/code/add_form_to_homepage.html

```
<label for='search_terms'>Search Terms</label>
<input id="search_terms" name="search_terms" type="text" />
```

关键在于`label`标签的`for`属性来匹配`input`标签的`id`属性，这是由在图7.8第8行调用的**text_field_tag**助手第一个参数决定的。这个通信允许Cucumber来决定图7.7: **When I fill in "Search Terms"...** 中第11行的名称"Search Terms"引用怎么样的表单区域。

你可能会记起4.1节中提到的，我们需要确保有一个新的控制器动作的路线。图7.9上图展示了你必须添加到`config/routes.rb`的路线，该路线是用来添加一个表单提交（`POST`）路线到动作中去。

然而，即使有了新的路线，这个步骤依然会因为一个异常失败：尽管我们有一个名为"Search TMDb"的按钮，**form_tag**指定了**MoviesController#search_tmdb**是一个控制器动作，应该接受一个表单，但这个方法现在在`movies_controller.rb`中还不存在。图7.1中说，我们现在应该使用测试驱动开发（TDD）技术来创建新的方法。但由于TDD是下一章的主题，我们将通过"作弊"来使这个情景运行起来。由于这是没有搜到电影的"悲伤路径"情景，我们将临时地创建了一个控制器方法使得它能 **永远** 地表现得和没找到一样。这样我们就能完成悲伤路径的测试。还有，图7.9展示了你应该添加到`app/controller/movies_controller.rb`的用以创建"假的"**search_tmdb**动作的代码。

如果你是刚开始接触BDD，这一步可能会使你感到惊讶。我们为什么故意构造一个并不真正调用TMDb的假的控制器方法，来假装搜索失败了？在这个例子中，答

代码文件：ch_bdd/code/controller_route.rb

```
# add to routes.rb, just before or just after 'resources :movies' :

# Route that posts 'Search TMDb' form
post '/movies/search_tmdb'
```

代码文件：ch_bdd/code/controller_method.rb

```
# add to movies_controller.rb, anywhere inside
#  'class MoviesController < ApplicationController':

def search_tmdb
  # hardwire to simulate failure
  flash[:warning] = "'#{params[:search_terms]}' was not found in TMDb."
  redirect_to movies_path
end
```

图 7.9: 上图：一个当一个表单被提交时触发这个进程的线路。下图：这个"假的"控制器方法永远表现得像没有匹配一样。它检索来自参数hash的用户在表单中输入的关键字（如我们在第4中看到的），在flash[]存储一个消息，然后把用户重定向到电影目录。回忆下第4章我们在app/views/layouts/application.html.haml中添加的代码来在每一个视图上显示flash目录

案是因为它能让我们完成情景的其他部分，确保HTMLP视图符合低精度草图，且视图转换顺序符合故事板的要求。事实上，一旦你做图7.9所示的修改，整个悲伤路径都会通过测试。视频7.8.1总结了我们目前为止所做的所有工作。

视频7.8.1: Cucumber第二部分.
光盘文件：ch_bdd/mov/SearchTMDbCuke.mp4
在这段视频中，我们构造了悲伤路径来阐明Cucumber的功能，因为它能使用已有的代码。图7.7第5行的第一步通过了测试，但第6行失败了，因为我们还没有更新`index.html.haml`使其包含新的页面的名称或包含一个用来输入一个电影来搜索的表单来。我们使用与4.4节和4.6节所述相同的方法，通过添加这个表单到`index.html.haml`来修正它。这里并没有匹配传入的URI的线路。为了使事情简化，我们使用4.1节讨论的方法为`config/routes.rb`中的动作建立了一条线路。当创建一个表单时，我们必须指定哪个控制器动作来接收它。我们将这个控制器方法命名为`search_tmdb`。（我们将在下一章实现这个方法）。一旦我们更新了`index.html.haml`，在`config/routes.rb`中创建一条新的线路，并且为控制器方法命名后，Cucumber会把所有的步骤都标为绿色。

那么快乐路径，即当我们搜索一个已经存在的电影时会怎样？观察那条路径上的前两个动作——打开Rotten Potatoes首页，确保那里有一个搜索表单，与图7.7中第9到第10行相对应的——和悲伤路径是一样的。这时应该就像在你脑袋里摇了巴甫洛夫铃铛一样，条件反射地问你自己如何能精简这些重复。

图7.10展示了答案。更新`features/search_tmdb.feature`来匹配该图，再运行`cucumber features/search_tmdb.feature`。不出意外地，第24行的步骤失败了，因为我们硬连接了控制器方法，假装没有在TMDb中找到任何匹配，结果是转到了首页而不是搜索结果页。此时，我们需要改变控制器方法到硬连接使它变成绿色，但这样就会使悲伤路径变成红色。下一章将介绍一个更好的方法，即我们将使用测试驱动开发（TDD）来开发 **真正的** 控制器动作，用以"欺骗地"建立输入以及全局的状态，用来在隔离的环境中测试特殊的条件。当你真正掌握了BDD和TDD的时候，你会在日常的软件开发中不断在这两种方法中反复迭代。

代码文件：ch_bdd/code/happy_and_sad_paths.feature

```
Feature: User can add movie by searching for it in The Movie Database (TMDb)

  As a movie fan
  So that I can add new movies without manual tedium
  I want to add movies by looking up their details in TMDb

Background: Start from the Search form on the home page

  Given I am on the RottenPotatoes home page
  Then I should see "Search TMDb for a movie"

Scenario: Try to add nonexistent movie (sad path)

  When I fill in "Search Terms" with "Movie That Does Not Exist"
  And I press "Search TMDb"
  Then I should be on the RottenPotatoes home page
  And I should see "'Movie That Does Not Exist' was not found in TMDb."

Scenario: Try to add existing movie (happy path)

  When I fill in "Search Terms" with "Inception"
  And I press "Search TMDb"
  Then I should be on the "Search Results" page
  And I should not see "not found"
  And I should see "Inception"
```

图 7.10: 通过background（背景）关键词，将需要在一个情景文件中的 每个 情景之前执行的步骤组合在一起，以避免这些快乐路径和悲伤路径的重复

小结：

- 在Saas应用中添加一个新的功能一般意味着你需要指定这个功能的用户界面，写下新的步骤定义，可能在Cucumber能成功将步骤标记为绿色之前还要写新的方法。
- 一般来说，你最好首先写完快乐路径的情景。我们从悲伤路径开始仅仅是由于它能让我们更好地阐明Cucumber的一些功能。
- **Background**（背景）关键词能通过将相关的情景写到同一个功能文件中来精简重复的步骤。
- 一般地，系统级别的测试，例如Cucumber中的场景不应该被方法中硬连接的假行为所“欺骗”。BDD和Cucumber都是关于行为的，而不是实现。所以我们使用其它方法，例如TDD（下一章将介绍）来实现真正的方法来使所有情景通过测试。

自我检查7.8.1. 对或错：你需要实现测试中所有的代码，Cucumber才会通过测试。
◇ 错误。一个悲伤路径能够在快乐路径测试需要的代码没有写完的情况下通过测试。

7.9 显式的和隐式的、命令式的和声明式的情景

现在我们看到了用户故事和Cucumber是怎样使用的，并且已经准备好讲述两个与对比的角度有关的重要的测试主题。

第一个是 **显式的和隐式的需求** 。计划文档模式中形式规划的一大部分是需求，而BDD是由涉众开发的用户故事。使用第1章的术语来讲，它们大多都与验收测试有关。隐式的需求是显式的需求的逻辑结果，大多与第1章中的集成测试有关。一个隐式的需求的例子是在Rotten Potatoes中，默认的是电影应该按上映日期的时间先后顺序排列。好消息是，如果你为显式的和隐式的需求都写了用户故事，那么你你可以用Cucumber来做到一石二鸟——创建验收测试 **和** 集成测试（下一章将展示如何使用另一个工具做单元测试）。

第二个对比角度是： **命令式和声明式的情景** 。图7.3中的示例情景是命令式的，你需要指定一个用户动作的逻辑顺序：填好表单、点击按钮等等。命令式的情景倾向于有复杂的 **When** （当.....时候）条件以及一系列**And**（以及）步骤。这些情景对确保用户界面的细节能匹配客户的期望是非常有用的，很多情景按这种方法写都会很快变得冗长乏味并且不够精简。

为了知道原因，假定我们想要写一个指定电影按字母顺序在电影页面上列出的功能。例如，“Zorro”应该在“Apocalypse Now”之后出现，即使“Zorro”是被首先添加的。简单地表达这种情景是非常令人厌烦的，因为大多数情景下它需要重复我们已有的“添加电影”情景中的代码——这并不是无重复的。

代码文件：ch_bdd/code/imperative_bad_scenario.feature

```
1  Feature: movies should appear in alphabetical order, not added order
2
3  Scenario: view movie list after adding 2 movies (imperative and non-DRY)
4
5    Given I am on the RottenPotatoes home page
6    When I follow "Add new movie"
7    Then I should be on the Create New Movie page
8    When I fill in "Title" with "Zorro"
9    And I select "PG" from "Rating"
10   And I press "Save Changes"
11   Then I should be on the RottenPotatoes home page
12   When I follow "Add new movie"
13   Then I should be on the Create New Movie page
14   When I fill in "Title" with "Apocalypse Now"
15   And I select "R" from "Rating"
16   And I press "Save Changes"
17   Then I should be on the RottenPotatoes home page
18   When I follow "Movie Title"
19   Then I should see "Apocalypse Now" before "Zorro"
```

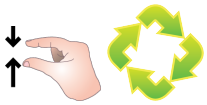
Cucumber应该是关于 **行为** 的，而不是关于实现的，即专注于正在做 **什么** 。在这个写得很差的情景中，只有第18行提到了我们关心的行为。

另一种方法是考虑步骤定义的使用来为你的应用构建 **域语言** （这与正式的 **特定域语言（DSL）不同** ）。一个域语言是不正式的，但它可以使用你的应用的特定的术语和概念，而不是一般的与用户界面实现有关的术语和概念。用域语言书写的步骤一般是更声明式而不是命令式的。域语言描述的是全局的状态而不是如何达到该全局状态的步骤序列，它们对用户界面细节的依赖更少。

一个上述情景声明式的版本可能会是这个样子：

代码文件：ch_bdd/code/declarative_good_scenario.feature

```
1  Feature: movies when added should appear in movie list
2
3  Scenario: view movie list after adding movie (declarative and DRY)
4
5    Given I have added "Zorro" with rating "PG-13"
6    And   I have added "Apocalypse Now" with rating "R"
7    And   I am on the RottenPotatoes home page sorted by title
8    Then  I should see "Apocalypse Now" before "Zorro"
```



代码文件：ch_bdd/code/declarative_steps.rb

```
1  Given /I have added "(.*)" with rating "(.*)"/ do |title, rating|
2    steps %Q{
3      Given I am on the Create New Movie page
4      When  I fill in "Title" with "#{title}"
5      And   I select "#{rating}" from "Rating"
6      And   I press "Save Changes"
7    }
8  end
9
10 Then /I should see "(.*)" before "(.*)" on (.*)/ do |string1, string2, path|
11   step "I am on #{path}"
12   regexp = /#{string1}.*#{string2}/m #  /m means match across newlines
13   page.body.should =~ regexp
14 end
```

图 7.11: 添加这些代码到movie_steps.rb会通过重用已经存在的步骤来创建新的步骤定义用以匹配下图中第5和第6行的声明式情景。步骤（第2行）重用了一系列步骤，步骤（第11行）重用了一个步骤。回忆图3.1中的%Q是给一个字符串用双引号的另一种语法，以及Given（给定），When（当.....时候），Then（然后）等等是增强可读性的同义词（我们将在下一章学习第13行出现的should（应当））

声明式的版本显然是更简洁简单的，更易于维护，更易于理解，因为其中的文本用自然的形式来描述应用的状态："我在Rotten Potatoes的首页看到它们按标题排列。"

好消息是，如图7.11所示，你可以 **重用** 已经存在的命令式步骤来实现情景。这是重用的一个非常有用的形式，随着应用的发展，你将发现你自己不断重用最开始的命令式情景中的步骤，来创建更多简明的、描述性的声明式情景。声明式的、面向域语言的情景将注意力放在被描述的功能上，而不是放在你需要建立和执行测试的低级步骤上。

小结：

- 如果为显式的和隐式的需求都写了用户故事，则可以使用Cucumber来做验收测试和集成测试。声明式的场景更简单，用更少的描述且比命令式的场景更易于维护。
- 随着经验的积累，绝大多数用户故事应该是为你特定应用创建的域语言编写的。可以用步骤定义创建这种域语言。这些用户故事应该较少地关注用户界面细节。特别是一些特定的用户故事，在这些故事中用户界面的细节用来描述有特定的商业价值（客户需求）。

■ 详细阐述：BDD生态系统

描述BDD的最优实践促进BDD的发展已经成为一个强劲的势头，尤其是在重视可测试性、美观以及自动文档化代码的Ruby社区里。好的情景不仅能作为应用设计者的文档，而且能作为可执行验收测试和集成测试。它们应该和代码自身一样美观。例如，来自RailsCasts的免费视频[9]描述了 **情景概要**，一个精简快乐和悲伤路径重复部分的方法，可以根据表单填充的不同而输出不同，和我们之前对快乐和悲伤路径做的对比类似。Cucumber wiki[10]是一个开始学习的好地方，但和所有编程一样，通过不断尝试、犯错误、校正以及美化代码和情景来学习BDD，你会收获最多。

自我检查7.9.1. 对或错：显式的需求一般是通过命令式的情景来定义的，而隐式的需

求一般通过声明式的情景来定义。
◇ 错误。这是两个独立的分类。每一种需求都可以使用任意一种情景来定义。

7.10 计划文档的视角

> 软件工程师都知道（但公众不一定知道），到目前为止最大类的[软件]问题都来自需求的引出、记录以及分析中产生的错误。
>
> ——Daniel Jackson，Martyn Thomas和Lynette Millett（三位作者），**Software for Dependable Systems: Sufficient Evidence?**，2007

回想一下，计划文档方法的期望是使软件工程预算可预测，并和土木工程一样可以做日程安排。神奇的是，用户故事、任务点、效率与计划文档方法的 **七个** 主要任务相一致，它们是：

(1) 需求获取

(2) 需求文档

(3) 成本估算

(4) 日程安排与进度监测

上述在瀑布模型和螺旋模型，以及RUP模型中，应在每个主要阶段的开始完成。需求是随着时间变化的，所以这些内容带来了以下任务：

(5) 需求、成本、日程管理变更

(6) 确保实现与需求功能一致

最后，由于准确的预算估计和时间安排对计划文档过程的成功是至关重要的，我们还有一个在BDD中没有的任务：

(7) 风险分析和管理

这样做的目的是通过预先想象估算预算和日程中的风险，项目可以有计划地避免或克服这些风险。

和我们将在第10章中看到的一样，计划文档过程假定每一个项目有一个经理。整个项目团队可能会参与到需求获取和风险分析中并把它们文档化，而估算成本、制定和维护日程计划，以及决定考虑哪个风险、如何克服或避免这些风险将由项目经理负责。

给项目经理的建议可以来自任何地方，从基于经验提供理论指导和经验法则的参与者，到评估过许多工程并提出预算估算及日程安排准则的研究者。当然，还有一些工具可以帮助我们。尽管有这些很有帮助的建议和工具，第1章的（Johnson 1995, 2009）项目统计表明，40%～50%的项目超出了他们预算和日程安排的1.7～3.0倍，还有20%～30%的项目取消或放弃了——这些都证明了准确做出精确预算和时间表是具有难度的。

现在我们给出这七个任务的简述，使你熟悉计划文档过程中都做了些什么，抛砖引玉以便在以后你会用到它们。这些简述帮助解释了敏捷宣言的灵感。如果你还不清楚如何成功执行这些任务，这可能更多是由于它们固有的困难而不是我们说的太简短。

1、需求获取

和用户故事一样，需求获取与所有涉众的参与有关，需要用到其中的几个技术。第一个是 **访谈**，涉众回答预先设定的问题或者仅仅是非正式的讨论。注意到访谈时的目标是理解开发的人员和组织关系，分析任务是如何 **真的** 被完成的，这不一定与官方说法相同。另一个技术是合作创建 **情景**，这个可以从一个对系统状态最初的假设开始，说明一个系统的好的情况和坏的情况的流程，列出其它的系统中发生的事情，以及在每个情景后系统的状态。与情景和用户故事相关的第三个技术是创建 **用例**，即一个人和一个系统之间用来达到某个目的步骤列表（参见7.1节的详细阐述）。

除上述 **功能性需求** 之外，**非功能性需求** 包括性能目标、可靠性目标等等。

2、需求文档

一旦获取了需求，下一步就是在在 **软件需求说明书**（**SRS**）中将这些需求文档化。图7.12给出了基于IEEE 830-1998标准的SRS的大纲。一个医患管理系统[11]的SRS是一个14页长的文档，但SRS通常有数百页。

作为创建SRS文档的一部分，我们需要检查：

- 有效性——这些需求真的需要吗?
- 一致性——需求彼此之间冲突吗?
- 完整性——所有的需求和约束条件都包括了吗?
- 可行性——需求真能被实现吗?

这四点的测试技术包括让涉众——开发者、客户、测试员等等——来校对这个文档，试着构建一个包括基本功能的原型，生成测试用例来检查需求。

一个项目可能拥有两种类型的SRS很有用：一种高层次的SRS用来给管理和市场部门，一个详细的SRS给项目开发组。前者一般是后者的一个子集。例如，高层次的SRS可能会省略与图7.12中3.2.1.3项相对应的功能性需求部分。

■ **详细阐述：形式化的规范描述语言**

形式化的规范描述语言例如Alloy或Z允许项目经理写可执行的需求，使验证实现变得更简单。不出意外地，所花费的代价是需要写下一个更复杂的文档，并且经常需要读一个更长的需求文档。而它的优势在于精确性，并有可能自动生成测试，甚至使用形式化方法来检验正确性的可能（参见8.10节）。

3、成本估算

项目经理接下来分解SRS到可以实现的任务，然后估算每个任务完成需要的时间。一个建议是，把任务分解成一周左右的是最好的。正如一个有多于7个任务点的用户故事都应当被分割成较小的用户故事，任何估算时间多于8周的任务都应当被进一步分割成更小的任务。

总的工作一般是按人—月来计算的，可能是为了表示对Brook的软件工程经典书籍《人月神话》（Brooks 1995）的敬意。经理们使用薪水和管理费用率来将人—月转换成实际的预算。

成本估算很可能需要做两次：一次用来投标合同，在中标后再做一次。第二次估算是在软件架构设计完成以后，这样就更容易精确地确定所有任务和每个任务的工作量了。

项目经理当然希望第二次估算不比第一次的大，因为第一次的是客户支付的价格。一个建议是把你原始的预算乘以1.3～1.5，用来添加一个安全界限来处理估算不

目录
1. 简介
  1.1 目的
  1.2 范围
  1.3 定义，缩略词，以及缩写
  1.4 参考
  1.5 综述
2. 系统概述
  2.1 产品概述
  2.2 产品功能
  2.3 用户特点
  2.4 约束
  2.5 假设与依赖
3. 具体要求
  3.1 外部接口要求
    3.1.1 用户接口
    3.1.2 硬件接口
    3.1.3 软件接口
    3.1.4 通信接口
  3.2 系统功能
    3.2.1 系统功能1
      3.2.1.1 功能简介/目的
      3.2.1.2 激励/响应序列
      3.2.1.3 相关功能需求
        3.2.1.3.1 功能需求1
        ...
        3.2.1.3.n 功能需求n
    3.2.2 系统功能2
    ...
    3.2.m 系统功能m
  3.3 性能需求
  3.4 设计约束
  3.5 软件系统属性
  3.6 其它需求

图 7.12: IEEE 830-1998标准的软件需求说明书推荐作法目录。我们可以看到在第3节是根据功能来组织的，但是该标准提供了许多其它的方法来组织第3节：根据模式、用户类、对象、激励、功能等级，甚至是混合多种的组织形式

准确或未预料到的事情。另一个方法是做三次估算：一个最好的、一个预期的、一个最坏的，然后用这些信息来做你最好的猜测。

有两种估算的方法，分别是经验性的和数量化的。第一个假定项目经理在公司或工业界有相当多的经验，可以依赖经验来做精确的估计。当项目与该组织曾经成功完成的任务相似的时候，估算的信心会增加。

量化或算法的方法是通过如代码量（LOC）等技术的方法来估算任务的编程的工作量，然后根据人月的生产率测量结果得到每个任务的工作量。项目经理可以从其它地方获得帮助来估计LOC，像效率一样，可以从该组织生产率的历史记录来计算任务的人－月数。

由于软件项目的成本估算保持着如此惨淡的记录，人们付出了相当大的努力来收集完成的项目的信息，寻找预测产出的模型（Boehm and Valerdi 2008）来提升量化的估计方法。下面是复杂的公式：

$$\text{工作量} = \text{组织因素} \times \text{代码长度}^{\text{长度惩罚}} \times \text{产品因素} \tag{7.1}$$

组织因素包括对这类产品的实践，代码长度按前述方法估计。长度惩罚反应了工作量与代码长度不是线性的关系，产品因素包括开发团队对这类产品的经验，可靠性要求，平台复杂度等等。例如来自实际项目的常数中，组织因素是2.94，长度惩罚在1.10～1.24之间，产品因素在0.9～1.4之间。

这些估算是数量化的，他们当然要依赖于项目经理对代码长度、长度惩罚、产品因素的主观判断。上述COCOMO公式的新一代形式要求项目经理选择更多的参数。COCOMO II添加了3个公式来调整以下估算：（1）开发原型、（2）代码重用量计算、（3）一个介绍详细结构的估算。最后一个公式通过添加有5个独立因子的一个归一化积扩展了长度惩罚，并用一个有17个因子的积替代了产品因素。

构造性成本模型（COCOMO）是1981年公式的基础。它在1995年的新一代叫做COCOMO II。

英国计算机协会调研了第1章中提到的超过1000个项目，发现92%的项目经理通过经验而非公式来进行估算（Taylor 2000）。

只有不到20%～30%的项目符合它们的预算和日程安排，剩下的发生了什么情况？另外20%～30%的项目确实被取消或放弃了，剩下的40%～50%的项目即使推迟完成了，对它们的客户仍然是有价值的。客户和供应商一般会谈判一个新合同来，减少有限的一部分功能以使其在接近的日期完成。

PERT 表示计划评价与审查技术（Program Evaluation and Review Technique），由美国海军于1950年发明，并用于其核潜艇项目。

■ **详细阐述：功能点**

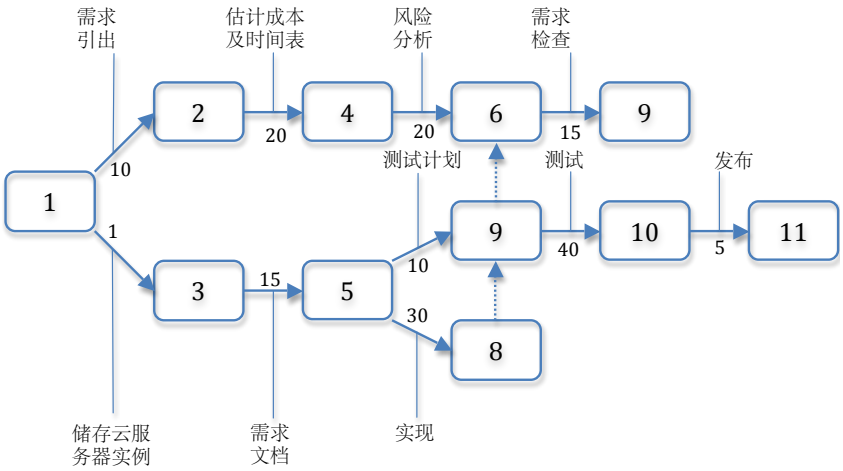
功能点是另一种测量LOC的方法，可以使估算更加准确。这个方法基于函数的输入、输出、外部查询、输入文件、输出文件，以及他们每个的复杂度。对应的生产效率测量就是每人每月完成多少功能点。

4、日程安排与进度监测

当SRS被分解成工作量被估算过的任务时，下一步就是使用日程安排工具来指出哪些任务可以被并行地执行，哪些有依赖性所以必须顺序地执行。这种格式一般是例子与箭头组成的图，像 **PERT（Program Evaluation and Review Technique）图**一样。图7.13给出了一个例子。这样的工具可以确定用来决定项目最小时间的 **关键路径** 。项目经理将图放在一个表上，箭头边标明项目里与之相关的人，然后将任务安排到人。

再次强调，这个过程一般会做两次，一次用来投标合同，另一次是在得到合同后且完成细节设计后进行的。安全界限再次被用到，用来确保第一次时间安排，即客户期望产品发布的日期，不比第二个版本的长。

和计算效率类似，项目经理可以通过比较任务预估的花费和时间，与实际的支出和迄今为止的项目进度，来确定项目是否落后了。一个使所有涉众都清楚项目状



需求蔓延 是开发者使用的术语，用来描述需求随着时间可怕地增加。

图 7.13: 编号的节点代表开发里程碑，带标记线的代表任务，箭头代表依赖关系。有分支线的节点代表并发的任务。另一边数字的线代表这个任务的时间分配。虚线表示不需要资源的依赖关系，所以这些任务没有分配时间

态的方法是在时间表中添加里程碑，使每个人都知道项目是否按日程安排和预算进行。

5、需求、成本、日程变更管理

如本书中多次提到的，由于很多原因，客户很可能会根据项目进展要求改变需求，包括体验过原型后对想要的产品有一个更好的理解，项目市场情况发生改变等等。对项目经理的挑战是随着项目的改变实时地维护需求文档、日程计划、成本预测。这样，变化的文档就和程序一样需要版本控制系统，所以和检查修改过的代码一样，也要检查修改过的文档。

6、确保实现与需求功能相符

敏捷过程联合了许多主要的任务到3个紧密耦合的任务中：用户故事、Cucumber中的验收测试，以及来自BDD/TDD过程的代码。因此，在特定的故事、测试和代码之间的关系就有一点小混乱。

然而，计划文档方法涉及更多没有紧凑耦合的机制。因此，我们需要工具来使项目经理能够检查实现与需求是否吻合。需求功能与实现之间的关系被称为 **需求追踪** 。实现追踪的工具本质上提供设计部分、某个功能的代码实现部分、检查它的代码审查部分以及验证它的测试部分之间的交叉引用。

如果同时有高层次的SRS和细节描述的SRS， **向前追踪** 表示从需求到实现的传统路径，而 **向后追踪** 是从细节描述的需求到高层次的需求的映射。

7、风险分析管理

在提升成本估算和日程安排精确性的工作中，计划文档方法从商学院借用了风险分析的方法。其思想是花费时间来确定预算和日程安排中潜在的风险，一个项目可以通过额外的工作来减小风险，或计划的改变来规避风险。理想情况下，风险识别和管理发生在项目的前三分之一时间。在开发周期后期发现风险则不是一个好兆

头。

风险被分类为技术的、组织的或商业的。一个技术风险的例子可能是选择的关系数据库不能承担项目所要求的任务规模，一个组织风险的例子可能是团队的许多成员不熟悉项目需要使用的J2EE，一个商业风险的例子可能是当项目完成的时候，产品在市场上已经没有了竞争力。

克服上述风险的做法示例分别是：使用一个更具可扩展性的数据库，将团队成员送到J2EE学习班，对已有的产品做竞争力分析，包括他们现在的功能和改进计划。

认清风险的方法是询问每个人他们遇到的最坏的情景。项目经理把它们放进一个“风险表”，给每一项分配从0～100%的发生概率，并且用数字1～4为它们的影响的严重性评分，代表可忽略的、边缘的、关键的和灾难性的。接下来你可以根据发生概率重要性的乘积对风险表进行排序。

有太多潜在的风险以至于项目无法包含全部风险，对应的建议是考虑排前20%的风险，期望上它们代表了预算和时间安排80%的风险。尝试考虑所有的风险可能导致比原来的软件项目更大的工作量！在螺旋和RUP模型里，减少风险是迭代的重要原因。迭代和原型都会减少项目的风险。

7.4节提到了敏捷开发中将询问客户关于项目风险的事情作为成本估算的一部分，但不同的是它常用来决定成本估算的范围而不是成为项目本身的重要一部分。

任务	计划文档模式	敏捷模式
需求文档	软件需求详述如IEEE 830-1998标准	用事故事，Cucumber，任务点，效率
需求获取	访谈，情景，用户，使用案例	
变更管理for 需求，日程计划，以及预算	文档和代码的版本控制	
需求保证功能	功能与测试、检查和代码之间的可追溯性	
日程安排与监测	在项目前期，根据成本估算预签定交货日期，使用PERT图、里程碑来监测进度	
成本估算	在项目前期，根据项目经理的经验或对任务规模和生产率度量的评估来签定价格	通过评估来选择合同中时间和材料的范围
风险管理	在项目前期，确定预算和日程安排的风险，并采取行动来克服或避免它们	

图 7.14：计划文档模式中需求相关的任务与敏捷方法之间的关系

小结：
就像修一座桥一样，软件工程最初的工作量是希望按预期的质量、成本、日程完成软件开发。可能由于仅有不到三分之一的软件项目能按时完成且不超预算，计划文档过程有很多步骤试着达到这个目标。敏捷开发在项目开始并不尝试预测成本和日程，取而代之的是与客户一起工作并频繁地迭代，以及在一段时间要求上达到一致，尽最大努力来达到客户的要求。对用户故事按难度分级，记录每个周期实际完成的任务点从而增加了准确评估的可能性。图7.14展示了这两种不同的理念下，分别需要进行的任务。

自我检查7.10.1. 说出三个计划文档模式中有助于需求获取需求的技术。

◇ 访谈、情景、和用例。

7.11 谬论和陷阱

谬论：如果一个软件项目落后于日程计划，你可以通过给项目中增加人手来赶工。

Fred Brooks的经典书籍 **《人月神话》** 的主题是，增加人手不仅不能带来帮助，而且会使情况更糟。原因有两部分：新来的人熟悉整个项目需要花费一段时间，而且由于项目的规模增大了，花在交流上的时间也增多了，人们实际做工作的时间就减小了。总而言之，所谓的Brooks法则，就是：

> 为已经延期的项目增派人手会让它拖得更久。
>
> ——Fred Brooks, Jr.

陷阱：有些用户搞不清哪些是模型，哪些是已开发功能。

作为一个开发者，这个陷阱对你来说可能是可笑的。但对非技术人员的客户来说，他们很难区分一个美化过的模型和一个开发中的功能。解决方法很简单：例如使用手画的草图的纸—笔方法和故事板来与客户达成一致——这样一来低精度的原型代表已经 **商定的** 功能而不是已经实现的功能，这样就没有歧义了。

陷阱：添加好看的但不能使产品更成功的功能。

软件开发者创造了他们认为很酷但客户抛弃了的代码，这种挫折是敏捷开发的一部分动机。添加一个你认为很强大的功能的诱惑是强大的，但当你的工作被丢弃的时候也是很令人失望的。用户故事帮助所有的涉众把开发任务按优先次序排开，减少开发者偏爱但却浪费工作量的功能的可能性。

陷阱：只用草图而不用故事板。

草图仅是静态的。与Saas应用的交互是随着时间而发生的一系列动作。你和你的客户不仅要在低精度用户界面草图的基本内容上达成一致，还要在与页面交互的时候会发生什么达成一致。给低精度草图"赋予生命"——"好的，你点击这个按钮，这是你所看到的，是你想要的吗？"——在用户故事被转化成测试和代码 **之前**，要走很长一段路来消除误解。

陷阱：单独使用Cucumber作为自动测试的工具而不是将其作为所有涉众共有的中间地带。

如果你看`web_steps.rb`，你会很快发现底层的、命令式的Cucumber步骤，例如"当我点击取消时"仅仅是Capybara的"无界面浏览器"接口的一个简单的包装，你可能会奇怪（和作者的一些学生一样）为什么我们要使用Cucumber。其实Cucumber的实际价值在于创建非技术人员的涉众和开发者能够理解的文档，**并且** 该文档能够作为自动验收测试和集成测试的基础。这就是为什么一个成熟的应用的Cucumber功能和步骤应当和发展成一个适合这个应用的"微型语言"的原因。举个例子，一个给医院护士安排假期日期的应用，应该会频繁使用例如 **轮班**、**工龄**、**假期**、**加班** 等等特定领域术语的情景，而不是集中于用户和软件视图的底层的交互。

陷阱：试着在你真正需要它之前预测到你需要的东西。

行为驱动设计（以及下章的测试驱动开发）的魅力在于，在你写下需要的代码**之前**，你已经写好了测试，然后你需要写能通过测试的代码。这种自上而下的方法更有可能使你的工作变得有效。而这在你预测自己需要什么的时候是很难实现的。这种方法也被称为YAGNI原则——你不需要它（You Ain't Gonna Need It）。

陷阱：小心使用否定式期望。

小心过地度使用 **然后我不应该看到.....** 因为它测试一个否定的条件，你并不能说出输出结果是不是你计划的。你只能说出结果 **不是** 预期的。有许多输出都不匹配，所以它一般都不是一个好的测试。永远只包含肯定的期望，例如 **然后我会看到.....** 来检查结果。

陷阱：小心使用肯定式期望。

即使你使用了 **然后我会看到.....** 的肯定式期望，那你想要看到的结果在页面上显示了多次该怎么办？例如，如果登录用户的名字是Emma，你的情景是检查Jane Austen的书 **Emma** 是否正确添加到了购物车，那么一个情景的步骤 **接下来我会看到"Emma"** 会在购物车不工作的情况下也通过测试。为了避免这种陷阱，使用Capybara的**内部**助手，它能约束匹配器的范围例如让 **我会看到** 能够匹配所给的CSS选择器的元素。就像然后我应当在`''div#shopping_cart''`里看到`''Emma''`，并使用显式的HTML属性`id`或`class`来命名你想在情景中使用的页面元素。Capybara文档[12]列出了所有的匹配器和助手。

7.12 结束语：BDD的利与弊

> 在软件中，我们很少遇到有意义的需求。即使遇到了有意义的需求，唯一衡量成功的标准就是我们的解决方案是否符合客户关于他们问题的多变的想法。
>
> ——Jeff Atwood，**Is Software Development Like Manufacturing?**，2006

图7.15说明了本章介绍的测试工具以及后面章节的测试工具之间的关系。Cucumber允许把用户故事当作功能、情景、步骤以及使用正则表达式的步骤与步骤定义间的匹配。步骤定义调用Cucumber和Capybara的方法。我们需要Capybara是因为我们在写SaaS应用，应用测试需要有一个扮演用户和web浏览器的工具。如果这个应用不是SaaS的，我们就可以直接调用Cucumber中测试应用的方法。

用户故事和BDD的优势在于创造了一个对所有涉众通用的语言，尤其是非技术人员的客户。BDD对客户需求没有被充分理解或频繁改变的项目是完美的，因而它经常也在这些项目上使用。同时用户故事也让分割项目到小的增量或周期变得容易，这更易于评估还有多少工作没有完成。3–5卡片和纸质用户界面原型的使用让非技术的客户能参与到我们的设计和优化中来，提升了客户对软件的满意度。迭代过程驱动着软件开发过程的改进。更进一步地，BDD和Cucumber自然地导致了在编码**之前** 写好测试，将改进验收和开发工作从调试除错变成了测试工作。图7.16是一个谷歌办公室里偏爱测试工作胜过调试除错的海报。

比较用户故事、Cucumber、任务点和效率与计划文档过程，我们会更清晰地发现BDD在敏捷过程中扮演着重要的角色。

(1) 需求获取

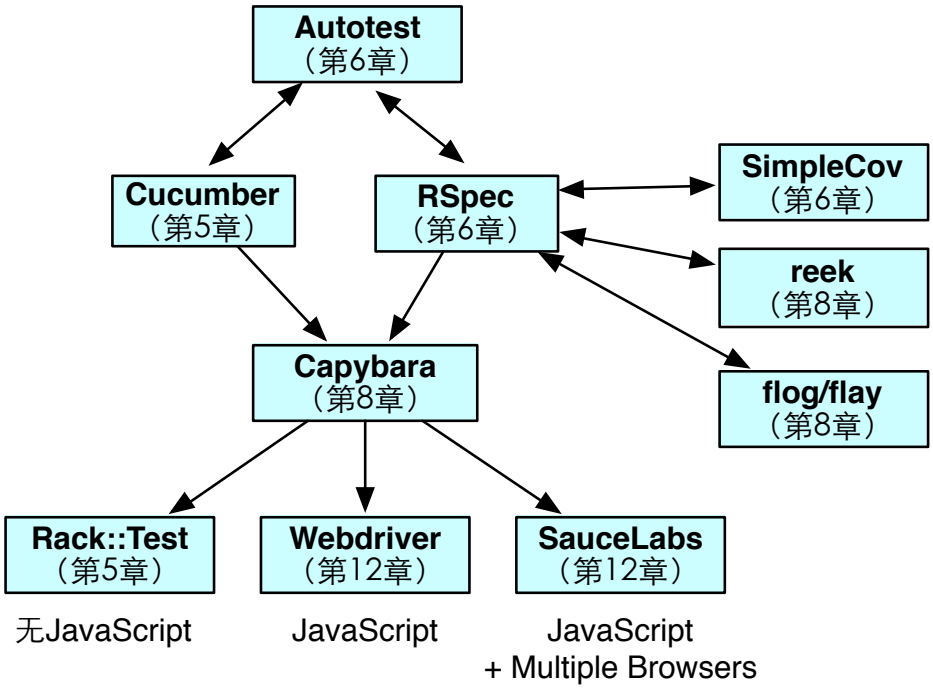


图 7.15：Cucumber、Rspec、Capybara以及本书中其它测试工具和服务的关系。由于应用还没用到JavaScript，我们在本章使用Rack::Test。如果用到了，我们就不得不使用速度慢但更完整的Webdriver。第12章展示了如何用SauceLabs中的服务来替代Webdriver并使用多个浏览器而不是一个来测试你的应用

图 7.16: 谷歌将这些海报放在休息室，以提醒开发者测试的重要性

(2) 需求文档

(3) 验收测试

(4) 功能与实现的可追踪性

(5) 日程安排与项目进度监测

用户故事和BDD不利的方面是，由于一些客户并不想参与到项目中来，在开发过程中持续地与客户接触可能会比较困难或花费过大。这个方法可能并不能扩展到很大型的软件开发工程或安全性要求很高的应用中。计划文档模式可能会更好地适应这两种情形。

BDD另一种潜在的不利的一面是，项目也许可以使客户满意，但不一定能开发出好的软件架构，软件架构是代码维护的重要基础。第11章讨论了设计模式，这些模式也应该是你的软件开发工具箱的一部分。找到适合实际情形的模式，并在需要的时候（参见第9章）重构代码来减小BDD产生糟糕的软件结构的可能。

总而言之，在Ruby社区（一个注重可测试性、漂亮以及自动文档化代码的地方）有很大的势头：对指定的行为写好文档并促进其最佳实现，将其作为应用开发者的文档，作为可执行验收测试和集成测试。Cucumber wiki[13]是一个好的开始学习的地方。

BDD可能并不是开发软件的最自然的方式。最开始，我们会被马上开始写代码所诱惑。然而，一旦你成功学习了BDD，对几乎所有的开发者来说都离不开它了。作者希望给你提示好的工具，虽然有时学起来很吓人的，但从长远来看将会加倍回报你的努力。在未来某时，我们相信你将按BDD的方法来写出漂亮的代码。

7.13 深入学习

- Cucumber维基[14]有文档、教程、样例、截屏、最优方法以及很多关于Cucumber的链接。

- **The Cucumber Book** （Wynne and Hellesoy 2012），是工具开发者及一个最早的尝试者合写的书，包括使用Cucumber的详细信息和例子，极好地讨论了BDD的最优方法，包括额外的Cucumber用法RESTful例如测试服务自动化。
- Ben Mabey[15]（一个核心Cucumber开发者）、Jonas Nicklas[16]以及其他几人，富有表现力地指出声明式和命令式的Cucumber情景的好处。实际上，Cucumber的主要作者Aslak Hellesøy，在2011年10月从Cucumber中故意地移除[17]了**web_steps.rb**（我们在7.6节遇到的），这是我们为什么不得不单独安装`cucumber_rails_training_wheels`工具包来获得例子的原因。

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

B. W. Boehm and R. Valerdi. Achievements and challenges in COCOMO-based software resource estimation. **IEEE Software** , 25(5):74–83, Sept 2008.

F. P. Brooks. **The Mythical Man-Month** . Addison-Wesley, Reading, MA, Anniversary edition, 1995. ISBN 0201835959.

D. Burkes. Personal communication, December 2012.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 1995. URL `http://blog.standishgroup.com/`.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 2009. URL `http://blog.standishgroup.com/`.

A. Taylor. IT projects sink or swim. **BCS Review** , Jan. 2000. URL `http://archive.bcs.org/bulletin/jan00/article1.htm`.

M. Wynne and A. Hellesoy. **The Cucumber Book: Behaviour-Driven Development for Testers and Developers** . Pragmatic Bookshelf, 2012. ISBN 1934356808. URL `http://www.amazon.com/Cucumber-Book-Behaviour-Driven-Development-Developers/dp/1934356808`.

注释

[1] `http://http://www.fandango.com/rss/moviefeed`
[2] `https://developers.google.com/maps/documentation/javascript/tutorial`
[3] `http://www.omdbapi.com`
[4] `http://imdb.com`
[5] `http://www.youtube.com/watch?v=mTYcHg51sWY`
[6] `http://docs.google.com`
[7] `http://campfirenow.com`
[8] `https://github.com/cucumber/cucumber/wiki/cucumber.yml`
[9] `http://railscasts.com/episodes/159-more-on-cucumber`
[10] `http://cukes.info`
[11] `http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/SupportingDocs/MHCPMSCaseStudy.pdf`
[12] `http://rubydoc.info/github/jnicklas/capybara/`
[13] `http://cukes.info`

[14]http://cukes.info
[15]http://benmabey.com/2008/05/19/imperative-vs-declarative-scenarios-in-user-stories.html
[16]http://elabs.se/blog/15-you-re-cuking-it-wrong
[17]http://aslakhellesoy.com/post/11055981222/the-training-wheels-came-off

7.14 建议的项目

练习7.1. 创建让你在RottenPotatoes情景中写出下列步骤的步骤定义：

代码文件：ch_bdd/code/ex/instance_variable_in_cucumber.feature

```
Given the movie "Inception" exists
And it has 5 reviews
And its average review score is 3.5
```

提示： Cucumber步骤定义中的实例变量是与情景相关联的，而不是步骤。

练习7.2. 假设在RottenPotatoes中，我们使用填写空白的表单的方式来代替选择评级和发行日期的方式。首先，对图7.3中的情景做合适的改变。列出`features/cucumber/web_steps.rb`中的步骤定义，使Cucumber现在能在测试中调用这些新的步骤。

练习7.3. 给图7.3中的功能添加一个悲伤路径的情景，该路径对应于用户将标题段留空的时会发生什么。

练习7.4. 写出一组后台的步骤，向RottenPotatoes网站添加一些电影。

练习7.5. 创建一个展示RottenPotatoes应用当前行为的低精度的原型。

练习7.6. 想出一个你想要添加到RottenPotatoes的功能，画出一个故事板来展示如何实现和使用它。

练习7.7. 创建一组如图7.11中的步骤，可以用来实现以下步骤：

代码文件：ch_bdd/code/exercise_step.rb

```
When / I delete the movie: "(.*)"/ do |title|
```

练习7.8. 使用Cucumber和Mechanize来给一个已经存在但没有测试装置的SaaS应用创建综合测试或验收测试。

练习7.9. 创建一个Cucumber步骤定义，使你能够检查一个页面上某字符串的多个实例，例如然后我应当看到`‘‘Hurrah’’`三次（`Then I should see ’’Hurrah’’ 3 times`）。

提示： 考虑你想将页面文档根据匹配的字符串分割成几大块时会发生什么。

练习7.10. 改进前一个练习的步骤定义，使那个匹配可以被限定在某个CSS选择器中，就像然后我应当在`‘‘div#congratulations’’`中看到3次`‘‘Hurrah’’`（`Then I should see ’’Hurrah’’ 3 times within ’’div#congratulations’’`）。

练习7.11. 将网上病人管理系统[1]中的两个段落转换成Connextra格式的用户故事。它们是SMART的吗？哪一个是功能性的，哪一个是非功能性的？

练习7.12. 将Rotten Potatoes的用户故事转换成一个软件需求说明书文档。用SRS方式表达，困难在什么地方？

练习7.13. 用一个随意指定的标准，例如时间，来估计计划文档过程的工作量，并与像Pivotal Tracker这样的工具记录的实际工作量比较。注意：旁边带有图标标记的项目来自ACM/IEEE 2013软件工程标准（ACM 2013）。

练习7.14. 描述需求获取过程中基础的难点和常见的方法。

练习7.15. 向前追溯和向后追溯的区别是什么？解释它们在需求确认过程中的扮演的角色。

练习7.16. 列出一些软件风险的例子。

练习7.17. 描述软件系统风险的不同分类。

练习7.18. 描述计划文档生命周期中风险的影响。

8 软件测试：测试驱动开发

Donald Knuth（1938——）最杰出的计算机科学家之一，1974年因在算法分析和编程语言设计的贡献，尤其是在他的多卷《计算机编程艺术》（The Art of Computer Programming）中的贡献获图灵奖。许多人把这个系列的书当作算法分析的权威参考。Knuth对找出他书中错误的"悬赏检查"是计算机科学家之间最有价值的奖品之一。Knuth还发明了被广泛应用的TEX排版系统，本书就是用它排版的。

一个重要的教训，可能就是"编写软件是很难的"。...TEX和METAFONT被证明比我做过的所有其它事情（例如证明定理或写书）还要难很多。创作优秀的软件需要比做其它事情有明显更高的准确度标准，它比其它的智力工作需要更长的注意力跨度。

——Donald Knuth, Keynote address to 11th World Computer Congress, 1989

8.1 背景：RESTful应用程序接口和Ruby工具包 240
8.2 FIRST、TDD、以及RSpec入门 241
8.3 TDD周期：红色——绿色——重构 244
8.4 更多的控制规则和重构 248
8.5 夹具和工厂 252
8.6 面向模型的TDD 254
8.7 为Internet建立桩 258
8.8 覆盖率概念、单元测试与集成测试 263
8.9 其它测试方法和术语 265
8.10 计划文档的视角 267
8.11 谬论和陷阱 270
8.12 结束语：TDD与传统的调试方法 271
8.13 深入学习 272
8.14 建议的项目 273

重要概念

本章的重要概念有测试编写、测试覆盖和测试级别。

创建好的测试的五个原则是快速、独立、可重复、自检验和及时性（FIRST）。我们使用 **模拟对象** 和 **桩** 来帮助维持测试的快速性，并且与其它类的行为保持独立。它们是 **嫁接** 的例子，通过嫁接可以在测试过程中改变程序的行为而不改变源代码本身。

敏捷生命周期中的测试遵循 **测试驱动开发** （ **TDD** ），有以下步骤：

- 从验收测试和集成测试开始的做法源于 **用户故事** ，编写测试你认为代码应该有但不存在的失败的 **单元测试** 。我们将使用RSpec工具来完成这一步骤。
- 写出只能通过一个测试的代码，并在继续下一个测试之前寻找 **重构** 代码的可能。由于失败的测试报告为红色，而通过的结果为绿色，所以这个序列被叫做 **红色—绿色—重构** 。
- 在测试中使用 **模拟** 和 **桩** 来把正在测试的代码行为与它依赖的类或方法的行为隔离开来。
- 各种各样的 **代码覆盖** 指标能帮助你决定哪一部分代码需要更多的测试。

对计划文档生命周期来说，可使用完全相同的概念但使用顺序甚至人员都有很多不同：

- 项目经理根据SRS来分配编程任务，所以 **单元测试** 在编码之后开始。 **质量保障** （ **QA** ）测试员接管开发者的工作来进行更高层次的测试。
- **自顶而下** 、 **自底而上** 以及 **两端同时开始** （三明治式）是把生成的代码结合在一起来执行 **集成测试** 的方法。测试计划和结果是文档化的，比如依据IEEE 829-2008标准。
- 在集成测试之后，发布给客户之前，QA小组执行一个 **系统测试** 。测试在达到一个特定的覆盖水平时停止，例如"覆盖到了95%的语句"。
- 对于小规模且重要的软件，一个替代测试的方法是 **形式化方法** 。对通过定理证明程序自动验证的或穷举的状态搜索得到的正确的程序行为，形式化方法会使用系统正确行为的形式化的描述，这两种方法得到正确的程序行为都会超出传统测试所能做到的。

它们测试的方法完全不同于敏捷和计划文档生命周期：当开始撰写测试时，测试的顺序甚至是谁来做测试都已经确定好了。

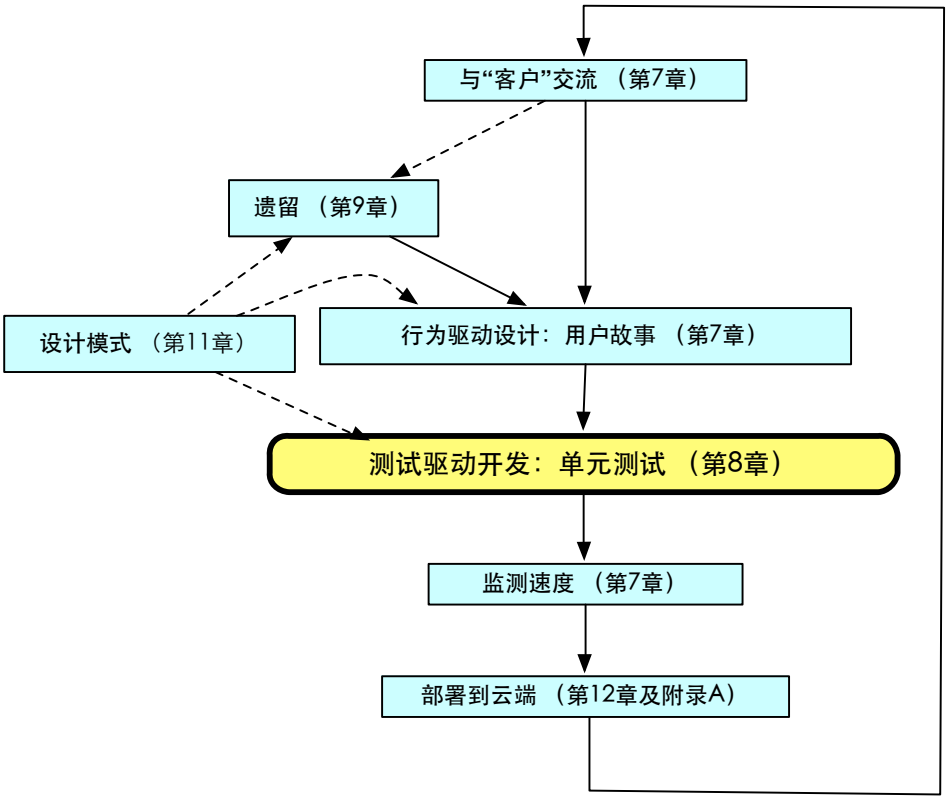


图 8.1: 敏捷软件生命周期及其与本书各章节的关系。本章着重讲驱动开发的一部分——单元测试

方法还是函数？ 根据面向对象编程（object-oriented programming）的术语，我们使用 **方法** 来命名某个类实现一种行为的代码片断，不管它更像一个返回数值的函数还是更像一个引起其它效果的过程。这样一段代码的历史术语也包括 **方法**、**程序**、**子程序** 和 **辅程序**。

8.1 背景：RESTful应用程序接口和Ruby工具包

第1章介绍了敏捷生命周期和区分软件保障的两个方面：验证（validation）（“你做了正确的事情吗”）和确认（verification）（“你把事情做对了吗”）。本章将重点关注确认——把事情做正确——通过把软件测试作为敏捷生命周期的一部分。图8.1强调了本章中涵盖的敏捷生命周期的部分。

尽管测试只是确认的一种技术，但是因为它的角色经常被误解，结果导致不能获得和软件生命周期中其它部分一样足够的重视，所以我们重点关注它。另外，我们将看到，以测试为中心的观点进行软件开发往往能提升软件的可读性和可维护性。换句话说，**可测试的代码常常是好的代码，反之亦然。**

从第7章开始，我们将继续Rotten Potatoes的一个新的功能，使电影信息能自动从公开电影数据库（TMDb）[2]导入。本章将开发必要的方法来完成这个功能。

和许多SaaS应用一样，TMDb被设计成面向服务架构的一部分：它有一组允许外部应用程序访问的 **API**（Application Programming Interface），并不仅仅是人进行网页浏览，来使用它的功能。和视频8.1.1显示的一样，TMDb的API是RESTful（Representational State Transfer）的，允许外部应用程序发起的每个请求都是完全自我包含的，如第2章。

视频8.1.1: 使用TMDb API.
光盘文件：ch_tdd/mov/tmdb_api_intro.mp4
我们通过构建一个RESTful的统一资源标识符（URI）来使用TMDb的API，用来实现像"通用标题来搜索电影"或是"检索一个给定电影的详细信息"等的适当功能。为了防止滥用并独立跟踪每个用户使用的API，每一个开发者必须首先从TMDb网站申请一个自己的 **API密钥** 。不包含有效的API密钥的URI申请不会被正确接受，取而代之的是会返回错误信息。对于包含有效API密钥的URI请求，TMDb返回一个用URI编码的包含请求结果的XML文档。这个流程——构造一个包含API密钥的RESTful的URI，接收XML应答——是与外部服务交互的常见模式。

通常情况下，从Rotten Potatoes中调用这样的API需要使用Ruby标准库中的**URI**类来构造包含API密钥的请求URI，使用**Net::HTTP**类来将请求发给`api.themoviedb.org`，并使用**Nokogiri**[3]之类的XML解析库来解析答复。但有时还可以通过使用已有的包来使解析更效率，就像在下面这个情况下做的一样。有一两个由用户贡献的基于TMDb的RESTful API的封装`ruby-tmdb`工具包。这在TMDb API的文档中被提到过，在谷歌搜索`tmdb ruby library`时它也会被排在最好结果的位置上。视频8.1.2展示了如何使用它。

视频8.1.2: 基于`ruby-tmdb`工具包的TMDb API简化用法.
光盘文件：ch_tdd/mov/ruby-tmdb.mp4
并不是所有的RESTful API都有一个对应的Ruby库，但像TMDb这样有对应Ruby库的程序库能把API细节隐藏在一些简单的Ruby方法后面。通过第3章介绍的Gemfile和Bundle安装的`ruby-tmdb`工具包，能管理URI构建并解析XML结果到哈希、字符串、日期等Ruby对象。

小结：
TMDb（电影数据库）有一个面向对象的API，能够通过HTTP请求发送正确构造的HTTP URI并返回HTTP响应，该响应主体是包含响应数据的XML文档。可以用一个开源的Ruby工具包来方便地构造合适的请求并解析XML响应，这可以用来替代直接处理URI和XML文档。

自我检查8.1.1. 对或错：为了从另一种例如Java的语言调用TMDb API，需要一个与`ruby-tmdb`工具包一样的Java程序库。
◇ 错误。API包含一系列HTTP请求和XML响应，所以只要能通过TCP/IP协议发送和接收数据，并有解析字符串（XML响应）的能力，就能使用API，并不需要一个特殊的程序库。

8.2 FIRST、TDD、以及RSpec入门

开发者"把代码抛给墙那边的 **质量保障人员（QA）** "，这并不是SaaS应用通常的做法，QA工程师手工测试软件并来填写故障报告所花费的时间也是也反映了此种情况。实际上，质量保障的主要考虑是分清组之间的责任而不是考虑一个好的过程的结果，这对SaaS应用是被当作过时的。如今，SaaS开发者承担比测试他们自己的代码以及参与到评审多很多的责任，QA的责任在很大程度上已经变成如何改善测试工具基础架构，帮助开发者写出更具可测试性的代码，确认用户报告的错误是

可重现的，就像我们将在第10章讨论的那样。我们将会看到，敏捷生命周期同样期望QA小组的成员是开发者。

如今的测试工作有许多都是自动的。自动测试并不是说测试是自动为你创建的，而是测试是能自我检查的：即测试代码自身能决定被测试的代码是否工作，不需要人来手工检查测试输出或与软件进行交互。创建一个好的测试有以下五个原则，可以用首字母缩写FIRST来总结：**F**ast（快速）、**I**ndependent（独立）、**R**epeatable（可重复）、**S**elf-checking（自校验）、**T**imely（及时），高度自动化测试工具的关键在于支持这五个原则。

- **F**ast（快速）：它应该能简单和快速地运行与当前的编码任务相关的测试用例，来避免妨碍你连贯的思路。我们将用名为Autotest的Ruby工具为此提供帮助。
- **I**ndependent（独立）：任何测试都不应依赖于由别的测试引出的先决条件，这样我们才能优先运行覆盖最近改动代码的测试子集。
- **R**epeatable（可重复）：测试行为应当不依赖于外部因素，例如改变日期或改变了它们的值会破坏测试的“特殊常数”。就像19世纪60年代好多程序在2000年到来时[4]发生的那样。
- **S**elf-checking（自校验）：每一个测试都应该是能够自己决定测试是通过还是失败，不依赖于人工来检查它的输出。
- **T**imely（及时）：测试应该在代码被测试的时候创建或更新。我们将看到，测试驱动开发中测试是在编码的 **前一刻** 才完成的。

千年虫程序错误在发作
这张照片是2000年1月3日拍摄的。（维基共享资源）

测试驱动开发（TDD）提倡使用测试来 **驱动** 代码开发。当TDD被用来创建新的代码，就像我们在本章所做的一样，它有时会被叫做 **测试优先开发**，因为测试早于任何被测试的代码存在。当TDD被用来扩展或修改遗留代码时，就像第9章讨论的那样，可能会为已有的代码创建新的测试。本章主要探索TDD，我们将会展示Ruby工具是如何支持TDD和FIRST的。尽管第一次使用TDD时可能会感觉它很奇怪，但它会产生通过良好测试的代码，使之更加模块化且比大多数其它方法开发的代码更易于阅读。虽然TDD显然不是唯一能达到这些目标的方法，但如果TDD被正确使用的话，就不容易产生有严重缺陷的代码。

我们将用RSpec来写测试，一种用来测试Ruby代码的 **特定域语言**（DSL）。一个DSL是一种小规模的编程语言，被设计用来简化处理一个单一区域（领域）内的问题而不考虑普适性。你已经看到了一些像描述网页的DSL等 **外部的**（独立的）DSL。RSpec是一种 **内部的** 或 **嵌入的** DSL：Rspec代码就是Ruby代码，但是利用了Ruby特性及语法的优势以便组成一个重点集中于测试工作的“小型语言”。正则表达式是Ruby中嵌入的另一种内部DSL的例子。

RSpec还可以被用来做集成测试，但是我们更喜欢Cucumber，因为它能促进客户和自动化验收之间的对话，就像集成测试一样。

Rspec的功能帮助我们获得代码应该怎样运行的 **期望结果**。这样的测试是可执行的规范或用Ruby写的“规则（specs）”，因此叫做RSpec。如何在获得被测试的代码之前得到测试的期望结果？答案会令人惊讶：编写检测 **我们希望有的代码** 的测试。这种做法迫使我们不仅要思考代码要做什么，还要思考代码应该怎样被它的调用者和协作者（其它和它一起工作的代码片断）使用。第7章我们在Cucumber情景步骤 **以及我单击“搜索TMDb”** 里这样做了：当修改电影列表视图（`views/movies/index.html.haml`）以包含一个“搜索TMDb”的按钮时，我们选择了**search_tmdb**作为响应这个点击的尚不存在控制器方法名称。当然，由于方法**MoviesController#search_tmdb**不存在，当你试着真正运行这个情景的时

代码文件：ch_tdd/code/spec/movies_controller_spec.1.rb

```
1 require 'spec_helper'
2
3 describe MoviesController do
4   describe 'searching TMDb' do
5     it 'should call the model method that performs TMDb search'
6     it 'should select the Search Results template for rendering'
7     it 'should make the TMDb search results available to that template'
8   end
9 end
```

图 8.2: MoviesController#search_tmdb的RSpec例子的框架。根据惯例先于配置的原则，app/controllers/movies_controller.rb的规则应该在spec/controllers/movies_controller_spec.rb等之中（请使用剪贴板来复制粘贴这些代码）

Bar#foo 是一种通知的Ruby符号，代表类**Bar**中的 **实例** 方法**foo**。这个符号**Bar.foo**代表 **类** 方法**foo**。

候这个Cucumber步骤会失败（显示红色）。本章的后续部分将会使用TDD来开发**search_tmdb**方法。

在MVC架构下，控制器的工作是响应用户交互，调用合适的模型方法来检索或管理任何必要的数据，以及生成一个合适的视图。我们可能因此将还不存在的控制器方法 **期望的** 行为描述成下面这样：

- 它应当调用一个模型方法将用户输入的搜索词条传给TMDb来执行搜索。
- 它应当选择搜索结果的HTML视图（用Rails语法，搜索结果 **模板** ）用来显示。
- 它应当使TMDb搜索结果在模板中可用。

注意这个需求列表中没有哪个方法或模板是已经存在的！这是TDD的本质：写出期望行为（规则）的一个具体而简明的列表，并用它来驱动方法与模板的创建。

图8.2展示了我们该如何用RSpec来表达这些需求。和第3章讲的一样，我们鼓励通过实践来学习。在创建这些文件之前，你需要建立Rotten Potatoes使用RSepc来进行测试，这需要以下三步：

你可以看看`spec/spec_helper.rb`里面有什么。

(1) 在Gem文件的**group :test**区块中，添加**工具包“rspec-rails”**

(2) 每一次更新Gem文件时，运行`bundle install--without production`

(3) 在Rotten Potatoes的应用根目录中，运行`rails generate rspec:install`来建立RSpec需要的文件和目录。

现在你已经准备好创建如图8.2所示的文件`spec/controllers/movies_controller_spec.rb`了。第一行加载了一些将会被所有RSpec测试使用的助手方法。通常下，Rails应用程序的所有规则文件的第1行都是这个。第3行说明接下来的规则**描述**了**MoviesController**类的行为。因为这个类有一些方法，第4行说明这是第一组规则，这些规则描述了搜索TMDb的方法的行为。正如你看到的，**描述**后面可以跟一个类名或一个描述性的文档字符串。

下面三行是 **样例** 的占位文本，一个RSpec术语，用于一小段测试**search_tmdb**方法 **某个** 特定行为的代码。我们还没有写测试代码，但下一个视频显示我们不仅仅能用`rspec`命令来执行这些测试框架，更重要的，我们可以使用`autotest`工具来自动运行它们。这个自动配置帮助提高生产效率，因为我们不需要在写代码和测试代码之间切换注意力。本章余下部分将假定`autotest`在运行，并且无论何时添加测试或应用代码，你都会马上从RSpec得到反馈。下一节将使用TDD创建第一个测试。

1、在你写任何新的代码前，写一个这段代码行为 **应当** 有的在 **某个** 方面的测试。由于被测试的代码还不存在，写下测试迫使你思考想要代码如何表现，以及与可能存在的协作者如何交互。我们把这种方法叫做“检测你想要的代码”。
2、**红色**步骤：运行测试，并确认它失败了，因为你还没有实现能让它通过测试的代码。
3、**绿色**步骤：写出能使 **这个** 测试通过但不损坏其它已有的测试的 **尽可能简单的** 代码。
4、**重构**步骤：寻找 **重构** 代码或测试的机会——改变代码的结构来消除冗余、重复、或其它可能由添加新代码而引起的丑陋现象。测试能保证重构不会产生错误。
5、重复以上步骤直到所有必须通过情景步骤的行为都完成。

图 8.3: 测试驱动开发（TDD）循环又被称为红色—绿色—重构，这是由于它第2∼4步的框架。最后一步假设你在开发能完成一个情景的代码，就像你在第7章里开始做的那样

ruby-debug和autotest
为了让`autotest`使用第4介绍的交互调试程序，添加**`require 'ruby-debug'`**到`spec/spec_helper.rb`，并在任何你想让程序动作停下来的地方插入**debugger**命令调用。

视频8.2.1: 运行空的测试框架和使用`autotest`自动运行.
光盘文件：ch_tdd/mov/tdd1.mp4
当运行`RSpec`命令时，不包含任何代码的样例（**it**子句）被显示为黄色，意味着“未解决”。你也可以明确地标记某个样例为**未解决**，并提供一个它为什么未解决的描述。与每次添加或改变代码来手工运行**spec**相比，可以使用`autotest`命令，无论何时改变一个规则文件或代码文件，适当的规则都自动重新运行。

小结：

- 好的测试应该满足**F**快速、**I**独立、**R**可重复、**S**自校验和**T**及时（FIRST）。
- RSpec是Ruby编写的用来写测试实例的特定域语言。按照惯例先于配置的原则决定了与一个给定的类文件相对应的规则文件应该放在哪里。
- 在一个规则文件中每个单一的由**it**方法引出的 **样例**，测试一个方法的单一的行为。根据它们测试行为的集合，**描述**将样例分层次地分类。

自我检查8.2.1. 一个单独的RSpec测试实例或 **例子** 可以由关键词____来引入。一组相关的例子可以由关键词____来引入，它可以被嵌套地组织例子到一个分层次的文件中。
◇ **it**; **describe**

自我检查8.2.2. 由于RSpec使用惯例先于配置的方式将测试匹配到类，应该将`app/models/movie.rb`的测试放到文件____中。
◇ `spec/models/movie_spec.rb`。

8.3 TDD周期：红色——绿色——重构

图8.3描述了基本的TDD方法。你或许认为我们在完成那些实例的代码之前写下图8.2中的三个测试实例违反了TDD方法论，但实际上，创建你将要编写的测试使用的**it**模块并没有任何问题。现在，是时候开始工作并继续进行测试了。

代码文件：ch_tdd/code/spec/movies_controller_spec.2.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      it 'should call the model method that performs TMDb search' do
6        post :search_tmdb, {:search_terms => 'hardware'}
7      end
8      it 'should select the Search Results template for rendering'
9      it 'should make the TMDb search results available to that template'
10   end
11 end
```

图 8.4: 填写第一个规则文件。其中第8行有一个"空的"it充当还没写出来的例子的占位符，第5～行有一个包含do...end块的it是真实的测试样例

图8.2中的第一个例子（测试实例）指出，**search_tmdb**方法应当调用一个模型方法来执行TMDb搜索，将用户输入的关键词传给那个方法。在第7章中，我们通过添加一个HTML表单更新了Rotten Potatoes的**索引**视图，该表单包括一个名为**search_terms**的输入区域供用户来填写，并提交给**MoviesController#search_tmdb**方法来处理。我们的测试样例因此需要模拟当用户在**search_terms**区域输入了一些内容并提交表格时发生的情况。我们知道，在一个Rails应用里**参数**哈希是随着表单中的数据提交自动填充的，所以控制器方法能检查它。幸运的是，RSpec提供了一个**发送**方法来模拟传递一个表单给控制器的动作：第一个参数是接收提交的动作的名称（控制器方法），第二个参数是传给控制器动作的**参数**哈希值。我们现在能够写下规则文件的第一行了，就像图8.4展示的那样。下一个视频说明，无论如何，我们都必须克服一系列障碍以便将到达红色—绿色——重构循环的红色阶段。

视频8.3.1: 开发第一个例子需要添加一个空的控制器方法并创建一个空的视图.
光盘文件：ch_tdd/mov/tdd2.mp4
为了解决RSpec的错误，首先需要创建一个空的控制器方法及其对应的路线，这样（用户提交的）动作才能有去的地方。需要创建一个空的视图，使控制器动作有可以渲染的东西。那一行测试代码要求我们确保新的控制器方法以及它最终渲染的视图有能有正确的名字和与之匹配的路径。

这时候RSpec报告了我们第一个例子为绿色。但这并不是真的准确，因为例子本身是不完全的：我们还没有真正检查**search_tmdb**是否调用了一个模型方法来搜索TMDb，就像规则文件要求的那样（我们故意这样做，以阐明使第一个规则运行起来的必要结构。一般情况下，由于每一条规则都比较短，你应该在重新运行测试之前完成规则）。

在没有模型方法存在的情况下，应该怎样检查**search_tmdb**调用了一个模型方法？重申一下，你是在为 **我们希望有的代码** 的行为写测试，就像图8.3中第1步指导的。现在假装有一个模型方法能完成我们想要的事情。在这种情况下，我们可能想传给这个方法一个字符串并返回与该字符串匹配的TMDb搜索结果**Movie**类对象的集合。 **如果** 这个方法存在，控制器方法就可以这样调用：

我们想要的代码是一个类的成员方法，因为在TMDb中寻找电影是一个与电影相关的一般行为，并不是类**Movie**的一个特定实例。

代码文件：ch_tdd/code/call_model_method.rb

```
1  @movies = Movie.find_in_tmdb(params[:search_terms])
```

图8.5展示了一个测试样例中断言这样的调用会发生的代码。在这种情况下，

代码文件：ch_tdd/code/spec/movies_controller_spec.3.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      it 'should call the model method that performs TMDb search' do
6        fake_results = [mock('movie1'), mock('movie2')]
7        Movie.should_receive(:find_in_tmdb).with('hardware').
8          and_return(fake_results)
9        post :search_tmdb, {:search_terms => 'hardware'}
10     end
11     it 'should select the Search Results template for rendering'
12     it 'should make the TMDb search results available to that template'
13   end
14 end
```

图 8.5：通过确认该控制器方法会调用Movie模型中我们想要的代码来完成例子。列表中第5～10行替代了图8.4中的第～7行

我们要测试的代码—— **主体代码** ——是**search_tmdb**。但是，我们要测试的一些行为看起来像是依赖于方法**find_in_tmdb**。由于**find_in_tmdb**的代码我们目前还没有，所以第6～8行的目的就是“假装”我们完成它时它表现出的行为。第6行使用RSpec的**mock**方法来创建一个包含两个“测试替身”**Movie**的数组。特别地，一个真正的**Movie**对象会对如**title**和**rating**之类的方法进行响应，但一个测试替身的对象会在你调用它的任何方法时抛出异常。面对这个事实，我们为什么还要用替身呢？原因在于这样可以把这些规则从**Movie**类的行为孤立出来，因为**Movie**类自身也可能会有错误。模拟就像木偶一样，它的行为可以被完全控制，允许我们将单元测试从其它协作类中孤立出来，使测试**独立**（FIRST中的I）。

实际上**mock**（**模拟**）的别名就是**double**（**替身**）。为了说明清晰，在需要假装对象来做事情时使用**mock**（**模拟**），在仅仅使用一个替身的时候使用**double**（**替身**）。

回到图8.5，第6和7行表达了**Movie**应该接受一个**find_in_tmdb**的方法调用，以及该方法应该接收一个参数**'hardware'**的期望。RSpec将会打开**Movie**类并定义一个名为**find_in_tmdb**的类方法，其目的是记录它是否被调用。如果这样的话，说明参数传递正确。极为关键的一点是，**如果Movie类中已经存在一个同名的方法，它将被这个方法桩临时“覆盖”掉**。这就是为什么我们的样例并不关心在“实际的”**find_in_tmdb**方法中写的是什么：因为它根本不会被调用！

使用**should_receive**为了临时覆盖一个“实际的”方法，它的测试目的是使用**嫁接**（seam）的例子：“一个你不需要编辑却可以改变程序的行为的地方。”（Feathers 2004）在这个样例中，**should_receive**通过在适当位置重载方法创建了一个嫁接，我们并不需要去编辑包含原始方法的文件（尽管此例中，原始文件并不存在）。嫁接在添加新的代码到应用的时候也很重要，但在本章的后续部分我们将看到更多的嫁接的测试例子。嫁接有用是因为它们让我们打破了一段代码及其与其协作者的依赖性，允许代码的协作部分在测试中有和在实际生活中不一样的行为。

第8行（接着第7行）指定了**find_in_tmdb**应当返回我们在第6行创建的替身的集合。这是完成了“我们想要的代码”的假象：调用一个并不存在的方法，并提供了如果它存在时我们期望的结果！如果省略**with**（包含），RSpec将仍然检查**find_in_tmdb**是否被调用了，但不会检查参数是不是期望的。如果省略了**and_return**，这个假的方法调用调用将会返回**nil**而不是我们选择的值。在任何情况下，每一个样例执行后，RSpec都会执行一个**拆除**步骤来将类复原到他们原来的状态，所以如果你想要在其它例子中执行相同的假的动作，我们需要在每一个样例里都指定它（尽管我们很快会看到精简这样重复的方法）。这种自动拆除是保持测试**独立**的另一个重要部分。

技术上来说，在这个样例中，省略**and_return**也是可以的，因为这个例子并不检查返回值，但我们保留它是为了起说明作用。

这个新版本的测试失败了，因为我们创建了**search_tmdb**需要调

用**find_in_tmdb**的期望，但**search_tmdb**还没有写。因此，从红色到绿色的最后一步就是给**search_tmdb**添加上足够的代码来通过这个测试。我们之所以说测试 **驱动着** 代码的创作，正是因为添加测试会导致失败，而我们必须通过在模型中添加新代码来解决这个问题。由于这个特殊的例子仅仅测试**find_in_tmdb**的方法调用，它足够按照我们头脑中"我们想要的代码"来给**search_tmdb**添加一行代码：

代码文件：ch_tdd/code/finish_controller_method.rb

```
1 | @movies = Movie.find_in_tmdb(params[:search_terms])
```

如果你刚刚接触TDD，这里有许多需要理解，尤其是使用Rails这样强有力的框架来测试一个应用。不用担心——现在你已经接触了主要概念，下一阶段的规则将变得更容易理解。熟悉接受这个系统需要花费一点时间，但我们已经发现学习这些的回报是相当值得的。请阅读下面的总结稍作休息，在继续学习之前复习一下本节的这些概念。

小结：

- TDD的红色—绿色—重构周期从写一个测试开始，因为它测试的 **主体代码** 还不存在，所以会失败（红色）。然后，添加最少的必要代码来使它通过这个测试例子（绿色）。
- 嫁接方法能让你改变应用特定位置的行为但不用编辑那里的代码。典型的测试装置经常通过使用**模拟**或称**替身**来创建嫁接，用以创建测试的替身对象，或者使用**should_receive...and_return**来将协作方法 **完全替换**（替代并控制返回值）。模拟和桩都是嫁接方法，通过将被测试代码的行为孤立于与之相关的代码，来帮助提升可测试性。
- 每一个例子都要建立先决条件，执行测试主体代码，对结果做出一定的断言。例如**should**（应当）、**should_not**（不应当）、**should_receive**（应当收到）、以及**with**（包含）等断言能够实现测试的**自校验（S）**，免除了程序员来检查测试结果的需要。
- 在每一个测试之后，一个自动的拆除步骤会将模拟和桩毁掉，并复原所有的期望，所以测试仍然保持**独立性（I）**。

■ **详细阐述：其它语言里的嫁接**

在像C这样的非面向对象的语言中，很难创建嫁接。由于所有的方法调用都在链接时确定，通常开发者创建一个包含“假的”（测试替身）版本的所需要的方法的库，然后小心地控制库连接顺序来确保测试替身版本被使用。类似地，由于C的数据结构是直接从内存中读取而不是通过存取方法，数据结构嫁接（模拟）通常是同**#ifdef TESTING**这样的预编译指令来创建的，用来在测试用途和产品用途时以不同方式编译代码。

在像Java这样的静态类型的面向对象语言中，由于方法调用是在运行时确定的，一种创建嫁接的方法是当编译测试装置时，创建测试类的子类并重载确定的方法。模拟对象也是可能的，虽然模拟对象必须满足完全实现编译器对“真正的”对象的期望，即便模拟仅仅做了实际对象会做的一小部分工作。JMock网站[5]演示了在一些Java中插入测试模拟的例子。

在Ruby这样的动态的面向对象语言中，你可以在运行时更新类，我们几乎可以在任何时间、任何地方来创建模拟。RSpec拓展了这个功能，允许我们仅仅创建每一个测试需要的特定的模拟和桩，使测试更易于编写。

自我检查8.3.1. 在图8.5中，为什么第7行的**should_receive**期望要在第9行的**post**动作之前？

◇ 期望需要来建立一个可以被监视且能确保被调用**find_in_tmdb**的测试替身。由于**发送**动作最终会导致调用**find_in_tmdb**方法，这个替身必须在**发送**发生之前建立。否则的话，实际的**find_in_tmdb**方法将被调用（在本例中，**find_in_tmdb**根本还不存在，因此测试会失败。）

8.4 更多的控制规则和重构

回到图8.2中列出的原始的规则框架来，第6行说**search_tmdb**应当选择“搜索结果”视图来渲染。当然，这个视图还不存在，但和上面写的第一个例子一样，这并不能阻止我们。

这真的是必须的吗？
由于默认的视图是由按惯例先于配置的原则决定的，我们这里真正做的所有事情是为了测试Rails的内置的函数。但是如果我们渲染了一个视图，动作成功了但出现一个不同的显示错误的视图，像这样的例子就需要来验证我们选择了正确的视图。

由于我们从第3章得知，方法**MoviesController#search_tmdb**的默认行为是尝试渲染视图`app/views/movies/search_tmdb.html.haml`（我们在第7章创建的），我们的规则仅仅需要确认控制器动作确实是尝试来渲染这个视图模板。我们将使用RSpec的**response**方法来完成这个任务：一旦完成了控制器规则中的**get**或**post**动作，**response**方法返回的主体将包含应用服务器对这个动作的应答，我们就可以确定应答是否按照期望 **将要渲染** 了一个特定的视图。图8.6的第15行举例说明了另一种RSpec断言： **object .should match-condition** 。在这个例子中， **match-condition** 是由**render_template()**提供的，所以这个断言当该主体（在本例中是来自控制器动作的响应）尝试渲染一个特定视图时被满足。我们将看到**should**和其它 **match-conditions** 一起的用法。否定的断言**should_not**（**不应当**）能被用来指定 **match-conditions**（**匹配条件**） 不是真的情况。

图8.6中有两件事情需要注意。首先，由于两个例子（第5～10行、第11～16行）都是自包含且**独立**的，我们需要为每一个都创建测试替身，并分别**发送**命令。第二，鉴于第一个例子使用了**should_receive**（**应当收到**），第二个例子使用了**stub**，创建了方法的一个测试替身但却 **没有** 确保方法必需被调用。 **如果** 方法被调用，这个替身会触发动作；但在方法从来没有被调用的情况下，触发动作也不算错误。把你的规则文件改成如图8.6所示的那样，`autotest`应该仍然会运行并且报告第二个例子通过测试。

代码文件：ch_tdd/code/spec/movies_controller_spec.4.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      it 'should call the model method that performs TMDb search' do
6        fake_results = [mock('movie1'), mock('movie2')]
7        Movie.should_receive(:find_in_tmdb).with('hardware').
8          and_return(fake_results)
9        post :search_tmdb, {:search_terms => 'hardware'}
10     end
11     it 'should select the Search Results template for rendering' do
12       fake_results = [mock('Movie'), mock('Movie')]
13       Movie.stub(:find_in_tmdb).and_return(fake_results)
14       post :search_tmdb, {:search_terms => 'hardware'}
15       response.should render_template('search_tmdb')
16     end
17     it 'should make the TMDb search results available to that template'
18   end
19 end
```

图 8.6: 填写第二个例子，用第11～16行代替图8.5中的第11行

在这个简单的例子中，你可能觉得在一个例子中使用**should_receive**以及在另一个例子中使用**stub**有点小题大做，但这里的目的是为了说明 **一个例子是为了测试一个单独的行为** 。第二个例子 **只** 测试是否选择了正确的视图来渲染。它 **没有** 测试是否调用了合适的模型方法——这是第一个例子的工作。实际上，即使方法**Movie.find_in_tmdb**已经实现好了，我们仍然要把它完全隔离在这些例子外，因为例子应该将要测试的行为孤立，使其与主体代码相关的其它类的行为没有关联。

在进行下一个例子之前，我们来考虑红色—绿色—重构过程中的重构步骤。注意到第6行和第12行是完全相同的，图8.7展示了一种用 **提取** 共同设置代码到**before(:each)**块中的方法来精简它们。正如它的名字所示，这段代码在每一个**describe**例子组中的例子之前都会被执行，和一个具有Cucumber特性的在第一个情景前执行的**Background**部分类似。还有**before(:all)**的代码块，是在整个组的测试中仅运行一次设置的代码。例如出现的难于调试的错误，只在以不同的顺序执行测试或只有一部分测试执行的情况下容易显露出来，使用这个代码块会增加测试彼此之间相互依赖的风险。

提取共同设置到一个 **前置** 块的概念是明确的，但由于RSpec实现的方法问题，我们不得不做出一个语法上的改变使它能正常工作。特别地，我们需要将**fake_results**变成一个实例变量**@fake_results**，因为在每个测试样例**do...end**模块中的局部变量在测试完成时都会消失。相反地，一组例子实例变量对该组所有的例子都是可见的。由于在**before :each**块中设置了它们的值，所以每一个测试样例将看到与**@fake_results**相同的初始值。

实例变量还是什么？ **describe** 创建及返回一个代表一组测试实例的新的**Test::Spec::ExampleGroup**对象。**@fake_results**是这个**ExampleGroup**对象的实例变量——而不是测试的类（**MoviesController**）。

现在只剩下一个例子要写，即检查TMDb搜索结果对响应视图是否可用。回想在第7章中，我们假设控制器动作将会创建**@movies**用以包含从TMDb中匹配到的电影列表，创建了`views/movies/search_tmdb.html.haml`。这是在**MoviesController#search_tmdb**中我们将调用**find_in_tmdb**的结果指派给实例变量**@movies**的原因。

RSpec的**assigns()**方法记录了在控制器方法中哪些实例变量接受了指派。这样**assigns(:movies)**返回的值（如果有的话）被赋值给了**search_tmdb**中的**@movies**，我们的规则文件仅仅需要验证控制器方法是否正确创建了这个变量。在我们的情形中，我们已经安排好将我们的替身当作虚构的方法调用的返回值，所

代码文件：ch_tdd/code/spec/movies_controller_spec.5.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      before :each do
6        @fake_results = [mock('movie1'), mock('movie2')]
7      end
8      it 'should call the model method that performs TMDb search' do
9        Movie.should_receive(:find_in_tmdb).with('hardware').
10         and_return(@fake_results)
11       post :search_tmdb, {:search_terms => 'hardware'}
12     end
13     it 'should select the Search Results template for rendering' do
14       Movie.stub(:find_in_tmdb).and_return(@fake_results)
15       post :search_tmdb, {:search_terms => 'hardware'}
16       response.should render_template('search_tmdb')
17     end
18     it 'should make the TMDb search results available to that template'
19   end
20 end
```

图 8.7：使用一个before块（第5～7行）使控制器例子精简重复

以**search_tmdb**方法正确的行为应该是把这个值赋给**@movies**，就像图8.8的第21行中断言的那样。

> ■ **详细阐述：比我们想要的还多？**
>
> 从严格意义上来讲，按照这个例子的目的，**find_in_tmdb**的桩可以返回任何值，例如字符串“I am a movie”（我是一部电影），都没有问题，因为这个例子测试的 **唯一的** 行为是正确地创建了对视图来说是可用的实例变量。特别地，这个例子并不关心这个变量的 **值** 是什么，或者**find_in_tmdb**是否返回了合理的东西。但由于我们已经创建了替身，在这个例子中使用它们是很容易的。

红色—绿色—重构模式的最后一个任务是重构步骤。第二个和第三个例子除了最后一行（第16行和第21行）外是完全一样的。为了精简它们的重复，图8.9使用**describe**开始的独立的嵌套样例组，将后面两个例子的公共行为部分放入他们自己的**before**块中。我们选择描述字符串`after valid search`（有效搜索后）来命令这个**describe**块，是因为这个组的例子假设的一个有效的**find_in_tmdb**调用实际发生了。（这个假设自身用第一个例子测试）

当例子被按组嵌套起来时任何与嵌套外层并且与**before**块有关的块都在相关的嵌套内层之前优先执行。所以，举例来说，考虑图8.9中第18～20行的测试样例，第5～7行的设置代码首先运行，然后是第14～17行的设置代码，最后例子本身（第18～20行）运行。

下一个目标将是使用TDD来创建我们到目前未完成的模块方法**find_in_tmdb**。由于这个方法应该调用真实的TMDb服务，我们将再次使用桩，这一次是要避免我们的例子依赖于远程互联网服务。

代码文件：ch_tdd/code/spec/movies_controller_spec.6.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      before :each do
6        @fake_results = [mock('movie1'), mock('movie2')]
7      end
8      it 'should call the model method that performs TMDb search' do
9        Movie.should_receive(:find_in_tmdb).with('hardware').
10         and_return(@fake_results)
11       post :search_tmdb, {:search_terms => 'hardware'}
12     end
13     it 'should select the Search Results template for rendering' do
14       Movie.stub(:find_in_tmdb).and_return(@fake_results)
15       post :search_tmdb, {:search_terms => 'hardware'}
16       response.should render_template('search_tmdb')
17     end
18     it 'should make the TMDb search results available to that template' do
19       Movie.stub(:find_in_tmdb).and_return(@fake_results)
20       post :search_tmdb, {:search_terms => 'hardware'}
21       assigns(:movies).should == @fake_results
22     end
23   end
24 end
```

图 8.8: 断言@movie被search_tmdb正确地创建。这个列表中第18～22行代替了图8.7中的第18行

代码文件：ch_tdd/code/spec/movies_controller_spec.7.rb

```
1  require 'spec_helper'
2
3  describe MoviesController do
4    describe 'searching TMDb' do
5      before :each do
6        @fake_results = [mock('movie1'), mock('movie2')]
7      end
8      it 'should call the model method that performs TMDb search' do
9        Movie.should_receive(:find_in_tmdb).with('hardware').
10         and_return(@fake_results)
11       post :search_tmdb, {:search_terms => 'hardware'}
12     end
13     describe 'after valid search' do
14       before :each do
15         Movie.stub(:find_in_tmdb).and_return(@fake_results)
16         post :search_tmdb, {:search_terms => 'hardware'}
17       end
18       it 'should select the Search Results template for rendering' do
19         response.should render_template('search_tmdb')
20       end
21       it 'should make the TMDb search results available to that template' do
22         assigns(:movies).should == @fake_results
23       end
24     end
25   end
26 end
```

图 8.9: 完成并重构search_tmdb的规则文件。从第13行开始的嵌套在一起的部分让我们能把图8.8中第14和15行和第19和20行的重复部分精简掉

小结：

- 红色—绿色—重构模式中重构步骤的例子是把公共设置的代码移到**before**块中，这精简了规则文件中的重复。
- 像**should_receive**一样，使用**stub**模块化创建一个“测试替身”方法供用户来测试，但与**should_receive**不同的是，**stub**并不需要这个方法被真实地调用。
- **assigns()**允许一个控制器测试来检查控制器动作设置的实例变量的值。

自我检查8.4.1. 指出下列每个Rspec架构是被用来（a创建一个嫁接，（b决定嫁接的行为（c都不是：

（1）**assigns()**；（2）**should_receive**；（3）**stub**；（4）**and_return**。

◇（1）c，（2）a，（3）a，（4）b。

自我检查8.4.2. 为什么**before(:each)**比**before(:all)**在大多数情况下更合适？

◇在**before(:each)**块前的代码在这个块里每一个规则运行之前运行，为这些规则设置完全相同的先决条件并保持它们的独立性。

8.5 夹具和工厂

在需要一个少量功能的替身来表示一个测试实例时，模拟和桩是合适的。但是假设你正在测试一个新的方法CMovie#name_with_rating，你知道它将检查一个**Movie**对象的**title**（**标题**）和**rating**（**评级**）属性。你可以创建一个知道所有信息的替身并传递这个替身。

代码文件：ch_tdd/code/mock_example.rb

```
fake_movie = mock('Movie')
fake_movie.stub(:title).and_return('Casablanca')
fake_movie.stub(:rating).and_return('PG')
fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

但是这里不使用模拟的原因有两个。首先，这个模拟的对象需要和真实的**Movie**对象有几乎一样多的功能，使用真实的对象可能会更好。第二，由于被测试的实例方法是**Movie**类自身的一部分，所以这并不需要将测试代码从其它类中孤立出来，使用真实的对象更有意义。

你有两种获得本测试中要使用的真实**Movie**对象的方法。一种是创建一个或多个**夹具**——用作一个或多个测试基准的一个确定状态。术语 **夹具** 来自制造行业：一个测试夹具是一个用来固定或支撑被测试对象的装置。由于Rails的SaaS应用中的所有状态都存储在数据库中，一个夹具文件定义了一组可以在测试运行前自动载入测试数据库的对象，所以你在你的测试中可能使用这些对象而不需要创建它们。和创建与拆除模拟和桩一样，测试数据库可以与 **每个规则文件** 前的夹具一起被擦除或重新加载，使测试保持**独立性（I）**。Rails会在文件中查找包含 **YAML**（Yet Another Markup Language）对象的夹具。如图8.10所示，YAML是一个使用属性来表示对象层次结构的很简单的方法，和我们在本章开头看到的XML类似。如图8.10所示，这个**Movie**模型的夹具是从`spec/fixtures/movies.yml`加载的。并且，通过使用它们的符合名称的方法，这些夹具对你的规则是可用的。

严格来讲，这并不是可擦除的，但在每一个规则完成时，该规则运行的 **数据库事件** 都会被回滚。

代码文件：ch_tdd/code/fixtures_example.rb

```
# spec/fixtures/movies.yml
milk_movie:
  id: 1
  title: Milk
  rating: R
  release_date: 2008-11-26

documentary_movie:
  id: 2
  title: Food, Inc.
  release_date: 2008-09-07
```

代码文件：ch_tdd/code/spec_using_fixtures.rb

```
# spec/models/movie_spec.rb:

describe Movie do
  fixtures :movies
  it 'should include rating and year in full name' do
    movie = movies(:milk_movie)
    movie.name_with_rating.should == 'Milk (R)'
  end
end
```

图 8.10: 在YAML文件（上半部分）中声明的夹具，是在规则（下半部分）被执行前自动加载到测试数据库的

但除非小心地使用，夹具可能干扰到测试的**独立性**，由于每个测试都隐式地依赖于装置的状态，所以改变夹具可能改变测试的行为。另外，尽管每个单独的测试可能仅依赖于一两个装置，但所有测试都需要的装置集合可能变得很臃肿。由于这个原因，许多程序员更偏好于使用另一种方法，即 **工厂** ——一个被设计出允许在测试时快速创建包含所有功能的对象（而不是模拟）。例如，流行的FactoryGirl[6] Rails工具允许你来定义一个Movie对象的工厂，为每一个测试快速创建你需要的对象，选择性地覆盖仅仅一部分属性，就像图8.11中那样（FactoryGirl是本书工具包的一部分）。在这个简单的应用中，使用一个工厂并不比直接调用**Movie.new**方法来创建个新的Movie对象有更多好处。但在更复杂的应用中，创建并初始化一个涉及很多步骤的对象——例如对象可能有很多需要在创建时初始化的属性——一个工厂能帮助精简测试先决条件（**before**块）并使你的测试代码流程合理化。

在添加更多功能之前，让我们再深入挖掘一下RSpec工作原理。RSpec的**should**是一个使用Ruby语言功能来提升可读性、消除测试和文档之间的隔阂的很好的例子。下面的视频解释了如何处理像**value.should == 5**这样的表达式的更多细节。

视频8.5.1: Ruby的动态语言特性如何使规则文件更具可读性.

光盘文件：ch_tdd/mov/should.mp4

Rspec把一个包含**should**方法的模块添加到了**Object**类。**should**期望传递一个可以评估断言状态的 **适配器** 。像**be**这样的RSpec方法能用来构造这样一个匹配，因为Ruby有灵活的语法和随意的括号，像**value.should be < 5**这样的断言可以通过完全括号化和反语法糖转化成**value.should(be.<(5))**来理解。另外，Rspec使用了Ruby的**method_missing**功能（参见第3章描述）来检测以**be_**或**be_a_**开头的适配器，允许创建**cheater.should be_disqualified**这样的断言。 **（注意：这个Alpha版本视频开头展示的规则与本节中开发的例子并不匹配。然而，这并不影响视频的主要意义，即阐明RSpec中should是如何工作的）**

代码文件：ch_tdd/code/factory_example.rb

```
# spec/factories/movie.rb

FactoryGirl.define do
  factory :movie do
    title 'A Fake Title' # default values
    rating 'PG'
    release_date { 10.years.ago }
  end
end
```

代码文件：ch_tdd/code/spec_using_factory.rb

```
# spec/models/movie_spec.rb

describe Movie do
  it 'should include rating and year in full name' do
    movie = FactoryGirl.build(:movie, :title => 'Milk', :rating => 'R')
    movie.name_with_rating.should == 'Milk (R)'
  end
end

# or if you mix in FactoryGirl's syntax methods (see FactoryGirl README):

describe Movie do
  subject { create :movie, :title => 'Milk', :rating => 'R' }
  its :name_with_rating { should == 'Milk (R)' }
end
```

图 8.11: 一个使用FactoryGirl代替夹具来创建真实（不是模拟的）对象进行测试的例子。工作框架使这样的对象创建流程化并保护了测试的独立性。添加gem 'factory_girl_rails'到Gemfile来使用FactoryGirl

小结：

- 当一个测试需要在实际的对象上操作而不是模型上时，真实的对象可以通过工厂来快速地创建，或预先加载进来作为一个夹具。但需要注意的是，夹具有可能引起测试之间微妙的相关性，破坏了**独立性（I）**。
- 测试是内部文档的一种形式。RSpes拓展了Ruby语言的功能，使你能够写出非常具有可读性的测试代码。像应用代码一样，测试代码是给人看的，而不是给计算机，所以花时间把代码变得可读不仅加深了你对它们的理解。而且，对你不在的时候处理这些代码的人来说，测试代码能将你的想法高效地文档化。

自我检查8.5.1. 假设一个测试组件包含一个测试，这个测试在一个表格中添加了模型对象，然后希望找到一个给定数目的模型对象作为返回结果。解释夹具的使用如何影响到这一组测试的独立性，以及如何用工厂来弥补这个问题。

◇ 如果夹具文件改变了，以致在这个表格中初始存在的条目数改变了，这个测试有可能会由于对表格初始状态的假设不再成立而在开始后很快失败。相反地，工厂能用来快速地创建这些仅在每个测试中分别需要的对象，或这一组测试需要的例子，所以没有测试需要依赖于全局的数据库“初始状态”。

8.6 面向模型的TDD

我们现在已经创建了新功能“搜索TMDb”三个部分中的两个：我们在第7章创建

代码文件：ch_tdd/code/spec/movie_spec.1.rb

```
require 'spec_helper'

describe Movie do
  describe 'searching Tmdb by keyword' do
    it 'should call Tmdb with title keywords' do
      TmdbMovie.should_receive(:find).with(hash_including :title => 'Inception
          ')
      Movie.find_in_tmdb('Inception')
    end
  end
end
```

代码文件：ch_tdd/code/movie.1.rb

```
class Movie < ActiveRecord::Base

  def self.find_in_tmdb(string)
    TmdbMovie.find(:title => string)
  end

  # rest of file elided for brevity
end
```

图 8.12: 上图：调用TMDb工具包的快乐路径规则，下图：快乐路径规则驱动的初始快乐路径实现

了视图，在上一节中使用TDD驱动了控制器动作的创建。完成第7章开始的用户故事，剩下所有工作就是模型方法**find_in_tmdb**，即实际使用面向服务的架构技术来与TMDb通信。使用TDD来驱动实现将会进展很快，因为我们已经了解了它的基础。

根据惯例先于配置原则，**Movie**模型的规则文件在`spec/models/movie_spec.rb`中。图8.12展示了调用**find_in_tmdb**的快乐路径，即所有事情都工作正常时的情况（完整的规则必须包含悲伤路径，就像我们很快看到的一样）。在全部的**describe Movie**中，我们为关键词搜索函数添加了一个嵌套的**describe**块。第一个规则是，当**find_in_tmdb**被传入一个字符串变量调用时，它应该将这个字符串变量赋给TMDb工具包的**TmdbMovie.find**方法的**:title**选项。我们使用**hash_including**这一适配器来表示它，该适配器详细说明了一个特殊的变量（本例中唯一的变量）必须是一个包含键/值对**:title =>'Inception'**的哈希值。这个规则马上会失效，因为还没有定义**find_in_tmdb**，所以我们又到了红色阶段。当然，在这一阶段**find_in_tmdb**是不重要的，所以图8.12中的下图展示了让我们从红色到绿色的初始实现。

但是，这个规则还差一点才完成！到目前为止，所有的测试样例是基于第7章中的用户故事描述的 **显式的要求**：当用户输入电影的名称并点击 **Search TMDb** 按钮，他会看到一个显示匹配结果的页面。然而，视频8.1.1显示了附加的 **隐式的要求**，即每一个TMDb请求都要包含一个有效的API密钥。更进一步地，视频8.1.2展示了`ruby-tmdb`工具包给出的一个空白而不是无效API密钥的 **不同** 错误，所以必须准备好处理这两种错误。实际上，尽管Ruby并不像Java那样要求声明异常，但理解我们所依赖的其它代码可能引起的异常，是开发健壮的程序的重要部分。因此，在与一个远程的服务交互时，我们还是需要理解可能的异常行为，以及它们在应用中会如何表现出来。我们的规则文件到目前为止并没有覆盖隐式的需要，而且因为我们已经将调用转到模拟的**TmdbMovie.find**，我们已经掩蔽了其它情况下会发生的错误。

所以必须论证模型方法的描述来抓住这些隐式的需求。

- 在提供了一个无效的密钥的情况下，它应该产生一个异常。

代码文件：ch_tdd/code/spec/movie_spec.2.rb

```
1  require 'spec_helper'
2
3  describe Movie do
4    describe 'searching Tmdb by keyword' do
5      it 'should call Tmdb with title keywords given valid API key' do
6        TmdbMovie.should_receive(:find).
7          with(hash_including :title => 'Inception')
8        Movie.find_in_tmdb('Inception')
9      end
10     it 'should raise an InvalidKeyError with no API key' do
11       lambda { Movie.find_in_tmdb('Inception') }.
12         should raise_error(Movie::InvalidKeyError)
13     end
14   end
15 end
```

图 8.13：我们想要的代码应当引起一个非常特殊的异常来标志丢失的API密钥（第11～12行），但因为是find_in_tmdb取而代之地抛出了它自己的异常，这个规则失败了

- 在提供了一个空白的密钥的情况下，它应该产生一个异常。

不幸的是，实现`ruby-tmdb`的方法产生了一个**运行时错误**（RuntimeError），即一种最常见的Ruby异常。在Rotten Potatoes中，根据按图8.13中要求修改过的规则文件，我们更倾向于，对这种类型的错误来说，**find_in_tmdb**引起了一种特定的异常。应注意我们重新命名了第一个规则文件来表明它是被用于API密钥有效时的改进的情形，并添加了一个新的规则文件来覆盖API密钥没有初始化的情形。

让我们花点时间来理解代码本身以及它为什么变成红色失败了。首先要注意，我们需要把对**find_in_tmdb**的调用“包含”在一个**lambda**中。我们**期望**这个调用抛出一个异常，但是如果一个规则抛出了一个异常，它就停止测试！所以为了使规则文件**自验证**，应在可用的**lambda**对象上调用**should**方法，这将使lambda在一个RSpec可以捕获任何异常并把它们匹配到我们的异常“可控制的环境”中被执行。

然而，测试失败消息是`uninitialized constant Movie::InvalidKeyError`。回顾一个Ruby类就是一个简单的常量，命名了一个类的对象，这个消息告诉大家我们引用了一个**Movie::InvalidKeyError**的异常类但没有在任何地方定义它。图8.14的第3行通过在**Movie**的命名空间（因此spec文件第12行使用符号**Movie::InvalidKeyError**）下定义这个新的类来修正了这个错误。当做了这个改变后，测试仍然是失败的，但现在它是因为已知的原因失败了：我们断言了**Movie::InvalidKeyError**应该被抛出，但被**TmdbMovie.find**中的**ArgumentError**替代了。图8.14给出了一个简单的方法使这个测试变成绿色通过。注意在第9行，我们将来自原始异常**参数类型错误**（ArgumentError）的错误信息当作新的异常描述消息来传递。

代码文件：ch_tdd/code/movie.3.rb

```
1  class Movie < ActiveRecord::Base
2
3    class Movie::InvalidKeyError < StandardError ; end
4
5    def self.find_in_tmdb(string)
6      begin
7        TmdbMovie.find(:title => string)
8      rescue ArgumentError => tmdb_error
9        raise Movie::InvalidKeyError, tmdb_error.message
10     end
11   end
12
13   # rest of file elided for brevity
14 end
```

图 8.14：在Movie类定义新的异常类型（第3行）并添加上我们希望的处理这个由TmdbMovie.find抛出异常的行为，重新抛出新的异常类型

小结：

- 有时明确的需求会引出额外的隐式的需求——类似显式的需求一样却不"可见"的额外约束，但为了满足显式的需求它们仍然必须被满足。隐式的需求和显式的需求一样重要，同样需要严格地测试。
- 如果需要检查对象代码抛出异常的情况，则可以通过创建一个lamda表达式，像**should**或**should_not**这样的预期的接收者表达式，并使用适配器**raise_error**来实现。
- 请参考例子中如何用**with**来执行一个局部的参数检查，我们用**hash_including**来匹配参数，这一方法检查那些类似于哈希值并包含着一些特殊的键/值对的参数，但对参数的其它内容并不做断言。

■ **详细阐述：声明和未声明的异常**

在Java这样的静态类型语言里，编译器强制每个方法都必须声明它可能抛出的异常。如果被调用者想要添加一个新的异常，被调用者的方法签名就会改变，需要重新编译调用者和被调用者。这种方法并没有扩展到SaaS应用中，这里有很多与其它像TMDb这样的服务的交互，这些服务的演化和行为并不受我们控制。我们必须依赖于远程服务的API文档来告诉我们哪儿可能出错，但同时必须假设其它没有文档化的失败也可能发生。在这些情况下，最好的方法就是捕获异常，记录详细的信息，并通知`ruby-tmdb`工具包的维护者或TMDb服务有一个未预料的错误发生。

自我检查8.6.1. 考虑图8.13中第11行，假设没有将**find_in_tmdb**方法包含在一个lamda表达式中，将会发生什么情况？为什么？

◇ 如果**find_in_tmdb**正确地引起了异常，那么规则测试将会失败，因为异常将使测试停止运行。如果**find_in_tmdb**没有成功抛出一个异常，那么规则测试仍会失败，因为断言**should raise_error**期望一个异常。这样测试将永远失败，不论**find_in_tmdb**是否正确。

自我检查8.6.2. 初始化一个有效的API密钥失败会抛出一个异常，为什么图8.13的第8行没有抛出一个异常？

代码文件：ch_tdd/code/movie.5.rb

```
1  class Movie < ActiveRecord::Base
2
3    class Movie::InvalidKeyError < StandardError ; end
4
5    def self.api_key
6      'cc4b67c52acb514bdf4931f7cedfd12b' # replace with YOUR Tmdb key
7    end
8
9    def self.find_in_tmdb(string)
10     Tmdb.api_key = self.api_key
11     begin
12       TmdbMovie.find(:title => string)
13     rescue ArgumentError => tmdb_error
14       raise Movie::InvalidKeyError, tmdb_error.message
15     end
16   end
17   # rest of file elided for brevity
18 end
```

图 8.15：一个在第5～7行定义的并在第10行使用的简单的初始化API密钥的方法。在真实的应用中，密钥字符串很可能是从一个配置文件中读取的，但创建这个存取方法使我们能够改变检索密钥的方法而不影响其它代码

确认 获取并使用你自己的TMDb API密钥[7]而不是硬编码在示例程序中的那个！

◇ 因为第6和7行用一个桩替代了方法**TmdbMovie.find**调用，防止了调用“真实的”方法而引起异常。

为什么在每个find之前都要设置密钥 而不是只设置一次？设置密钥是一个方便的操作，所以检查密钥是否被初始化而添加的额外的代码将影响代码的简明性而不提升性能。如果设置密钥代价比较高，一次初始化可能更明智。

8.7 为Internet建立桩

上面的悲伤路径处理了根本没有初始化API密钥的情景。如果指定一个TMDb key，但它并不是有效的会怎样？这最终需要我们添加一些代码来实际初始化API密钥。图8.15添加了一个密钥的存取方法并设定了每一次调用**find_in_tmdb**的密钥值。

但如果密钥是有效的，那么这种改变应当导致规则测试失败，因为第12行的规则将不再引起异常。我们必须更新规则文件来故意将API密钥设置成空字符串（就像我们没有初始化它时的样子）。幸运的是，我们已经知道怎么做了：我们隔离了**Movie.api_key**存取器并返回一个空的字符串，如图8.16所示。这个改变使第12行的规则恢复为绿色。注意通过调用存取器方法设置一个密钥的做法，在这个测试中创建了一个嫁接，使我们可以使用桩测试来产生一个失败值的嫁接。如果密钥是用变量来设置的，这将很难做到。在这里使用这个嫁接，我们最终能够测试非空但无效的密钥的行为。我们可以很容易地写出图8.17中的规则，因为大部分我们已经写好了。可能令你很惊讶的是，规则并没有通过：

代码文件：ch_tdd/code/tmdb_runtime_error.txt

```
expected Movie::InvalidKeyError, got #<RuntimeError: Tmdb API returned
status code '404' for URL:
'http://api.themoviedb.org/2.1/Movie.search/en/json/INVALID/Inception'>
```

这里有两个问题。首先，如视频8.1.2所示，当密钥为空时**TmdbMovie.find**引起**ArgumentError**，当密钥是无效的时候却引起**RuntimeError**。（大概的原因是空密钥显然是无效的，但非空密钥只能通过调用TMDb服务来测试。）引起不同异常的问题可以通过添加另一个**rescue**捕获**RuntimeError**异常的子句来解决，如图8.18所示。

这将我们引到了第二个更严重的错误：这个规则每次运行时实际上调用

代码文件：ch_tdd/code/spec/movie_spec.3.rb

```
require 'spec_helper'

describe Movie do
  describe 'searching Tmdb by keyword' do
    # first spec elided for brevity
    it 'should raise an InvalidKeyError with no API key' do
      Movie.stub(:api_key).and_return('')
      lambda { Movie.find_in_tmdb('Inception') }.
        should raise_error(Movie::InvalidKeyError)
    end
  end
end
```

图 8.16: 因为图8.15使用了存取器方法Movie.api_key来获取密钥的值而不是从一个变量来指定，它提供了一个嫁接使我们能够控制测试的行为

代码文件：ch_tdd/code/spec/movie_spec.4.rb

```
require 'spec_helper'

describe Movie do
  describe 'searching Tmdb by keyword' do
    # first 2 specs elided for brevity
    it 'should raise an InvalidKeyError with invalid API key' do
      Movie.stub(:api_key).and_return('INVALID')
      lambda { Movie.find_in_tmdb('Inception') }.
        should raise_error(Movie::InvalidKeyError)
    end
  end
end
```

图 8.17: 测试非空但无效的密钥引起了一个失败，因为TmdbMovie.find引起了与一个无效的密钥的异常，不同于空白或未初始化的异常

代码文件：ch_tdd/code/movie.4.rb

```
class Movie < ActiveRecord::Base

  class Movie::InvalidKeyError < StandardError ; end

  def self.api_key
    'cc4b67c52acb514bdf4931f7cedfd12b' # replace with YOUR Tmdb key
  end

  def self.find_in_tmdb(string)
    Tmdb.api_key = self.api_key
    begin
      TmdbMovie.find(:title => string)
    rescue ArgumentError => tmdb_error
      raise Movie::InvalidKeyError, tmdb_error.message
    rescue RuntimeError => tmdb_error
      if tmdb_error.message =~ /status code '404'/
        raise Movie::InvalidKeyError, tmdb_error.message
      else
        raise RuntimeError, tmdb_error.message
      end
    end
  end
  # rest of file elided for brevity
end
```

图 8.18: 和处理参数类型错误（**ArgumentError**）一样处理运行时错误（**RuntimeError**）能使测试变为通过，但由于运行时错误是一个一般性的捕获所有未处理的运行时异常，我们特别地检查了在视频**8.1.1**中看到的匹配**TMDb**错误信息的异常信息。在其它情况下，我们仅仅重新引发原来的RuntimeError

代码文件：ch_tdd/code/spec/movie_spec.5.rb

```
1  require 'spec_helper'
2
3  describe Movie do
4    describe 'searching Tmdb by keyword' do
5      it 'should call Tmdb with title keywords given valid API key' do
6        TmdbMovie.should_receive(:find).
7          with(hash_including :title => 'Inception')
8        Movie.find_in_tmdb('Inception')
9      end
10     it 'should raise an InvalidKeyError with no API key' do
11       Movie.stub(:api_key).and_return('')
12       lambda { Movie.find_in_tmdb('Inception') }.
13         should raise_error(Movie::InvalidKeyError)
14     end
15     it 'should raise an InvalidKeyError with invalid API key' do
16       TmdbMovie.stub(:find).
17         and_raise(RuntimeError.new("API returned status code '404'"))
18       lambda { Movie.find_in_tmdb('Inception') }.
19         should raise_error(Movie::InvalidKeyError)
20     end
21   end
22 end
```

图 8.19：当一个无效的密钥传入时模拟TMDb API的行为（第16和17行）破坏了无效API密钥规则和TMDb自身之间的依赖性

了TMDb服务，使它不仅不**快**（F）（它可能会花费数秒钟来完成每次调用）而且不**可重复**（R）（这个测试将在TMDb宕机或计算机没联网时表现得不同）。

我们已经知道如何来修正它：引入一个嫁接将调用者孤立于被调用者，就像之前对**TmdbMovie.find**所做的一样。由于现在知道**TmdbMovie.find**在接收到一个无效的密钥时会抛出一个**RuntimeError**，所以可以用一个桩来模拟那样的行为。注意到这个桩淘汰了保存一个无效的密钥值的需要，这使得规则测试为绿色但却使它与实际的TMDb服务“断开连接”。

图8.19引出了一个普遍的问题：什么时候需要建立外部方法的桩？我们选择构建桩**find_in_tmdb**来对TMDb进行集成测试，但是其实有多个地方都可以使用桩。`ruby-tmdb`工具的桩方法将允许我们的测试在**Movie.find_in_tmdb**和`ruby-tmdb`的交互中捕获错误。更加根本的是，我们可以用一个像`FakeWeb`[8]的工具来隔离除了特殊的URI（通用资源标识符）之外的整个网页，URI可以在从Ruby程序访问时返回一个事先准备好的响应。举例来说，我们可以通过剪贴一个活动调用的结果创建一个事先准备好的XML文档，并在使用合适的RESTful URI来尝试连接TMDb时使用FakeWeb来整理返回的文档。从一个集成测试的立场来看，这是一个最现实的测试方法，因为桩的行为是“最远的”——我们在请求流中尽可能晚地构建桩。这样，当创建Cucumber情景来测试外部服务整体时，`FakeWeb`是一个合适的选择。从一个单元测试点的视角（就像我们在本章采取的）来看这并不怎么引人注目，因为我们在考虑一个特定类方法的正确行为，并且我们并不介意“在附近”构建桩以便于在一个可控制的环境中观察这些行为。

小结：

- 为了测试与外部服务通信的代码，我们创建了桩模块来模拟服务的行为，目的是确保我们测试**快速**（F）并且**可重复**（R）。
- 我们可以通过阅读服务的API文档以及直接观察服务在一系列情况下真实的行为来了解我们要模拟的行为。

测试样例结构：

- **before(:each) do... end**
  设定在每个规则之前执行的先决条件（使用**before(:all)**使其仅执行一次，风险由你自己承担）
- **it 'should do something' do... end**
  对应于一个行为的单独的例子（测试样例）
- **describe 'collection of behaviors' do... end**
  一组相关的例子的组合

模仿和桩：

- **m=mock('movie')**
  创建一个不包含预先定义方法的模仿对象
- **m.stub(:rating).and_return('R')**
  代替**m**上已有的**分级**（rating）方法，或在没有的情况下定义一个新的**分级**（rating）方法，它返回一个事先准备好的响应**'R'**
- **m=mock('movie', :rating=>'R')**
  将以上两个例子结合起来的捷径
- **Movie.stub(:find).and_return(@fake_movie)**
  在 **Movie.find**被调用的时间强制返回**@fake_movie**，但不要求一定要被调用。

有用的方法和控制器规则文件对象： 你的规则文件必须在`spec/controllers`子目录中以使这些方法可用。

- **post '/movies/create',**
  {:title=>'Milk', :rating=>'R'}
  引起一个`/movies/create`的POST请求并将传入的哈希值作为**参数**（params）传递**get**，**put**，**delete**同样可用。
- **response.should render_template('show')**
  检查控制器动作对这个控制器的模型执行**show**模板的情况。
- **response.should redirect_to(:controller => 'movies', :action => 'new')**
  检查控制器方法重定向到**MoviesController#new**而不是执行一个视图的情况。

图 8.20: 本章中介绍的一些非常有用的RSpec方法。更多细节及这里没列出的其它方法参见完整的RSpec文档[9]

方法调用中的断言：也可以是否定的，例如**should_not_receive**

- **Movie.should_receive(:find).exactly(2).times**
  创建**Movie.find**桩并确保它被恰好调用了两次（如果你不关心调用了多少次，请省略**exactly**；**at_least()**和**at_most()**同样可用。
- **Movie.should_receive(:find).with('Milk','R')**
  检查**Movie.find**被调用时传入的恰好是两个有这样值的参数的情况。
- **Movie.should_receive(:find).with(anything())**
  检查**Movie.find**被调用时传入的是一个参数，而不检查这个参数的值的情况。
- **Movie.should_receive(:find).**
  with(hash_including :title=>'Milk')
  检查**Movie.find**被调用时传入了一个参数的情况，这个参数必须是一个包含键**:title**和值**'Milk'**的哈希值（或者像哈希值的参数）。
- **Movie.should_receive(:find).with(no_args())**
  检查**Movie.find**被调用时没有传入参数的情况。

适配器

- **greeting.should == 'bonjour'**
  比较它的参数是否与断言的接收器一致。
- **value.should be >= 7**
  比较它的参数与给定的值，**value.should(be.>=(7))**的语法糖。
- **result.should be_remarkable**
  在**result**上调用**remarkable?**（注意问号）（来判断**result**是否有效）。

图 8.21：本章中介绍的有用的RSpec方法附加部分总结（续）

代码文件：ch_tdd/code/coverage_example.rb

```
1  class MyClass
2    def foo(x,y,z)
3      if x
4        if (y && z) then bar(0) end
5      else
6        bar(1)
7      end
8    end
9    def bar(x) ; @w = x ; end
10 end
```

图 8.22: 一个简单的阐述基本覆盖概念的代码样例

■ **详细阐述：红色—重构—绿色—重构?**

有时从红色到绿色并不仅仅是添加新代码的问题，而有可能是首先需要重构已有的代码。例如，如果图8.15第10行中从（**Tmdb.api_key=@api_key**）设置API密钥，而不是调用第5~7行的存取器方法，我们将要通过重构来创建存取器方法以获得图8.16第7行的测试嫁接。这个重构很简单，但是更复杂的重构可能需要代码临时改变，并使一些绿色的测试变成红色。在继续新的测试前必须修改这些测试来使它们再次通过测试，这是非常重要的。记住我们的目标是永远保持代码有效。

自我检查8.7.1. 说出两种当单元测试实际调用一个外部服务作为测试的一部分时可能违反FIRST原则的项。
◇ 测试可能不再是快速的，因为调用一个外部服务会比本地计算花费长非常多的时间。测试也可能不再是可重复的，因为超出我们控制的环境可能影响它的输出，例如外部服务临时不可用的情况。

自我检查8.7.2. 如果**TmdbMovie**对空密钥或无效密钥会引起 **同样的** 异常，如何简化图8.19中的规则文件?
◇ 只需要一个要检查的情形，规则文件中第10~14行或第15~19行可以从移除。

8.8 覆盖率概念、单元测试与集成测试

我们需要多少测试才够？一个简单却不幸经常听到的答案是“在交付日期之前做尽可能多的测试。”一个粗糙的替代方法是 **代码测试比（code-to-test ratio）**，即除掉注释的代码行数除以所有类型的测试的行数。在产品系统里这个比率一般比1小，就是说，测试的行数比应用代码的行数要多。Rails应用根目录里面的命令`rake stats`能够根据RSpec测试的行数和Cucumber情景的行数计算这个比率。

一个更精确的解决这个问题的方法是根据 **代码覆盖** 来看。由于测试的目的是至少按照它应该在产品中表现的样子来使用测试主体代码，这个由测试组件实际测试的可能比例是多少？令人惊讶的是，测量覆盖率并不像你猜想的那样简单直接。下面（参见图8.22）是一个简单的代码片断以及一些经常在例子中应用的覆盖率的定义。

- S0或称为“方法覆盖”：每一个方法都被测试组件执行了至少一次吗？满足S0要求至少分别执行**foo**和**bar**一次。

有时会用一个下标，即S_0来写。

- S1或称为“调用或入口/出口覆盖”：每一个方法都从每个可能调用它的地方被调用了吗？满足S1要求从第4行和第6行分别调用**bar**。
- C0或称为“语句覆盖”：把条件语句的两个分支也分别计算为一个单独的语句，每个源代码的语句都被测试组件执行了至少一次吗？除了调用**bar**之外，满足C0要求参数**x**为真并且至少调用的**foo**一次（否则，第4行的声明将永远不会被执行），以及参数**x**为假时调用的**foo**至少一次。
- C1或称为“分支覆盖”：每一个分支都被执行了一次吗？满足C1要求参数**x**为真和为假，以及**y**和**z**满足使第4行的**y && z**取值分别为真假时各调用**foo**一次。一个更严格的条件，即 **决策覆盖**，要求每一个独立影响条件表达式的 **子表达式** 都要从真到假遍历决定的。在本例中，一个测试需要额外地分开设置**y**和**z**使条件**y && z**在**y**和**z**分别为假时调用一次。
- C2或路径覆盖：每个代码中可能的路径都被执行了吗？在本例中，**x**、**y**、**z**是被当作布尔变量的，一共有8种可能的路径。
- 改进的条件/决策覆盖（MCDC）结合了以上级别的子集：每一个程序中的入口和出口至少都被调用了一次，每一个程序中的决策都被遍历了所有的可能输出至少一次，每一个决策中的条件都展示了它独立地影响决策的输出的情况。

达到C0覆盖要求是相对简单的，一个100%的C0覆盖并非不可能。达到C1覆盖会更难一些，因为它要求更小心地构建测试样例，确保每个分支的每个方向都至少被执行了一次。C2覆盖是最难的，并不是所有的测试专家都同意100%路径覆盖所能带来的附加价值。因此，代码覆盖数据，对反映他们认真测试过或没有测试过的代码的比例，以及展示测试组件的整体全面性是最有价值的。下一个视频展示了如何使用Ruby工具包SimpleCov[10]（包括在本书工具包中）来快速地检查RSpec测试的C0覆盖情况。

视频8.8.1: 使用SimpleCov检查C0覆盖.
光盘文件：ch_tdd/mov/simplecov.mp4
作为Ruby工具包的SimpleCov工具，能测量并显示规则文件的C0覆盖情况。可以放大到每个文件来看哪些特定的行被测试所覆盖。

本章中以上关于覆盖率的讨论，重点在单元测试上。第7章解释了用户故事是怎样自动变成验收测试的，这些是 **集成测试** 或 **系统测试**，因为每个测试（即每个情景）都使用了应用中许多不同部分的很多代码，而不是依赖于假装的对象如模拟或桩来将类从其协作者孤立出来。集成测试是重要的，但也是不充分的。它的解析度太低：如果一个集成测试失败了，由于测试涉及了许多部分的代码而很难定位到失败原因。它的覆盖率一般也较低：因为即使是一个单一的情景也涉及很多类，它只执行了每个类中的一小部分代码路径。由于相同的原因，集成测试也常常花费更长的时间来执行。另一方面，单元测试执行却很快，并能很精确地和主体代码隔离开来（提高覆盖率解析度同时易于错误定位）。但由于它们依赖于虚拟的对象来隔离主体代码，它们可能掩盖那些只会在集成测试中出现的问题。

在这些覆盖级别之间，还有一种 **功能测试**，执行了一段代码的一个良好定义的子集。它们依赖于模拟和桩来将相关的类而不是一个单独的类或方法隔离出来。举例来说，图8.9中的控制器规则文件如使用**get**和**post**方法来提交URI到应用，这意味着它们依赖于路由子系统的正常工作，以便将这些调用发送到合适的控制器方法（你可以自己看看临时移除`config/routes.rb`中的**resources :movies**一行然后试着

重新运行控制器规则文件）。无论如何，通过对正常情况下会与数据库通信的模型方法**find_in_tmdb**建立桩模块，使控制器规则文件仍然与数据库相隔离。

换句话说，优秀质量保证需要好的覆盖率以及三种测试的混合进行。图8.23总结了不同类型测试的相对优点和缺点：

	单元	功能	综合/系统
测试什么	一个方法/类	一些方法/类	大块的系统
Rails示例	模型规则文件	控制器规则文件	Cucumber情景
偏向的工具	RSpec	RSpec	Cucumber
运行时间	非常快	快	慢
错误定位	非常好	中等的	一般
覆盖	非常好	中等的	一般
使用模仿和桩	非常多	中等的	很少/不使用

图 8.23: 单元测试、功能测试、集成测试及全系统测试的不同点总结

小结：

- 静态和动态地衡量覆盖率，包括代码与测试比（由`rake stats`生成）、C0覆盖率（由SimpleCov生成），以及C1—C2覆盖率，是量化测试组件测试程序中不同代码路径广泛程度的指标。
- 单元、功能及集成测试在运行时间、解析度（错误定位能力）、执行不同代码路径的能力以及“智能检查”整个应用的能力上有所不同。三种测试对软件质量保证都非常重要。

自我检查8.8.1. 为什么高测试覆盖率并不是一定意味着一个良好测试的应用?
◇ 覆盖率并不意味着测试有质量。然而，低覆盖率肯定意味着一个没有良好测试的应用。

自我检查8.8.2. C0代码覆盖率和代码与测试比之间有什么不同?
◇ C0覆盖率是一个测试组件执行的所有语句的比例的 **动态的** 测试。代码与测试比是比较总代码行数与总测试行数的一个 **静态的** 测试。

自我检查8.8.3. 为什么像第7章描述的那样在Cucumber情景中大量地使用**模拟**或**桩**不是好方法?
◇ Cucumber是一个全系统测试和验收测试的工具。这样的测试特别地倾向于运行整个系统，而不是它的某些“假的”部分，就像我们在本章中使用嫁接一样。（然而，如果“整个系统”包括与我们不能控制的外部服务的交互，例如与TMDb的交互，为了测试我们确实需要一种制造“假的”行为的方法。这是练习8.1题目的主题）。

8.9 其它测试方法和术语

软件测试的范畴和软件工程一样宽广，一样历时长久，并且有它专门的研究领域。它的技术包括从证明覆盖率相关的形式体系，到选择创建哪种测试的经验技术，以及定向随机测试。取决于一个组织的“测试文化”，你可能听到一些与本章中不同的术语。Ammann和Offutt的 **Introduction to Software Testing** （Ammann and

Offutt 2008）是这方面最好综合参考书之一。他们的方法是将代码片断分成 **基本块**，每一块从头到尾执行而没有分支，然后把基本块填入一张图里，代码中的分支对应着图中有多个出边的节点。我们可以把测试想像成“覆盖这个图”的问题：每一个测试样例都通过了图中它访问的节点，测试组件执行完后所有节点被访问的比例就是测试覆盖率。Ammann和Offutt继续分析了许多软件的结构方面特点，包括如何抽取这样的图，以及提出测量获得这些图的覆盖率的系统自动化技术。

一个从这种方法中引出的观点是：上一节描述的测试级别指的是 **控制流覆盖**，因为它们只关心特定的代码部分是否被执行过。另一种重要的覆盖率标准是 **定义—使用覆盖率** 或 **DU—覆盖率**：给定一个程序中的变量**x**，如果考虑**x**每次被赋值及被使用出现的地方，则DU—覆盖率是指在一个测试组件下，所有**x**被定义的点和**x**被使用的点的 **二元组** 被测试到的比例是多少。这个条件比所有路径覆盖的条件弱一些，但能找到控制流覆盖单独测试时漏掉的错误。

另一类测试术语 **白盒测试**（也叫 **透明测试**）的设计反映对软件真实的，而不是外部指标所蕴涵的实现的认知；另一个区别于它的测试术语是 **黑盒测试**，其设计仅仅基于软件的外部指标。举个例子，哈希表的外部指标可能仅仅规定了什么时候存储一组键/值对，以及随后通过键读取值时，我们要能获得存储的值。一个黑盒测试会指定一个随机的键/值对集合来测试其行为。对应地，一个白盒测试会利用该哈希函数的知识，创建有很多哈希碰撞的最坏情况的测试数据。类似地，白盒测试可能关注边界值——更可能用到不同部分代码的参数的值。在我们的TMDb例子中，可以看到`ruby-tmdb`工具包在密钥空白而无效的情况和密钥无效但不是空白的情况时表示不同，所以即使所有空白的密钥都是无效的，也需要分开来测试代码路径。相反地，我们用任何代表性的非空白且无效的密钥测试“非空但无效”的代码路径——从几个不同的非空且无效的密钥的测试中都不会获得新的信息。

由Ammann和Offutt发明的 **变异测试**，是一种对程序源代码进行微小但符合语法的改动的自动测试技术，例如将**a+b**替换为**a-b**，或将**if (c)**替换为**if (!c)**。很多这样的改变至少会引起一项测试失败，所以一个 **没有** 导致任何测试失败的变异，意味着测试覆盖度不够或者这是一个奇怪的程序。Ruby工具包Heckle结合RSepc来执行变异测试，但不幸的是，它与本书中使用的Ruby1.9版本不兼容。在我们写作本书的时候，还没有一个与Ruby1.9整合的变异测试工具出现。考虑到Ruby社区对测试的强调重视，这种情况很快将会改变。

模糊测试 包括扔给应用随机的数据，然后看它会出现什么问题。约1/4的常见的Unix实用工具会在模糊测试中崩溃，微软估计约有20∼25%的程序错误是通过这种方法发现的。**普通模糊测试（dumb fuzzing）** 工具能生成完整的随机数据，而 **智能模糊测试（smart fuzzing）** 工具能够考虑到应用的结构。例如，Rails应用的智能模糊测试可能包括会把提交的表单中，或是嵌入到网页视图的URI中的变量及值进行随机化，创建语法上有效但可能引起错误的URI。用于SaaS的smart fuzzing也可以包括像SQL注入之类的跨网站的脚本攻击，我们将在第12章中具体讨论。Tarantula[11]（一个抓取网站的模糊爬虫）是一个用来对Rails应用进行模糊测试的Ruby工具包。

其它测试方法的小结：
我们可以把测试想像成“覆盖一个”可能的软件行为的“图”。这个图可能代表控制流（基本的块覆盖）、变量赋值和调用（DU—覆盖）、一个随机输入空间（模糊测试），或是一个对代码中特殊错误的可能的测试空间（变异测试）。这些不同的方法是互补的并且趋向于发现不同类型的错误。

自我检查8.9.1. 微软的Zune音乐播放器有一个声名狼藉的错误，它使Zune在2008年12月31日全部“锁定”。后面的分析显示这个错误是每个闰年的

最后一天引起的。怎样的测试——黑盒测试、透明测试（参见1.6节）变异测试、还是模糊测试——最有可能发现这个错误?

◇ 一个针对闰年特殊代码路径的透明测试可能是最有效的。模糊测试也有可能有效：因为这个错误在大约每1460天会发生一次，经过数千次的模糊测试可能就能发现它。

8.10 计划文档的视角

项目经理从需求计划阶段获得软件需求说明书并把它分成独立的程序单元。开发者编写每个单元的代码，然后执行单元测试来确保它们正常工作。在很多组织中，质量保障人员执行剩下的更高层次的测试，例如模块测试、集成测试、系统测试以及验收测试。

将单元集成起来并执行集成测试有三个途径：

(1) 自顶而下，从树结构的顶部开始，依次展示单元之间的依赖关系。自顶而下方法的优势在于你可以很快使高层的功能工作起来，例如用户界面，这允许涉众及时提供应用的反馈以帮助做出改进。此途径的问题意味着在初始联合时需要创建很多桩来使应用勉强运行。

(2) **自底而下**，从依赖树的底部开始逐步向上建立。这不需要桩，因为你可以集成所需要的所有模块。可惜的是，编写完所有代码并将它们都集成起来之前，你不会获得任何关于应用运行起来的信息。

(3) **两端同时开始**，不出意外地，试着通过同时从两端集成从而能够把两方面的好处结合起来。这样，你可以通过选择性地从底部集成一些单元来试着减少桩的数量，并通过选择性地从顶部开始集成来使用户界面更早地运作起来。

在集成测试之后，QA测试员要做的是系统测试，使得整个应用能正常工作。这是在展示给客户让他们来体验前的最后一步。记住系统测试不仅测试如性能之类的非功能性需求，同时也测试SRS中的功能性需求特性。

一个关于计划文档模式的问题是，如何决定测试什么时候完成。典型地，一个组织在将产品交付给客户前会要求达到一个测试覆盖标准。例如语句覆盖（所有的声明都被执行了至少一次），或所有的用户输入的地方都用正确的输入和错误的输入测试过。

在计划文档过程中，客户的最后一步测试是在他们的环境中使用产品并决定是否接受产品。这意味着，目标确认实际可行而不仅仅是验证可行。在敏捷开发中，客户参与到了应用原型试验的早期过程，所以在敏捷周期中进行验收测试之前没有单独的系统测试。

正如你对计划文档过程期望的那样，文档在测试中扮演着很重要的角色。图8.24给出了一个基于IEEE 829-2008标准的测试计划的提纲。

测试对软件工程来说是根本的，引用另一位图灵奖获得者的话：

> 程序测试可以用来展示错误的存在，但不能展示它们的不存在！
>
> ——Edsger W. Dijkstra

Edsger W. Dijkstra（1930——2002）1972年因为在程序开发语言方面的重要贡献获图灵奖。

因此，有很多的研究工作在探验证方法而不仅仅是测试方法。总的来说，这些技术是都 **形式化分析**。一般的策略是从一个形式化的规范开始，来证明代码的行为与规则的行为相符。这些是由人或计算机完成的数学证明。常用的两种方法是 **自动定理证明** 或 **模型检测**。定理证明使用了一组推理规则及一组逻辑公理来从头开始完

综合测试计划提纲
1. 简介
1.1. 文档标识符
1.2. 范围
1.3. 参考
1.4. 系统概述及主要功能
1.5. 测试综述
1.5.1 组织
1.5.2 综合测试计划
1.5.3 完整性水平测试
1.5.4 资源小结
1.5.5 责任
1.5.6 工具、技术、方法、以及试题

2. 综合测试计划细节
2.1. 包括定义及测试级别的测试步骤
2.1.1 步骤：管理
2.1.1.1 活动：测试成本管理
2.1.2 步骤：获取
2.1.2.1 活动：获取支撑测试
2.1.3 步骤：供应
2.1.3.1 活动：计划测试
2.1.4 步骤：部署
2.1.4.1 活动：概念
2.1.4.2 活动：需求
2.1.4.3 活动：设计
2.1.4.4 活动：实现
2.1.4.5 活动：测试
2.1.4.6 活动：安装/检出
2.1.5 步骤：操作
2.1.5.1 活动：操作测试
2.1.6 步骤：维护
2.1.6.1 活动：维护测试
2.2. 测试文档需求
2.3. 测试管理需求
2.4. 测试报告需求

3. 常规
3.1. 术语表
3.2. 文档变更过程及历史

图 8.24: 符合IEEE 829-2008标准的集成测试计划提纲

成证明。模型检测通过彻底地搜索所有系统在运行时可能进入的状态来检验选择的属性。

因为形式化分析是计算密集型的，它们一般只在检查修复错误的代价很高、功能很难测试、验证的对象并不太大的情况下使用。这样的例子包括像网络协议这样的硬件重要组成部分或医疗设备这样的对安全性要求极高的软件系统。为了让形式化分析能实际工作，软件的规模必须是很有限的：迄今为止最大的形式化验证软件是一个代码少于10000行操作系统内核，它的验证花费大约为每行代码500美元（Klein et al. 2010）。

因此，形式化分析 不 适合改变频繁的功能复杂的软件，一般来说，SaaS就是这类软件。

参考下形式化分析的花费，**NASA**花费每年35000000美元来维护420000行用于航天飞机的项目代码[12]，大约是每年每行代码80美元。

任务	计划文档模式	敏捷模式
测试计划及文档	软件测试文档例如IEEE 829-2008标准	用户故事
编写代码和测试的顺序	1、编写代码部分 2、单元测试 3、模块测试 4、集成测试 5、系统测试 6、验收测试	1、验收测试 2、集成测试 3、模块测试 4、单元测试 5、编写代码部分
测试员	开发者完成单元测试；QA测试员完成模块、集成、系统和验收测试	开发者
测试什么时候停止	公司政策（例如，语句覆盖，快乐和悲伤用户输入）	所有测试通过（变成绿色

图 8.25: 计划文档模式和敏捷方法中测试任务的关系

小结：
测试及形式化分析能减少设计中的错误风险。

- 与BDD/TDD不同，计划文档过程在编写测试之前从编写代码开始。
- 然后开发者执行单元测试。
- 尤其是在大的项目中，不同的人执行不同级别的测试。将单元测试放在一起的集成方法有自顶而下、自底而上、两端同时开始三种。
- 测试员会做一个独立的系统测试，以确保产品在提交给客户进行最后的验收测试前通过了功能性的和非功能性的需求。
- **形式化分析** 依赖于形式化的规范，以及自动化证明或彻底的状态搜索来比测试能做到的更多的验证，但执行这种方法太昂贵了，如今只能应用在软件或硬件的一些小的、稳定的、关键的部分。
- 图8.25展示了计划文档模式与敏捷过程的结果不同的测试任务。

自我检查8.10.1. 比较和对照自顶而下、自底而上、两端同时开始的集成策略。
◇ 自顶而下的方法需要桩来执行测试，但它可以早点让涉众感受到应用是如何工作的。自底而上的方法不需要桩，但可能需要在涉众能看到它工作之前写好所有的东西。两端同时开始的三明治式集成从两端开始以综合上述两者的好处。

8.11 谬论和陷阱

谬论：100%的测试覆盖率的全部通过的测试意味着没有错误。

这个陈述不成立的原因有很多。完整的测试覆盖并不意味着每个独立的测试都有质量。同样地，一些错误可能需要通过一定的值作为方法参数（例如引起一个除0错误），因此控制流测试经常不能检查出这个错误。在应用与类似TMDb之类的服务间的交互还可能有错误，屏蔽这样的服务而执行的本地测试可能掩盖这些错误。

陷阱：教条主义地坚持100%的测试覆盖率以及交付之前测试要全部通过（绿色）。

如上所述，100%的测试覆盖率在比C1标准高的层次不仅难以达到，而且即使达到了也不能保证没有错误。测试覆盖是一个估算测试组合整体综合性的有用工具，但更高可信度要求各种各样的测试方法——集成测试和单元测试，模糊测试和人工构造测试样例，定义使用覆盖率以及控制流覆盖率，变异测试来检查测试策略中额外的盲点，等等。实际上，第12章将讨论诸如安全性和性能之类的操作问题，这将要求优于本章中描述的面向正确性的测试的额外的测试策略。

谬论：对应用中确信正确的代码你不需要测试。

坚持100%的覆盖率可能会适得其反，另一个极端也是。在生产式系统（非注释代码被所有类型的测试分割开来）中 **代码测试比** 一般比1小，也就是说，测试的代码比应用的代码要多。在一个极端的例子中，一个包括Rails的SQLite数据库的测试代码有应用代码的超过1200倍之多[13]，因为它有很多的使用方法，并且它在很多不同类型的系统上都得工作正常！虽然仍然有关于测量代码测试比率有多大用处的争论，但是鉴于Ruby的高效以及它是优秀的精简测试代码的工具，`rake stats`测出的比例在0.2～0.5之间是比较合理的目标。

陷阱：过度地依赖于某一种测试（单元、功能、集成）。

尽管100%的单元测试测试其覆盖率也不能提供不同类之间交互行为的信息，你仍然需要创建测试，用来执行不同类之间的交互行为（功能性测试或模块测试），以及执行应用中涉及很多类的并在很多地方改变状态的完整的路径测试（集成测试）。相反地，集成测试只涉及所有可能的应用路径的很小一部分，因此使用了每个方法中的很少一部分行为，所以它们不是好的单元测试覆盖的替代品，好的单元测试能确保较低级别的代码正确工作。在谷歌和其它地方使用的一个普遍的法则（Whittaker et al. 2012）是“70—20—10”：70%集中于短的单元的测试，20%涉及多个类的功能性测试，10%的完整测试或集成测试。

陷阱：由于过度使用桩而处在试验阶段的集成点。

模拟和桩能带来很多好处，但它们也会在集成点，即类或模块交互的地方——隐藏潜在的问题。假设**Movie**与另一个类**Moviegoer**有交互，但为了对**Movie**做单元测试，调用**Moviegoer**的方法被屏蔽了，反之亦然。因为桩是写的相关类的“假”的行为，我们不再清楚**Movie**是否“知道如何正确地”与**Moviegoer**“交流”。如果功能测试和集成测试有好的覆盖率，并且没有隔离跨越类边界的所有调用，可以避免这个陷阱。

陷阱：在写完代码之后写测试，而不是之前。

从测试一段代码的观点出发想想“我们想要的代码”，容易得到可测试的代码。这看起来像明显的无意义的重复，但在你不考虑测试，并且试着把代码写好之后才编写测试时，你会惊讶地发现你得到了模拟的残骸（参见下一个陷阱）。

另外，第1章讲的传统的瀑布模型生命周期，测试在代码完成之后进行，但对SaaS应用可能在“公开测试”阶段停留数月，没有人会坚持测试应当只在测试阶段进行。不论是为了修正错误还是创建新的方法，提前编写测试都能躲开这个陷阱。

陷阱：连环残骸模拟。

已有的模拟可以帮助从其协作者中隔离你的测试，但它协作部分的协作部分呢？假设**Movie**有一个**pics**的属性返回一个与电影相关的图片列表，每一个都是一个拥有**format**属性的**Picture**对象。你在尝试模拟测试中使用的**Movie**对象时，你意识到传递**Movie**的方法也将会调用**pics**的方法，所以你会发现你在做这样的事情：

代码文件：ch_tdd/code/mock_trainwreck.rb

```
movie = mock('Movie', :pics => [mock('Picture', :format => 'gif')])
Movie.count_pics(movie).should == 1
```

这种现象叫做 **连环残骸模拟**，表示测试（**count_pics**）中的方法拥有对**Picture**内部结构过多的知识的现象。在第9章和第11章中将看到一组额外的指导来帮助你检测和解决这样的 **代码味道**。

陷阱：不经意地创建了关于规则运行顺序的依赖性，例如使用了before(:all)。

如果你指定了在整组测试实例中仅执行一次的动作，你就在这些测试实例中不经意地引入了依赖性。例如，如果一个**before :all**块给变量赋值，测试实例A改变了变量的值，如果A通常是在测试实例B之前执行的，B就依赖于A。那么，B的将来的行为就有可能决定于与B是否先被执行。由于`autotest`优先运行与最近更改的代码相关的测试，这处情况是可能发生的。因此最好的做法是在任何可能的时候使用**before :each**及**after :each**。

8.12 结束语：TDD与传统的调试方法

本章使用了RSpec来开发了一个使用TDD单元测试的方法。尽管开始时TDD可能比较陌生，但很多尝试使用它的人很快发现他们已经用过TDD需要的单元测试技术，但以一种不同的工作流程来使用。通常，一个传统的开发者会编写一些代码，假设它已经能工作，通过运行整个应用来测试它，并修改错误。正如一个MIT程序员1968年在第一届软件工程大会上惋惜的那样：“我们就像莱特兄弟建造飞机一样建立了一个系统——创建了所有东西，把它推离悬崖，让它崩溃，并重新开始。”

当错误发生时，如果检查代码不能揭露错误，一个传统的开发者接下来会试着在可疑的区域插入打印语句，打印出相关变量的值或者指示出执行分支的哪一条路径。而一个TDD开发者会写使用**should**结构构造断言语句来替代。

如果还不能找到错误，传统的开发者会将代码隔离出一部分，小心地设置条件来跳过他不关心的方法调用，或改变变量的值来使代码执行到可能有错误的路径上。例如，他可能利用调试器设置一个断点，在越过断点继续运行之前进行人工检查或操纵变量的值。相反地，TDD开发者会将可疑的代码路径使用桩和模拟来孤立

起来，来控制某个特定的方法被调用时发生的情况并决定执行哪条分支结构。

到现在为止，一个传统的开发者绝对相信自己能够找到错误并且不需要重复这个乏味的手工过程，但是这经常被证明是错的。TDD开发者把每一个行为都隔离在了它自己的规则文件中，所以重复这个过程仅仅是重新运行规则，它甚至可以用`autotest`来自动执行。

换句话说，如果我们首先写了代码然后再改正错误，我们最终将用到与TDD需要的同样的方法，但是却更低效、更需要人工，所以生产力会更低。

但如果使用TDD，那么错误可以在写代码的同时立刻被发现。如果代码第一次正常工作，那么使用TDD工具能给我们一个回归测试来发现未来可能蔓延到这部分的错误。

8.13 深入学习

- 谷歌引入并且规模化了本章中介绍的技术，并将其灌输到公司的测试文化中。**How Google Tests Software**（Whittaker et al. 2012）是对其一个难得的讲解。这一测试文化广泛被其竞争对手学习。
- 在线RSpec文档[14]给了我们在高级测试情景中使用的完整的细节及附加的功能。
- **The RSpec Book**（Chelimsky et al. 2010）是RSpec的一本权威参考书，它包括功能示例、原理介绍以及远远超过这里介绍的典范做法。

P. Ammann and J. Offutt. **Introduction to Software Testing**. Cambridge University Press, 2008. ISBN 0521880386. URL `http://www.amazon.com/Introduction-Software-Testing-Paul-Ammann/dp/0521880386`.

D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. **The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends (The Facets of Ruby Series)**. Pragmatic Bookshelf, 2010. ISBN 1934356379. URL `http://www.amazon.com/RSpec-Book-Behaviour-Development-Cucumber/dp/1934356379`.

M. Feathers. **Working Effectively with Legacy Code**. Prentice Hall, 2004. ISBN 9780131177055. URL `http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052`.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. **Communications of the ACM (CACM)**, 53(6):107–115, June 2010.

J. A. Whittaker, J. Arbon, and J. Carollo. **How Google Tests Software**. Addison-Wesley Professional, 2012. ISBN 0321803027. URL `http://www.amazon.com/Google-Tests-Software-James-Whittaker/dp/0321803027`.

注释

[1]http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/CaseStudies/MHCPMS/SupportingDocs/MHCPMSCaseStudy.pdf
[2]http://themoviedb.org
[3]http://nokogiri.org
[4]http://en.wikipedia.org/wiki/Y2k
[5]http://jmock.org/getting-started.html
[6]https://github.com/thoughtbot/factory_girl_rails
[7]http://api.themoviedb.org
[8]http://fakeweb.rubyforge.org
[9]http://rspec.info
[10]https://github.com/colszowka/simplecov
[11]https://github.com/relevance/tarantula
[12]http://www.fastcompany.com/magazine/06/writestuff.html
[13]http://www.sqlite.org/testing.html
[14]http://rspec.info

8.14 建议的项目

练习8.1. 完成第7章开始的Cucumber情景中的happy path来从TDMb中检索电影。为了保持情景的**独立**于实际的TMDb服务，你可能需要下载使用`FakeWeb`工具包来将对TMDb服务的调用“隔离”开。

练习8.2. 我们现在的控制器文档只覆盖了测试**search_tmdb**的happy path：**find_in_tmdb**现在会引出一个**InvalidKeyError**，控制器文档是不完整的，因为我们没有测试在那个情景下控制器方法做了什么。使用TDD来创建规则文件和控制器代码来处理这个情形。

练习8.3. 在8.3节中，我们创建了方法**find_in_tmdb**的桩来将控制器的测试从其它类隔离出来。这种桩在Java中如何处理？

练习8.4. 其于以下规则文件，**Foo**的实例必须响应哪个方法才能通过测试？

代码文件：ch_tdd/code/ex/rspec_matcher.rb

```
1  require 'foo'
2  describe Foo do
3    describe "a new foo" do
4      before :each do ; @foo = Foo.new ; end
5      it "should be a pain in the butt" do
6        @foo.should be_a_pain_in_the_butt
7      end
8      it "should be awesome" do
9        @foo.should be_awesome
10     end
11     it "should not be nil" do
12       @foo.should_not  be_nil
13     end
14     it "should not be the empty string" do
15       @foo.should_not == ""
16     end
17   end
18 end
```

练习8.5. （难）在上一个练习中，解释为什么将第15行重写为下面的一行会导致不正确的测试行为？

代码文件：ch_tdd/code/ex/should_not_equal.rb

```
@foo.should != ""
```

练习8.6. 在第7章中，我们在Rotten Potatoes的首页上创建了一个“Find in TDMb（在TMDb中搜索）”的按钮来将数据发送给**search_tmdb**，但我们从来没有写验证这个按钮发送的正确动作的规则。请用RSpec的**route_to**断言适配器来写这个规则，并把它添加到控制器规则中。
提示：由于这个路由与基本的CRUD动作不对应，可能不能使用嵌入的RESTful URI助手来指定这个路线，但可以使用**:controller**以及**:action**参数来**route_to**（路由到）到明确指定的动作

练习8.7. 通过在`movies_controller_spec.rb`中创建额外的规则来增加`movies_controller.rb`的C0覆盖率到100%。

练习8.8. 写一个测试样例序列，能够最终判断电子邮件是有效还是无效的。从最简单可能的事情开始，并在添加更多功能的时候保持回归性。

练习8.9. 在1999年，耗资1.65亿美元的火星气候探测人造飞船在进入火星大气时由于负责推进器软件的小组将推力按公制（SI）单位表示，而另一个负责推进器软件不同部分的小组将它们按英制单位表示导致烧毁。哪一种正确性测试——单元测试、功能测试、还是集成测试——对发现这个错误是有效的。

练习8.10. （讨论）描述形式化分析在复杂软件开发中扮演的角色，并按测试的验证与确认技术比较它们的用途。

练习8.11. 比较并对照集成测试的策略，包括自顶而下、自底而上，以及两端同时开始（三明治式）集成。

9 软件维护：使用重构和敏捷方法改进遗留软件

Butler Lampson（1943——）是帕洛阿尔托研究中心（Xerox PARC）的技术领导人，它在19世纪70年代的全盛时期，发明了图形用户界面、面向对象编程、激光打印机以及以太网。三位PARC的研究员因为他们在那儿的工作最终获得图灵奖。Lampson在1994年因为在分布式个人计算环境——包括工作站、网络、操作系统、编程系统、显示、安全、文档打印等技术——的开发与实现方面的贡献获得图灵奖。

可能并不存在构建系统的“最好的”方法，甚至某些主要部分都没有。避免选择一个糟糕的方法，并且在不同部分之间有明确的责任划分，是非常重要的。

——Butler Lampson，**Hints for Computer System Design**，1983

9.1 什么是代码“遗留”，敏捷开发对其有什么作用？ **278**
9.2 探索遗留代码库 **280**
9.3 使用鉴定测试来创建基准事实 **283**
9.4 注释 **286**
9.5 度量、代码味道及SOFA **288**
9.6 方法级别的重构：使用嫁接替代依赖 **291**
9.7 计划文档的视角 **296**
9.8 谬论和陷阱 **300**
9.9 结束语：持续的重构 **301**
9.10 深入学习 **302**
9.11 建议的项目 **304**

重要概念

就像鲨鱼必须不断运动来求生一样，软件必须不断改变来保持它的可行性与活力。本章的重要概念是，敏捷开发对软件维护和改进遗留代码都是一个好的方法，以及 **重构** 在所有开发过程中来保持代码的可维护性都是必要的。

当编写代码时， **软件度量（software metrics）** 和 **代码味道（code smells）** 能够识别难以阅读的代码。通过 **重构** 进行的转换能够提高软件度量并且消除代码味道。方法应该是简短的，只做一件事情，有少量的参数，并且维护一个单层次的抽象（SOFA）。

使用敏捷生命周期来改进遗留代码：

- 理解 **转变点** 的代码，在这里可以做出合理的修改。阅读并改进注释是一个理解代码的途径。
- 在阅读测试、设计文档，以及检查代码等过程中探索每个涉众的观点以确定它如何工作。
- 在修改代码 **之前** 编写 **鉴定测试** 来提高测试覆盖率。

对计划文档生命模式的生命周期来说：

- 在维护和评估 **变更请求** 的代价期间， **维护经理** 管理该项目。
- 使用成本效益分析， **变更控制委员会** 将变更请求分类。
- 和敏捷开发一样，维护时需要通过 **回归测试** 来确保新的版本正常工作，以及通过 **重构** 来使代码易于维护。

令人惊讶的是，敏捷过程与计划文档生命周期的维护阶段的很多需求都有对应。

敏捷和计划文档过程有相同的维护目标以及很多相同的技术，但敏捷开发强调持续增量来重构而不是重新编写所有代码。

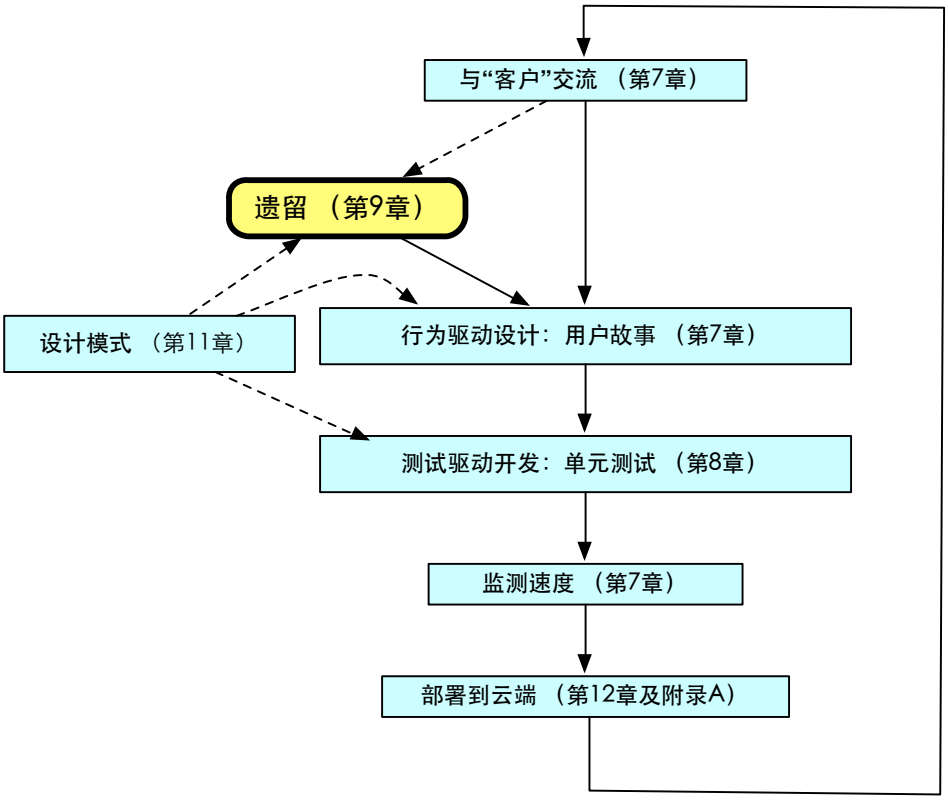
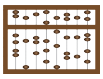


图 9.1：敏捷软件生命周期及它与本章节的关系。本章涵盖了敏捷技术对改进遗留应用程序的作用

9.1 什么是代码“遗留”，敏捷开发对其有什么作用？

> 持续的改进—[软件]系统必须进行持续的改进，否则它们会逐渐变得不再符合需求
>
> ——Lehman的软件演化第一法则



如第1章所述，由于 **仍旧满足用户需求**， **遗留代码** 仍然在被使用，即使它的设计实现已经过时或鲜为人知。本章将介绍如何将敏捷技术应用于增加或修改遗留代码。图9.1在整个敏捷生命周期的背景下强调了这个主题。

可维护性 是指一个产品被改善的容易程度。在软件工程中，维护包括以下四个方面（Lientz et al. 1978）：

- 纠错性维护：修复缺点及错误。
- 完善性维护：扩展软件功能以符合新的用户需求。
- 适应性维护：复制到一个新的运行环境工作，即使没有加入新的功能；例如，适应产品托管环境的改变。
- 预防性维护：改进软件的结构以增强未来的可维护性。

在遗留代码上进行这些种类的维护是一种需要在实践中学到的技能：我们将提供可以使用的多种技术，但是并没有提供捷径。这就是说，所有维护工作的关键部分之一是 **重构** ，一个改变（有可能改进）代码结构而不改变软件功能的过程。本章的主题是 **持续的重构能改善可维护性** 。因此，本章的一大部分将关注重构。

软件的任何片断，不管设计得如何良好，最终都会慢慢超出它原本的设计范围。这个过程导致了维护性的挑战，其中之一就是如何处理遗留代码。由于原来的设计者离开很长时间，并且软件累积了许多没有被现在的设计文档涵盖的 **补丁** ，从而使生成的代码无法得到很好的理解时。这就被一些开发者称之为“遗留”。在有经验的从业者中流传着一个更无奈的观点，就是这样的文档怎么样都不会有太大帮助。一旦开始开发过程，必要的设计变更会引起系统渐渐远离原来的设计文档，而文档并未得到更新。在这样的情况下，开发者必须依赖如图9.2所示的 **非正式的** 设计文档。

高可读性单元，功能和集成测试（第8章）	Git提交日志文件（第10章）
低精度（lo-fi）UI模型和Cucumber类型的用户故事（第7章）	注释和RDoc类型的嵌入到代码中的文档（9.4节）
有关应用架构的白板草图照片，类关系等等（9.2节）	存档邮件、wiki/博客、笔记、代码和设计评审的视频记录。例如Campfire[1]或Basecamp[2]（第10章）

图 9.2: 由于最新格式的设计文档是有价值的，敏捷开发建议我们应该更多地注重与工作代码“更近”的文档

我们如何改进没有好的文档的遗留代码呢？正如Michael Feathers在 **Working Effectively With Legacy Code** （Feathers 2004）中所写的那样，有两种方法来改变已有的软件： **编辑并祈祷** 或者是 **覆盖并修改** 。遗憾的是，第一种方法太常见了：熟悉软件中需要更改的一小部分，编辑对应的代码，手工探索寻找这是否破坏了任何东西（尽管这很难确定），然后部署并祈祷它不会产生任何问题。

与此相反， **覆盖并修改** 要求创建能够覆盖你将要更新的代码的测试（如果它们不存在），使用它们作为一个“安全系统”来检查由于修改而引起的非计划的行为改变，就像回归测试发现之前能工作的代码在现在出现的错误一样。覆盖和修改的观点引出了Feather关于“遗留代码”的更精确的定义： **不论谁在何时编写，缺乏足够的测试，导致修改起来让人没有自信的代码。** 我们将在后面使用这个概念。换句话说，三个月前你在不同的项目中编写，现在不得不重新阅读和修改的代码，也可以当作遗留代码。

幸运的是，我们学的开发新软件的敏捷技术同样能帮助我们处理遗留代码。事实上，理解并改进遗留软件的任务，可以看做是长时间内“拥抱改变”的例子。对于经过严格测试的结构良好的软件，可以使用BDD和TDD通过微小但是有信心的步骤来增加额外的功能。对于结构不好或正在测试的代码，需要通过四个步骤“辅助”我们自己进入到期望的情况中：

(1) 确认 **改变点** ，即遗留系统中你将要做出改变的地方。9.2节描述了一些有用的探索技术，并介绍了一种统一建模语言（UML）表示应用中主要类之间的关系。

(2) 如果有必要，添加 **鉴定测试** 来了解代码现在是如何工作的，在做出任何改变之前建立一个基础的“基准情况”。9.3节介绍了如何使用你熟悉的工具来完成这个工作。

(3) 决定改变点是否需要 **重构** 来使已有的代码更具可测试性或更适应需求的改

变。举例来说，消除使代码难以测试的依赖性。9.6节介绍了一些来自许多重构分类的广泛应用的技术，它们已经发展成为敏捷开发的一部分。

(4) 一旦改变点周围的代码都被重构并且都被测试所覆盖，就可以做出需求的改变，并使用新建的测试作为回归测试，还可像第7章和第8章中那样添加新增代码的测试。

敏捷开发对遗留代码的作用小结：

- 可维护性是指软件能被改进、适应新的运行环境、修补以及被改进以使未来的维护更容易的程度。软件维护的一个主要部分是重构，是敏捷过程中改善软件结构来使它更具可维护性的核心部分。因此持续的重构能够增强软件的可维护性。
- 对遗留代码的工作首先要探查并理解代码库，尤其是要理解你期望做出改变的 **改变点** 附近的代码。
- 如果没有好的测试覆盖，我们就缺乏信心来确认重构或提高代码质量都可以保护已有的行为不被改变。因此，我们采用Feather的定义——“遗留代码是没有测试的代码”——并在重构或改进遗留代码之前在需要加强测试覆盖的地方创建鉴定测试。

■ 详细阐述：嵌入程序中的文档

RDoc是一个在Ruby代码中寻找特殊格式的注释并生成代码程序文档的文档系统。它与JavaDoc类似并受其启发。RDoc语言很容易通过the Ruby Programming wikibook（Ruby编程维基）[3]的例子来学习。RDoc默认的HTML输出，可以在Rails documentation（文档）[4]中看到。在你探索并理解遗留代码的时候考虑添加RDoc文档，在Rails应用的根目录下运行`rdoc .`来在每一个`.rb`文件的目录中生成RDoc文档，`rdoc --help`显示了其它选项，Rails应用目录中`rake -T doc`列出了其它文档相关的Rake任务。

自我检查9.1.1. 为什么很多软件工程师相信，在改进遗留代码的时候，好的测试覆盖比设计详细的文档或结构良好的代码都重要?
◇ 没有测试，就不能确认对遗留代码进行更改时能够保留它已有的行为。

9.2 探索遗留代码库

> 如果选择了正确的数据结构并经过很好的组织，那么算法将几乎永远是不言而喻的。数据结构，而不是算法，是编程的核心。
>
> ——Rob Pike

探索代码的目标是从用户和开发者的视角来理解应用。使用何种特定的技术依赖于当前的目标：

- 你接触了一个全新的项目，需要理解应用的整体结构，把你的发现过程随时写下来，这样别人就不需要重复你的发现过程了。

代码文件：ch_legacy/code/setup_development_db.txt

```
# on production computer:
RAILS_ENV=production rake db:schema:dump
RAILS_ENV=production rake db:fixtures:extract
# copy db/schema.rb and test/fixtures/*.yml to development computer
# then, on development computer:
rake db:create        # uses RAILS_ENV=development by default
rake db:schema:load
rake db:fixtures:load
```

图 9.3: 可以创建一个与产品数据库有相同架构的空的开发数据库，然后用支架（fixtures）填充它。尽管第8章警告了不要滥用支架，但是在这种情况下可以使用它们来从生产环境中复制已知行为到开发环境中

- 你需要理解被要求做的某个改动会对哪些部分产生影响。
- 你正在寻找需要优化的地方，因为你正处在移植或更新一个遗留代码库的过程中。

就像在第2章中探究SaaS结构一样，使用高度作为类比，可以遵循一些“由外而内”的步骤从多个层次上理解一个遗留应用的结构：

(1) 检出一个应用的分支，在开发环境中运行该应用程序。

(2) 学习并复制用户故事，如果有必要，和涉众一起工作。

(3) 检查数据库模式及大部分主要类之间的关系。

(4) 浏览所有代码来量化代码质量和测试覆盖率。

由于在正在运行的应用上的操作可能危及用户数据或影响用户体验，所以第一步是让应用在开发环境或分隔的环境中运行，这样干扰它的操作就不会引起客户的不满意。创建一个你从不打算提交回去的代码库的 **草稿分支**，这样它就可以用来做实验。如果没有一个现成的供开发用的数据库，那么创建一个。一个简单的方法是在产品数据库不太大的情况下复制一个，从而避免数不清的陷阱：

- 应用可能有很多反映表格行的关系，诸如“has-many”或“belongs-to”关系。如果不知道关于这些关系的细节，你可能会创建一个无效的数据子集。以Rotten Potatoes为例，你可能不小心地得到这样一个**视图**，它的`movie_id`和`moviegoer_id`指向不存在的电影或看电影的人。
- 复制数据库消除了由于数据库的实现不同而导致的产品和开发之间不同的行为。在不同数据库中，一些类型的数据，例如日期等，是用不同的方法表示的。
- 副本可以提供你一个实际且有效的数据来进行开发。

如果不能复制产品数据库，或者成功复制了它但在开发过程中使用它会很不灵活，那么可以从真实数据库中提取固定数据[5]然后按照图9.3中的步骤来创建一个开发数据库。

一旦应用在开发环境中运行，应该找一两个有经验的用户来展示他们如何使用该应用，并标出在试用过程中他们期望的改变（Nierstrasz et al. 2009）。一如既往地请他们讨论演示样本；尽管他们的评论经常用用户体验术语表示（“现在我正在添加Mona为一个管理员用户”），如果应用是使用BDD创建的话，这些评论可能反映原

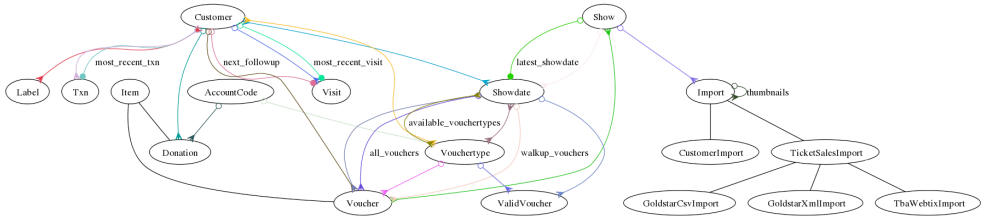


图 9.4: 这个简化的统一建模语言（UML）类图，是由`railroady`工具包自动生成的，展示了Rails应用中一个小型电影院的售票管理、捐赠以及出勤绩效的模块。有箭头的边或环表示类之间的关系：一个Customer类有许多Visits和Vouchers成员（空心圆到箭头），有一个most_recent_visit（开口圆到箭头），并且有且属于多个Labels（箭头到箭头）。普通的边表示继承：Donation和Voucher是Item的子类。（这里所有重要的类都继承于ActiveRecord::Base，但`railroady`只画出了该应用的类。）我们将在第11章中看到其它类型的UML图

始的用户故事，因此也会反映应用的结构。在演示中应经常询问问题，如果可以联系到应用的维护者，让他们也来观看演示。在9.3节将看到这些演示如何构成“基准情况”测试的基础来支撑你的改变。

一旦有了关于应用是如何工作的模糊想法，就应该看一下数据库模式。Fred Brooks、Rob Pike以及很多人都指出了理解数据结构是理解应用逻辑的关键。你可以使用交互的数据库GUI（图形用户界面）来探索它的结构，但你可能会发现`rake db:schema:dump`会更高效，它会用4.2节介绍的迁移DSL来创建包含数据库模式的`db/schema.rb`。其目的是将架构匹配到应用的整体结构。

图9.4展示了一个由`railroady`工具包生成的简化的统一建模语言（UML）类图，它涵盖了最重要的类及它们最重要的属性之间的关系。这个图表最初看起来可能过于庞大，但由于并不是所有的类都有一样重要的角色，你可以识别出“高度相关”的类，它们可能在应用功能的中心位置。例如图9.4，**Customer**和**Voucher**类相互之间连接并且与许多其它的类连接。接下来你就可以识别出数据库模式中与这些类相关的表了。

在熟悉了应用的结构应用程序的框架、最重要的数据结构以及主要的类之后，就可以准备好看代码了。查阅代码的目的是感知它的整体质量、测试覆盖率以及可以体现理解及改进它的难度的其它统计数据。因此，在深入每一个文件之前，运行`rake stats`来获得每个文件代码的总行数以及测试的总行数。这个信息可以告诉你哪一个类最复杂因而有可能是最重要的（高代码量）、最好的测试过的（最好的代码测试比率）、简单的“辅助”类（低代码量）等等，在类图数据库模式辅助下的加深理解。（在本章后续部分，我们将介绍如何利用一些附加的质量度量标准来评估代码，告诉你最麻烦的工作在哪里。）如果已有测试组件，那么运行它们。假设大部分测试通过了，阅读测试来帮助理解原来开发者的意图。然后花一小时（Nierstrasz et al. 2009）检查最重要的类的代码以及你确信你要改进（the **change points**）的部分，现在你应该对它们有一个好的了解了。

遗留代码探索小结：

- 探索代码的目的是从多个涉众的视角理解应用是如何工作的，包括变更请求的用户及创建原始代码的设计者与开发者。
- 阅读测试、阅读设计文档（如果有效的话）、检查代码以及画出或生成UML类图来识别应用中重要的实体（类）之间的关系，有助于代码探索。
- 一旦你成功接触到了在实际生产中演示的应用，下一步就是在通过复制或用支架创建数据库来使它在开发环境中运行，并使测试组件也能在开发环境中运行。

■ 详细阐述：类——责任——协作者（CRC）卡片

CRC卡片（图9.5）是在1989年提出的[6]一种用来帮助面向对象设计的方法。每一个卡片确定一个类，它的职责以及与之交互完成任务的协作类。如这个外部视频[7]所示，一个小组设计的新代码选择了一个用户故事（7.1节）。在每一个故事步骤中，这个小组确定或创建参与了这一步的类的CRC卡片，并确认这些类有必要的职责和协作者来完成该步骤。否则，类或职责的集合可能就不完整，或是类之间的职责划分需要改变。当探索遗留代码时，你可以创建CRC卡片来文档化你看到的这些类，同时遵循通过与其它用户故事相关的模型和视图来处理用户故事步骤的控制器方法流程。

自我检查9.2.1. 即使你没有打算立刻对代码做出任何改变，那么在开发环境中运行应用也很重要的原因是什么？

◇ 一些可能包括的原因有：

(1) 对SaaS来说，已有的测试可能需要访问测试数据库，在实际生产中可能不能访问。

(2) 部分探索任务可能与交互调试器或其它工具的使用有关，这会减慢运行的速度，干扰正在运行的站点。

(3) 部分探索工作可能需要更新数据库中的数据，而对实际的用户数据你不能这样做。

9.3 使用鉴定测试来创建基准事实

如果没有覆盖计划中影响到的代码部分的测试（或测试太少），则需要创建一些测试。在鉴于给定对代码如何工作的理解有限的情况下，你如何来完成这个工作？一种方法是从通过创建 **鉴定测试** 来建立“基准情况”的基础开始：在捕获并记录一段代码的 **实际的**、**当前的** 行为后编写测试，即使这些行为有错误。通过创建能自动**重复**的，模拟现在代码正确运行的测试（参见8.2节），你能确保这些行为在你修改和改进了代码之后仍然一致，就像高级别的回归测试。

通常以一个综合级别的鉴定测试开始是最简单的，例如Cucumber情景，因为它们对应用工作做了最少的假设，并只关注于用户体验。实际上，好的情景基本上使用了“领域语言”，而不是用命令式的步骤（7.9节），来详细地描述用户交互行为，

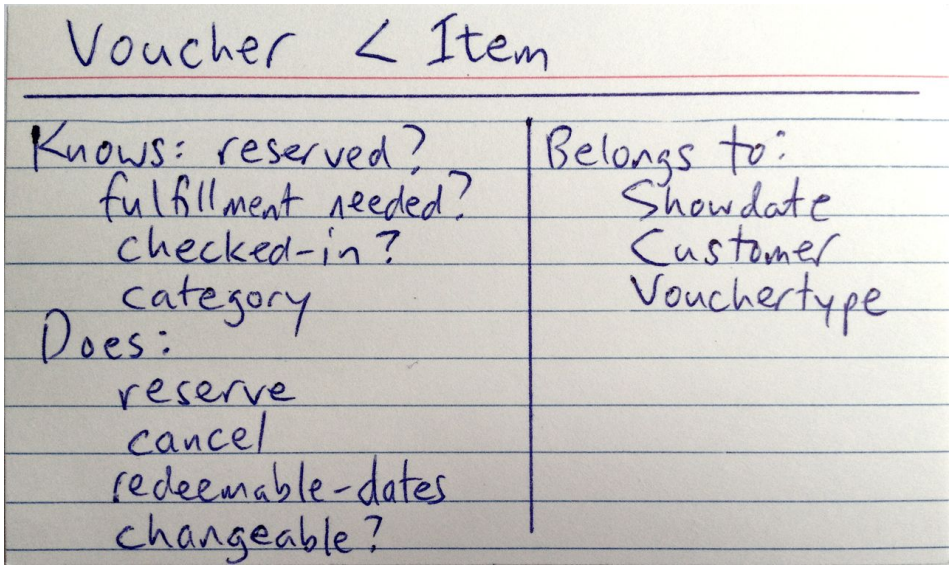


图 9.5: 3英寸宽5英寸长（76mm宽127mm长）的“类——责任——协作者”（CRC）卡片代表了图9.4中的Voucher类。左边一栏表示Voucher类的职责——需要知道的（实例变量）或需要做的（实例方法）事情。由于Ruby实例变量都是通过实例方法来访问的，我们可以通过搜索文件`voucher.rb`中的实例方法和对attr_accessor调用来确定职责。右边一栏表示Voucher的协作类。对Rails应用来说，可通过观察`voucher.rb`中的has_many和belongs_to来确定大部分的协作类

代码文件：ch_legacy/code/date_calculator_bad.rb

```
# WARNING! This code has a bug! See text!
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

图 9.6：这个方法是很难理解，很难测试的，因此根据Feathers对遗留代码的定义，很难去改进。事实上，它包含一个错误——这个例子是微软Zune音乐播放器中一个简化版本的错误，该错误使得任何Zune在2008年12月31日起永久冻结，并且在重启之前，唯一的解决方法是等到2009年1月1日的第一分钟。视频9.3.1展示了该错误与修复方法

代码文件：ch_legacy/code/spec/date_calculator_characterization_spec.rb

```
require 'simplecov'
SimpleCov.start
require './time_setter'
describe TimeSetter do
  { 365 => 1980, 366 => 1981, 900 => 1982 }.each_pair do |arg,result|
    it "#{arg} days puts us in #{result}" do
      TimeSetter.convert(arg).should == result
    end
  end
end
```

图 9.7：这个简单的规则来源于视频9.3.1中展示的逆向工程技术，达到了100%的C0覆盖率并能帮助我们找到图9.6的一个错误

从这点上说，以命令式场景开始也是很好的，因为我们的目的是增加覆盖率，并提供基准事实来创建更多的细节测试。一旦有一些绿色的集成测试，你可以把注意力放在单元或功能级别的测试上，就如敏捷生命周期中TDD在由外而内跟从BDD。

综合级别的鉴定测试仅仅抓取了我们观察到的行为而没有要求我们理解这些行为是 **如何** 发生的，而一个单元级别的鉴定测试要求我们理解行为的实现。例如，考虑图9.6中的代码。我们将在下一节进行细节的讨论，这个代码有很多问题，不仅仅是包含一个错误。方法**convert**在给定一个起始年份（在这里是1980年），以及从这一年1月1日起经过的天数的条件下计算当前年份。你如何在没有理解方法的逻辑细节的情况下构建单元测试？

在一段我们还不理解的代码里，Feathers描述了一个用于“逆向工程”规则的有用的技术：创建一个包含很可能失败的断言的规则。运行这个规则，使用错误信息的信息来改变规则以匹配实际的行为。视频9.3.1展示了我们如何对**convert**做这件事，结果得到了图9.7中的规则，甚至还找到了过程中的一个错误！

视频9.3.1: 为TimeSetter创建鉴定测试规则.
光盘文件：ch_arch/mov/characterization_spec.mp4
我们创建了在错误结果处插入断言，然后基于实际的测试行为修改它们的规则。我们的目标是尽可能完整地获取当前行为，以便在代码变更破坏了当前行为的时候能马上知道，所以我们的目标是100%的C0覆盖率（尽管这并不能保证没有错误），这是很有挑战性的，因为这里的代码没有很好地分层。我们努力的结果是找到了一个错误，该错误在2008年12月31日使数以千计的微软Zune播放器瘫痪。

鉴定测试小结：

- 为了在缺乏测试的情况下覆盖并修改代码，首先创建鉴定测试来获取代码目前运行的情况。
- 综合级别的鉴定测试，例如Cucumber情景，由于只获取外部可见的应用行为，所以从它开始鉴定测试通常更容易。
- 为了给我们没有完全理解的代码创建单元或功能级别的鉴定测试，我们可以编写一个在错误结果处插入断言的规则，基于错误信息修复这个断言，且重复这个过程直到有足够高的覆盖率。

自我检查9.3.1. 标出以下每项是单元和功能测试的目标，还是鉴定测试的目标，或者两者都是：

(1) 提高覆盖率

(2) 测试边界条件和极端状况

(3) 文档化应用代码的目标和行为

(4) 防止回归（重现先前的错误）

◇ （1）和（3）是单元、功能以及鉴定测试的目标。（2）和（4）是单元和功能测试的目标，但不是鉴定测试的目标。

■ **详细阐述：应该通过的测试规则却没有通过，该怎么应对？**

如果测试组件已经过时，一些测试可能失败并标为红色。与其在理解代码之前试着修复测试，不如将它们标记为“阻塞”（例如，使用RSpec的**pending**方法）并加上一条注释来提醒你稍后回来找到它们失败的原因。提升覆盖率的同时，专注于保留已有功能的当前任务，不要再同时分散精力来尝试修复错误。

9.4 注释

遗留代码不仅仅只缺乏测试和好的文档，它的注释也经常缺失或与代码不一致。迄今为止，我们并没有提供如何写好注释的建议，因为我们假定通过本书你已

代码文件：ch_legacy/code/badcomments.rb

```
# Add one to i.
i+=1;

# Lock to protect against concurrent access.
SpinLock mutex;

# This method swaps the panels.
def swap_panels(panel_1, panel_2)
#...
end
```

图 9.8: 糟糕的注释示例，陈述了显而易见的事情。你可能会惊讶，即使在其他一些写得很好的应用中，注释也经常模仿代码（这些关于注释的例子及建议来自John Ousterhout）

代码文件：ch_legacy/code/goodcomments.rb

```
# Good Comment:
# Scan the array to see if the symbol exists

# Much better than:
# Loop through every array index, get the
# third value of the list in the content to
# determine if it has the symbol we are looking
# for. Set the result to the symbol if we
# find it.
```

图 9.9: 与描述你如何实现它相比，示例中的注释提升了抽象的级别（这些关于注释的例子及建议来自John Ousterhout）

经知道如何写好代码。现在我们对注释提供一个简短的建议，只要你写出过成功的鉴定测试，捕捉通过给遗留代码添加注释所学到的东西将是明智的。

理想情况下，你在写代码的同时也写注释。如果稍后回来，你会发现自己忘记了设计思想，所以这时写的注释仅仅是模仿代码。唉，这就是遗留代码常见的错误。

注释应当描述代码中并不显而易见的事情。这个建议是把双刃剑，因为它意味着

- **不要** 仅仅重复代码中显而易见的事情。图9.8给出了糟糕的注释示例。
- **要** 在低层次和高层次都思考什么不是显而易见的。图9.9 给出了好的注释示例。

显而易见是对稍后出现并阅读你的代码的人而言的，而不是最初的开发者。不是显而易见的例子包括变量的单位、代码不变量、需要一个特殊实现的微妙问题。编写代码的时候文档化出现在你脑海中的设计问题是尤其重要的。解释 **为什么** 代码要用这种方式写。在这种情形下，你在尝试文档化另一个程序员脑子里的东西。一旦你弄明白了，一定要在你忘记之前写下来。

一般情况下，注释应当比代码有更高的抽象程度。程序员的目的是编写类及其它代码来隐藏复杂性，这意味着，为了使代码更容易使用而不是更容易编写。抽象可能从实现来说并不明显，注释应当抓住这个抽象。例如，调用一个方法需要知道什么？在调用一个方法之前不应该需要阅读它的代码。

我们对本书中内容感到兴奋的一个原因是，书中几乎每个其它软件工程启示都配有一个工具，该工具可以让你停留在正确的道路上，并让其他人更易于检查这一点。唉，这并不是关于注释的启示的实例。比自律好实施的唯一机制是我们在10.7节中描述的检查。

注释小结：

- 注释最好在写代码的同时编写，它不是事后的想法。
- 注释不应当重复代码中显而易见的东西。例如，解释 **为什么** 代码用这种方式来编写。
- 注释应当有比代码有更高的抽象层次。

自我检查9.4.1. 对或错：遗留代码经久不衰的一个原因是它一般有好的注释。
◇ 错误。虽然我们希望这是真的。注释经常会丢失或者与代码不相符，这是我们称它为遗留代码而不是漂亮的代码的一个原因。

9.5 度量、代码味道及SOFA

> 质量下降——除非软件有严格的维护和适应运行的环境，否则[软件]系统的质量将表现出下降的趋势。
>
> ——Lehman的软件演化第七法则

本书的一个重要主题是软件工程不仅是创建工作代码，而且是创建“漂亮的”工作代码。本章会阐明为什么我们相信：漂亮的代码更容易维护，而且花费更少。漂亮的代码能够比硬件保持更长时间的生命力，即使工程师的美感没有被漂亮的代码所感染，他们也会欣赏减少生命周期维护成本带来的实际经济优势。

你如何分辨代码是不够漂亮的，以及如何来改善它？我们都看到过不够漂亮的代码例子，即使我们一直不能找出问题所在。我们可以通过两种方法确定问题：数量上使用 **软件度量** 以及质量上使用 **代码味道** 。这两种方法都是有用的，可以告诉我们代码之间的不同，我们将在图9.6中的丑陋的代码上应用这两种方法。

计划文档模式 的软件项目有时包括基于软件度量的特定合约要求。

软件度量 是代码复杂度的数量化测量，一般情况下可以作为彻底地测试一段代码的困难程度。有许多度量的标准存在，关于它们有用性、有效性以及价值的“合理范围”有很多不同的意见。许多度量都是基于程序的 **控制流程图** ，其中每一个节点都代表一个 **基本块** （一组永远在一起执行的声明），节点之间从A到B的边代表在一些代码路径中B基本块在A基本块之后马上执行。

图9.10展示了图9.6中代码的控制流程图，可以用来计算两种广泛应用的指标或方法层级的复杂性：

软件工程师Frank McCabe Sr.在1976年发明了圈复杂度度量。

(1) **圈复杂度** 测量一段代码中非线性路径的数目。

(2) **ABC得分** 是一段代码的**赋值**（Assignments）**分支**（Branches）以及**条件**（Conditionals）数目的加权和。

这些分析一般在源代码上执行，它们最初是给静态类型语言开发的。对于动态语言，由于元编程等其它可能引起运行时控制流程图改变的机制，分析更为复杂。尽管如此，它们仍是有用的一阶度量，并且正如你可能期待的，Ruby社区已经开发了用于测量它们的工具。`saikuro`能计算出一个简化版本的圈复杂度，`flog`能计算出多种适合Ruby习语的用合适的方式加权的ABC分数。这两个以及更多的工具都包括在`metric_fu`工具包（本书配套软件的一部分）中。在Rails应用上运行`rake metrics`能得到包括这些在内的很多度量，并能突出显示多个度量指标都超

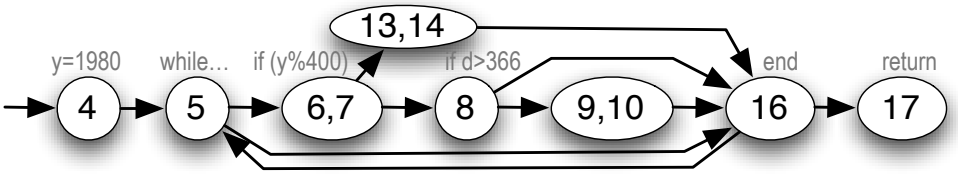


图 9.10: 图9.6中代码的控制流程图中节点数目与其对应的代码行数。圈复杂度是$E - N + 2P$，其中E是边的数目，N是节点的数目上，P是连通分支的数目。convert方法能得出圈复杂度为4，和saikuro得出的一致，flog方法得出ABC得分（赋值、分支、条件）为23。图9.11将这些分数放在了一个表中

度量标准	工具	目标分数	章节索引
代码测试比	`rake stats`	$\leq 1:2$	8.8节
C0覆盖	`SimpleCov`	$\geq 90\%$	8.8节
ABC分数	`flog (rake metrics)`	< 20/方法	9.5节
圈复杂度	`saikuro (rake metrics)`	< 10/方法	9.5节

图 9.11: 一些有用的度量，包括计算它们的Ruby工具以及建议的"正常"范围的总结，强调了美观性和可测试性之间的联系（圈复杂性的推荐值来自NIST，即美国国家标准技术研究协会（the U.S. National Institute of Standards and Technologies）。工具包`metric_fu`包含有`flog`、`saikuro`以及在第11章将会看到的用于计算度量的额外工具

出它们推荐范围的代码部分。图9.11总结了到现在我们看到的与可测试性有关，因而与代码美观性有关的有用的度量。

第二种找出代码错误的方法是通过寻找 **代码味道**，即源代码还没有被度量标准获取的结构特点。和真实的气味一样，代码味道能将我们的注意力吸引到 **可能** 有问题的地方。Martin Fowler关于重构的经典著作Fowler et al. 1999列出了22种代码味道，其中4种如图9.12所示。Robert C. Martin的 **Clean Code**（Martin 2008）有一个令人惊讶的包括63种之多的代码味道的更完整的分类目录，其中有3种是专门关于Java的，6种是关于测试的，其它的是比较普遍适用的。

设计味道（见第11章）告诉我们出现在类之间交互上的，而不是出现在一个特定类的方法的错误。

4个出现在Martin的Clean Code一书中中的特殊的味道值得我们重视，因为它们是你可以通过简单的重构修复的错误的征兆。这4种味道由首字母缩略词**SOFA**表示的，代表了一个编写良好的方法应当有的：

- 保持**简短**（Short），以便能迅速获取主要目的。
- 只做**一件**（One）事情，以便测试能集中于彻底检查这一件事情。
- 输入**少量**（Few）的参数，以便非常重要的参数值组合都能被测试到。
- 维持一个固定级别的**抽象**（Abstraction），以便它不会在 **如何做** 和 **怎么做** 之间来回跳转。

图9.6至少违反了第一条和最后一条，同时也表现出了其它味道，我们可以通过对它运行`reek`看到：

代码文件：ch_legacy/code/reek_output.txt

```
time_setter.rb --- 5 warnings:
  TimeSetter#self.convert calls (y + 1) twice (Duplication)
  TimeSetter#self.convert has approx 6 statements (LongMethod)
  TimeSetter#self.convert has the parameter name 'd' (UncommunicativeName)
  TimeSetter#self.convert has the variable name 'd' (UncommunicativeName)
  TimeSetter#self.convert has the variable name 'y' (UncommunicativeName)
```

名称	症状	可能的重构方法
霰弹式修改	对一个类或一个方法进行细小的改动，结果波及到其它类或方法，使它们也产生了许多细小的改动。	通过移动方法或移动字段来把所有的数据或行为集中到一个地方。
数据泥团	同样的三到四个数据项似乎经常作为参数一起传递，或在一起处理。	使用提取类或保存整个对象来创建一个类，它能将数据聚集起来，并传递该类的实例。
过度亲密	一个类了解另一个类实现（方法或属性）的信息了解太多。	如果这些方法确实在别处需要，那么就移动方法或移动字段；如果两个类之间有重叠，那么就使用提取方法，或使用委托来隐藏实现细节。
重复样板	在许多不同地方有一些完全相同或几乎相同的代码（不够DRY）。	使用提取方法将多余的代码装入，以使在重复的地方都能调用。在Ruby中，你甚至可以用**yield**来提取“封闭的”代码从而将它变成到不重复的代码。

图 9.12: Fowler的22种代码味道的列表中有4个命名古怪的代码味道，和重构有关，如果采取措施，可以修复这些味道（其中一部分将在下一节看到）。参考在Fowler书里表格中提到而没有在本书中介绍的重构

不够精简（第2行）。诚然这仅仅是一个很小的重复，但和任何味道一样，询问我们自己代码为什么是这个样子是有价值的。

无意义的命名（第4～6行）。出现的整型变量**y**（第6、7、10、14行）以及另一个变量**d**——它们可能是什么？对这个问题而言，类**TimeSetter**将时间赋值给谁，以及在**convert**中什么被转换成了什么？40年以前，由于内存非常珍贵，所以变量名字尽可能地短以给代码留出更多空间。今天，没有任何借口给变量无意义的名称；图9.13给出了一些建议。

类别	指导方针	例子
变量或类名	名词短语	**PopularMovie, top_movies**
有副作用的方法	动词短语	**pay_for_order, charge_credit_card!**
返回数值的方法	名词短语	**movie.producers, actor_list**
布尔表达式或方法	形容词短语	**already_rated?, @is_oscar_winner**

图 9.13: 变量命名指南基于简单的英语，摘自Green and Ledgard 2011。由于磁盘空间是免费的，以及现代编辑器有自动补全功能，节省了你重新输入全名的时间，你的同事将会感谢你使用了@is_oscar_winner而不是OsWin

一个方法过长（第3行）。一个方法中更多行的代码意味着更多隐藏错误的地方，更多需要测试的路径，以及测试中需要更多的模仿和模块。然而，过长的方法真的成为了很多特定问题中浮现出来的一个征兆——在本例中，导致坚持一个级别的**抽象**失败。如图9.14所示，**convert**真正包含着少量的高层次的步骤，每一个都可以被分成一个子方法。但在代码中，没有办法区分步骤或子步骤之间的界限，使方法更难于理解。事实上，第6～8行嵌套的条件使程序员难以用人脑来“排查”这些代码，而且因为你不得不选择一组测试实例来覆盖所有可能的代码路径而使测试更复杂化。

古老的智慧，对24行每行80个字符的文本终端，一个方法的代码不应该超过一个屏幕。一个现代的22英寸显示器能显示它的10倍多，所以如今像SOFA这样的指南更值得依赖。

作为这些缺点的一个结果，你可能需要努力工作以解决这个有关的简单方法做了什么。（你可能会抱怨这些代码缺少注释，但是一旦上述的味道被修复，就几乎不再需要注释了。）敏锐的读者可能会发现1980、365和366这些常量，并且推断出这个方法处理了与闰年有关的事情，以及1980是一个特殊的年份。事实上，**convert**通过给出起始年份即1980年，以及从这一年1月1日开始经过的天数，根

代码文件：ch_legacy/code/date_calculator_pseudocode.txt

```
start with Year = 1980
while (days remaining > 365)
  if Year is a leap year
    then if possible, peel off 366 days and advance Year by 1
  else
    peel off 365 days and advance Year by 1
return Year
```

图 9.14: 给出从一个开始年份（1980）的第一天起经过的天数，用伪代码写出对当前年份的计算会清楚很多。注意到方法做了什么 更易于快速地掌握，尽管每一步在写成代码的时候都要分解成更细小的细节。我们将对Ruby代码进行重构以使它和这份伪代码一样简明清楚

据图9.14中的简单步骤来计算当前年份。

在下一节中，我们将 **重构** 这些Ruby代码以使它和此伪代码一样透明——应用在不改变功能的情况下提升结构的转换。

小结：

- 软件度量提供了关于代码质量的量化测定。但哪种度量是最有用的，以及它们"正常的"值应该是多少（尤其对于Ruby这样的动态语言），众说纷纭。像圈复杂度和ABC得分这样的度量能用来指导你搜索尤其需要注意的代码，就如较低的C0覆盖率意味着没有测试良好的代码。
- 代码味道对导致代码难以阅读的问题提供了定性但详细的描述。根据你使用的分类不同，有超过60种特定的味道已经被确定。
- 首字母缩写词SOFA指出了一个方法的4个令人满意的特点：它应当是**简短**（**Short**）的，做**一件**（**One**）事情，有**少量**（**Few**）的参数，维持单一的**抽象**（**Abstraction**）层次。

自我检查9.5.1. 给出Ruby中一个能曲解度量的动态语言特征的例子，如圈复杂度或ABC得分度量。

◇ 任何元程序设计的机制都可以完成这个。一个简单的例子是**s="if (d>=366) [...]"; eval s**，对于这个字符串的求值能够引起条件语句被执行，即使代码本身仅仅包含一个对变量的赋值以及一个对**eval**的调用而没有条件语句。一个微妙的例子是像**before_filter**（Section5.1）这样的方法，它实质上在一个控制器动作被调用前向方法列表中添加了一个新的方法。

自我检查9.5.2. 从单元可测试性的角度来看，哪一条SOFA的指导方针——要**简短**，做**一件**事情，有**少量**参数，维持单一层次的**抽象**——你认为是最重要的?

◇ **少量**的参数意味着在方法中代码路径依赖于参数的可能性更小，使测试更易于处理。**简短**的方法当然是易于测试的，但这个特点通常是在其它三个都存在的情况下。

9.6 方法级别的重构：使用嫁接替代依赖

> 日趋增长的复杂性——随着[一个软件]系统的进化，除非有方法来维持或降低，否则它的复杂性就会增长。
>
> ——Lehman的软件演化第二法则

有了9.3节中的鉴定规范，我们就有了一个坚实的其础来支撑我们的重构方法，用来修复在9.5节中发现的问题。术语 **重构** 不仅仅涉及一个一般的过程，同时也涉及特定的代码转化方法的实例。因此，和代码味道一样，我们提到重构方法的分类，有很多这样的分类供我们选择。我们更喜欢Fowler的分类目录，所以本章的示例遵循Fowler的术语并和他的书 **Refactoring: Ruby Edition** （Fields et al. 2009）中的第6、8、9、10章有交叉引用。由于代码味道和重构方法之间的对应关系并不完美，一般情况下，这些章节中的每一章都描述了一组与特定的代码味道或问题对应的方法级别的重构，接下来的章节描述了影响了多个类的重构，我们将在第11章中学习。

每一个重构方法都由一个描述性的名称和通过每一步都是微小改进的一步一步转化代码的过程组成，每一步后都会进行测试。大多数重构至少会导致暂时的测试失败，因为单元测试经常依赖于实现，这正是重构改变的东西。重构过程的一个关键目标是最小化测试失败（红色）的时间总和。这个想法要求每一个重构的步骤都要足够小，使得在进行下一个步骤之前调整测试使其通过不太困难。如果发现从红色回到绿色比预想要难很多，那么必须确定你对代码的理解是否完整，否则你在重构的时候肯定破坏了什么东西。

开始进行重构可能会挺打击人的：不知道该在哪里重构，决定如何改进一段代码可能是困难的。理解重构的说明或什么时候使用它们的动机可能还比较困难。直到你有了改进一段代码的经验，不要被这个先有鸡还是先有蛋的问题打击到信心；就像TDD和BDD一样，一些开始看起来挺打击人的东西很快就会变得熟悉起来。

作为一个开头，图9.15展示了我们将在代码中应用的四种Fowler的重构方法。在本书中，每一个重构方法都伴随着一个例子和一个用来执行重构的极其详细的操作步骤列表，一些情况下还会参考对应用这个方法可能必要的其它重构方法。例如图9.16展示了应用提取方法重构的前几个步骤。记住这些例子，我们就能重构图9.6中的代码。

名称（章节）	问题	解决方案
提取方法（6）	你有一个可以集中起来的代码碎片。	将碎片放入一个方法中，方法名解释了方法的目的。
分解条件语句（9）	你有一个复杂的条件（if-then-else）语句。	从条件中提取方法，“then”部分，以及“else”部分。
用方法对象替代方法（6）	你有一个很长的方法，它使用了局部变量，以致于你不能提取方法。	将这个方法变成它自己的对象，以便所有的局部变量都成为该对象的实例变量。这样你就能将该方法分解成这个对象的其它方法了。
使用符号常量替代魔法数（8）	你有一个有特殊含义字面数值。	创建一个用它的意义命名的常量，并用它来替代该数值。

图 9.15: 四个重构的例子，括号中是它们在Fowler的书中出现的章节数。每一个重构方法都有一个名字、一个它所解决的问题以及解决这个问题的代码转换的综述。Fowler的书还介绍了每个重构方法的详细的机制，如图9.16所示

函数过长 是图9.6中最明显的代码味道，但它仅是许多特定问题导致的整体征兆。由**convert**得出的较高的ABC分数（23）暗示了我们需要开始集中注意力的地方：第6和7行的条件**if**难以理解，而且条件嵌套了两层。如图9.15所建议的，一个很难阅读的条件表达式可以通过应用最常见的 **分解条件式** 重构方法来改进，而这种方法又依赖于 **提取方法** 。我们将一些代码移到一个有描述性名称的新的方法里，如图9.17所示。需要注意的是，除了使条件句更具可读性之外，**leap_year?**的单独定义

(1) 创建一个新的方法，用它的目的来命名它（即它是做什么的，而不是怎么做的）。如果你想要提取的代码非常简单，比如说是一个简单的消息或是函数调用，那么如果新方法的名称能够更好地阐述代码的目的，你就应该提取它。如果你不能想出一个更有意义的名称，就不要提取这段代码。

(2) 把提取的代码从源方法复制到新的目标方法中。

(3) 浏览提取的代码，注意任何对源方法中局部变量的引用。这些是新方法的局部变量和参数。

(4) 观察是否有临时变量仅在提取的代码中使用过。如果有，在目标方法中声明它们为临时变量。

(5) 察看提取的代码是否更新过局部变量。如果一个变量被更新了，看看你能不能将提取出的代码当作一个基准，把结果赋给相关的变量。如果这种方法很棘手，或者有超过一个这样的变量，目前你就不能提取这个方法。你可能需要使用 **分裂临时变量** 的方法并再次尝试提取。你可以通过 **使用疑问替代临时变量** （参见样例讨论）的方法来消除局部变量。

(6) 将提取的代码读出的局部变量通过参数传递给目标方法。

(7) .....

图 9.16: Fowler的提取方法重构的详细步骤。在他的书中，每一个重构方法都被描述为一步一步的转化过程，同时过程也可能参考其它转换方法

也使闰年的计算可以独立进行测试，并在第6行提供了嫁接，我们可以在这里存根该方法来简化**convert**的测试，与8.7节结尾的详细阐述中的例子类似。一般情况下，当一个方法混合了 **做什么** 的代码和 **怎么做** 的代码，这可能是一个警告，警告你来检查是否需要使用提取方法以便维持一致的抽象级别。

条件句也是两层嵌套的，使代码难于理解并增加了**convert**的ABC得分。 **分解条件式** 重构也能通过将条件句的每一分支替换为一个提取出的方法来打破复杂的条件。需要注意的是，尽管条件句有两个分支与图9.14中伪代码的第4∼6行对应，但是它们都有改变**d**和**y**的值的 **副作用** （因此我们在提取的方法中使用**!.**）为了使这些副作用对**convert**可见，我们必须把本地变量变成贯穿**TimeSetter**的类变量，并在这一步给它们更具描述性的命名**@@year**和**@@days_remaining**。最后，由于**@@year**现在是一个类变量，我们不再需要显式地把它传递给方法**leap_year?**。图9.18展示了我们的结果。

只要我们清理干净，图9.18中的代码也修改了两个小的代码味道。第一个是无意义的命名：**convert**没有很好地描述这个方法做什么，它的参数名称**d**也是没有意义的。另一个是如第4行的1980等“魔法数”即字面常数的使用。我们将 **用符号常量代替魔法数** （Fowler的书第8章）的方法应用于它，用更有意义的常量名称**STARTING_YEAR**代替它。那么其它常数，例如365和366呢？在本例中，留着它们本来的样子可能对大多数程序员来说足够熟悉，但是如果看到的是351而不是365，以及如果第26行（在**leap_year?**中）使用了常数19而不是4 ，你可能不会察觉到对 **希伯来日历** 的闰年的计算。记住重构仅仅对读者来说改进了代码，计算机并不在意这些。所以在这样的情况下，使用你的判断来判定进行多少重构才是足够的。

在这里的情形中，对图9.18中重构过的代码重新运行`flog`，能使新命名的**calculate_current_year**的ABC得分从23.0降到6.6，远低于NIST建议的阈值10.0。同时，现在`reek`只报告了两个味道。第一个是对辅助方法**peel_off_leap_year**以及**peel_off_regular_year**的“低内聚”。这是一个设计味道（design smell），

代码文件：ch_legacy/code/extract_method.rb

```
1  # NOTE: line 7 fixes bug in original version
2  class TimeSetter
3    def self.convert(d)
4      y = 1980
5      while (d > 365) do
6        if leap_year?(y)
7          if (d >= 366)
8            d -= 366
9            y += 1
10         end
11       else
12         d -= 365
13         y += 1
14       end
15     end
16     return y
17   end
18   private
19   def self.leap_year?(year)
20     year % 400 == 0 ||
21       (year % 4 == 0 && year % 100 != 0)
22   end
23 end
```

图 9.17： 对图9.6的第3和4行应用提取方法重构，用一个我们声明能使类的实现细节很好封装起来的命名贴切的方法（第19～22行）替换条件句，能使条件句的目的立刻变得清晰（第6行）。更透明的，我们可以再次对leap_year?使用提取方法，提取出方法every_400_years?和every_4_years_except_centuries?。注意第7行反映了视频9.3.1中描述的错误修正

代码文件：ch_legacy/code/decompose_conditional.rb

```
1  # NOTE: line 7 fixes bug in original version
2  class TimeSetter
3    ORIGIN_YEAR = 1980
4    def self.calculate_current_year(days_since_origin)
5      @@year = ORIGIN_YEAR
6      @@days_remaining = days_since_origin
7      while (@@days_remaining > 365) do
8        if leap_year?
9          peel_off_leap_year!
10       else
11         peel_off_regular_year!
12       end
13     end
14     return @@year
15   end
16   private
17   def peel_off_leap_year!
18     if (@@days_remaining >= 366)
19       @@days_remaining -= 366 ; @@year += 1
20     end
21   end
22   def peel_off_regular_year!
23     @@days_remaining -= 365 ; @@year += 1
24   end
25   def self.leap_year?
26     @@year % 400 == 0 ||
27       (@@year % 4 == 0 && @@year % 100 != 0)
28   end
29 end
```

图 9.18： 我们通过把每一分支替换为一个提取的方法分解了第7行的条件句。注意到总的代码行数增加的同时，convert自身变得更短了，它的步骤现在也和图9.14中的伪代码对应得更紧密了，将细节委派给提取的辅助方法的同时维持了同一级别的抽象

代码文件：ch_legacy/code/replace_method_with_method_object.rb

```
# An example call would now be:
#  year = TimeSetter.new(367).calculate_current_year
# rather than:
#  year = TimeSetter.calculate_current_year(367)
class TimeSetter
  ORIGIN_YEAR = 1980
  def initialize(days_since_origin)
    @year = ORIGIN_YEAR
    @days_remaining = days_since_origin
  end
  def calculate_current_year
    while (@days_remaining > 365) do
      if leap_year?
        peel_off_leap_year!
      else
        peel_off_regular_year!
      end
    end
    return @year
  end
  private
  def peel_off_leap_year!
    if (@days_remaining >= 366)
      @days_remaining -= 366 ; @year += 1
    end
  end
  def peel_off_regular_year!
    @days_remaining -= 365 ; @year += 1
  end
  def self.leap_year?
    @year % 400 == 0 ||
      (@year % 4 == 0 && @year % 100 != 0)
  end
end
```

图 9.19：如果我们使用Fowler的书中建议的重构方法，代码将会更整洁，因为我们将使用实例变量而不是类变量来追踪这些副作用。但由于calculate_current_year现在成了一个实例方法，所以这改变了它被调用的方式。这将改变现有的代码和测试，所以这种办法可能会被推迟到重构过程的后面一些

我们将在第11章中讨论它的意义。第二个味道是在方法中声明类变量。当我们应用分解条件式和提取方法时，我们将局部变量转换成了类变量**@@year**和**@@days_remaining**以便于新提取的方法能够成功修改这些变量的值。我们的解决方法是有效的，但比 **用方法对象替代方法** （Fowler的书第6章）更笨拙一些。在那个重构方法中，原始的方法**convert**被转换成了一个对象 **实例** （而不是一个类），其实例变量捕捉了对象的状态，然后辅助方法在这个实例变量上操作。

图9.19展示了应用这种重构方法的结果，但有一个重要的警告。到目前为止，没有任何重构方法导致鉴定测试失败，因为规则是仅仅调用**TimeSetter.convert**。但应用 **"使用方法对象代替方法"** 的策略在某种程度上改变了**convert**的调用接口，因而使测试失败了。如果在处理实际的遗留代码，我们就需要找到每一个调用**convert**的地方，使它改为使用新的调用接口，并改变所有因此失败的测试。在实际的项目中，我们想要避免不必要的改变破坏调用接口，所以需要仔细地考虑通过应用重构方法获得可读性，是否值得其带来破坏性改变的风险。

重构方法小结：

- 一个重构是对一段代码进行的特殊转化，包括一个名称，何时使用这个重构方法和它能做什么的描述，以及一个详细的操作步骤序列。有效的重构方法应当提升软件度量，消除代码味道，或兼而有之。
- 尽管大多数重构方法不可避免地引起一些已有的测试失败（如果没有的话，问题中的代码很可能未测试好），重构过程的一个关键目的是最小化这些需要修改的测试的数量，并使它们再次变成绿色所需的时间的总和。
- 有时应用一个重构方法可能导致需要首先递归地应用简单的重构方法，就像**分解条件式** 可能需要应用 **提取方法** 一样。

■ **详细阐述：重构方法与语言选择**

一些重构方法弥补了编程语言可能鼓励写出不好的代码的特性。例如，一种对在**封装字段** 中添加嫁接的建议的重构，其中直接访问对象的实例变量被获得方法（getter）和设置方法（setter）替代了。这在Java中是有意义的，但正如我们看到的，获得方法和设置方法是提供在Ruby对象的外部获取其实例变量的 **唯一** 方法。（这个重构方法在对象自己的方法内仍有意义，就像3.4节结尾处的详细阐述建议的那样。）类似的，**泛化类型** 这一重构方法建议创建更多一般的类型以提高代码可共享性，但Ruby的mixin功能和鸭子类型（duck typing）使这样的共享变得容易。正如我们将在第11章中看到的那样，同样的情形，一些设计模式在Ruby中其实是不需要的，因为它们解决的问题在动态语言中是不会出现的。

自我检查9.6.1. 以下哪一个 **不是** 方法级别的重构目标：（a）降低代码复杂度，（b）消除代码味道，（c）消除错误，（d）提高可测试性？

◇（c）。虽然调试是重要的，但重构目的是在改变代码结构的同时 **保护** 代码当前的功能。

9.7 计划文档的视角

第1章的术语 **生命周期** 的一个意义就是，一个软件产品在开发完成后进入了维护阶段。大约三分之二的成本是在维护上，而只有三分之一在开发上。公司每年收取年度维护费用，大约相当于10%的软件费用，用于支付给团队来做软件维护工作。

遵循计划文档过程的典型组织一般有不同的团队来做开发和维护工作，开发者在项目一经发布的时候就会被重新分配到新的项目上。因此，我们现在有一个 **维护经理** 来接管开发阶段项目经理的职位，并且还有 **软件维护工程师** 在修改代码的团队里工作。不幸的是，软件维护工作有个单调乏味的名声，所以通常情况下由一个组织里最新的人或至少是不熟练的项目经理和工程师来执行。许多组织使用不同的人来进行测试的质量评定和编写用户文档等工作。

对于使用计划文档过程开发的软件产品，维护的环境和开发的环境是非常不相同的：

- **工作软件** ——一个工作软件产品在整个阶段中都能固定功能范围内的，并且新的发布必须与已有的功能不冲突。

- **用户协作** ——与其说是努力满足一个作为协商合同的部分的规格，不如说这一阶段的目标是与用户一起工作，来为下一个版本改善软件。

- **响应变化** ——基于对产品的使用，用户发送一连串的 **变更请求** ，可以是新功能也可能是错误修复。维护阶段的一个挑战是以什么样的优先顺序来实现一个更改请求，以及它应当出现在哪一个版本中。

变更请求在IEEE标准中又称作 **维护请求** 。

回归测试在维护中扮演着一个更重大的角色，可以在开发新功能时避免损坏旧的功能。重构也扮演着一个更重大的角色，因为你需要重构来实现一个更改请求，或简单地使代码变得更具可维护性。如果一个公司在开发一个不是它维护的软件，那么在计划文档的最初过程中很少有动机来利用额外的花费和时间使产品更易于维护，这也是重构在开发过程中扮演一个更小的角色的原因。

正如上面提到的， **变更管理** 是基于客户及其他涉众做出的变更请求，用来修复错误或是改善功能（参见10.7节）。他们一般填写 **变更请求表** ，这是用一个票据跟踪系统跟踪的，所以每一个请求都会得到回应并被解决。变更管理的一个关键工具是版本控制系统，它跟踪了所有的对象的所有修改，正如我们在10.4节和10.5节提到的。

先前的段落应该听起来挺熟悉，因为我们在介绍敏捷开发。实际上，这三个描述是从敏捷宣言（参见1.9节）中抄出来的。因此， **维护本质是一个敏捷过程** 。变更请求就像用户故事，变更请求分类与任务点分配以及使用Pivotal Tracker来决定如何优先排序故事很相像，软件产品新版本发布与工作原型中的敏捷周期一样。计划文档模式的维护甚至将一个大的变更请求分割成许多小的请求，使它们更容易被评估和实现同样的策略，和我们对用户故事中超过8个任务点（参见7.3节）时所做的一样。因此，如果同样的团队在开发和维护软件，当使用敏捷生命周期时，在第一次产品发布之后没有任何改变。

虽然有一篇论文声称已经成功地用一个敏捷过程来维护使用计划文档过程开发的软件（Poole and Huisman 2001），正常情况下使用计划文档模式来开发的组织也会使用计划文档模式来维护。和我们在前面章节看到的一样，这个过程需要一个强有力的项目经理来进行成本评估，制定日程表，降低项目风险，并且为项目的每一部分规划出详细的计划。这个计划可以在多个文档中反映出来，就像我们在图7.12和图8.24，以及图10.9、图10.10和图10.11中看到的那样。因此，计划文档过程中变更的影响不仅是变更代码的成本，还有变更文档和测试计划的成本。计划文档模式中有更多的对象，在作出变更的时候就需要更多的工作来同步地维持它们的一致性。

变更控制板 检查所有重大的请求来决定某个变更是否应该被包含到系统的下一个版本中。这个组需要评估变更的成本来决定是否批准变更请求。维护经理必须评估实现每一个变更的时间和精力，和项目经理在项目开头（参见7.10节）做的非常像。这个组还要询问QA小组关于测试成本，包括运行所有回归测试和为一个变更开发新的测试（如果需要的话）的成本。文档组还要评估变更文档的成本。最后，用户支持组检查是否有一个工作区来决定变更是否是紧急的。除了成本，该组在决定做什么的时候需要考虑变更之后的产品的价值增长。

为了帮助跟踪计划文档过程中我们必须要做的事情，你不会惊讶于学习IEEE提供的标准来寻求帮助。图9.20展示了IEEE维护标准1219－1998中的一个维护计划提纲。

理想情况下，变更可以被安排好和一个即将来临的发布版本同步地维持代码、文档以及计划。唉，一些改变是如此紧急，以至于一切其它的事情都必须扔掉来使新的版本尽可能快地提供给客户。例如：

- 软件产品崩溃了。

目录
1. 简介 2. 参考 3. 定义 4. 软件维护综述 4.1 组织 4.2 调试优先级 4.3 资源摘要 4.4 责任 4.5 工具、技术、和方法 5. 软件维护过程 5.1 问题/修正的鉴别/分类，以及优先级 5.2 分析 5.3 设计 5.4 实现 5.5 系统测试 5.6 验收测试 5.7 将会 6. 软件维护报告需求 7. 软件维护管理需求 7.1 异常解决与报告 7.2 偏差政策 7.3 控制程序 7.4 标准、实践、和惯例 7.5 性能追溯 7.6 计划质量控制 8. 软件维护文档需求

图 9.20: 在IEEE 1219－1998中用于系统和软件工程的维护标准的维护计划提纲

- 一个安全漏洞被发现了，使得产品收集的数据格外容易受到攻击。
- 依赖的操作系统或库有了新版本，促使产品通过变更才可以继续提供功能。
- 一个竞争者发布了某个产品或功能，如果不跟上的话将显著影响与客户的贸易。
- 新通过的法律影响了产品。

团队在紧急情况过后将更新文档和计划仅是一种猜想，而实际上紧急情况可能会太频繁以至于维护团队无法将所有事情保持同步。这样的拖延使得代码变得更难维护，反过来导致更多重构代码的需求。代码越“难看”，就越容易出错，越花费时间去重构！作为敏捷模式自然的一部分，它很少由变更控制委员会来批准需要重构的变更，因为这些变更要昂贵很多。

回填（Backfilling）是维护工程师用来描述在紧急情况发生后将代码回滚到同步状态的术语。

除了评估变更控制板上每一个潜在变更的成本之外，一个组织的管理层可能会询问一个项目每年维护费用是多少。维护经理可能基于软件度量进行这个评估，和项目经理使用它们来评估开发一个项目的成本非常之像（参见7.10节）。度量对维护来说是与开发不同的，因为它们在测量维护过程。可能表明增加维护的困难度的度量的例子包括：分析或实现一个变更请求的平均时间以及产生或批准的变更请求数。

在软件产品生命周期的某一点，是否到了要替代某个功能的时候这一问题出现了。一个替代的与重构相关的方法是 **重新规划（reengineering）** 。与重构一样，这个想法是保持相同的功能性但是使代码更易于维护。示例如下：

- 改变数据库模式。
- 使用逆向工程工具来改善文档。
- 使用一个结构分析工具来识别和简化复杂的控制结构。
- 使用一个语言翻译工具来将像C和Cobol这样的面向过程的语言代码转化成像C++或Java这样的面向对象语言的代码。

让人感到有希望的是，重新规划比从草稿重新实现软件产品更便宜且更容易成

小结：
本节的主旨是你可以把敏捷模式开发当作一个维护过程，在其中改变是基础，你会与用户持续地接触，产品的新迭代会例行地部署给用户，就和新版本发布一样。因此，回归测试和重构是敏捷过程中的标准，就像它们在计划文档模式的维护阶段一样。在计划文档过程中：

- **维护经理** 扮演着项目经理的角色，他们与客户和上层管理者接触，做出成本和日程评估，文档化维护计划，并且管理 **维护软件工程师** 。
- 客户和其他涉众发出 **变更请求** ， **变更控制委员会** 根据变更的收益以及来自维护经理、文档小组、QA小组的成本评估结果将变更分类。
- **回归测试** 在维护中扮演重大的角色，用来确保新的功能不与原有的功能冲突。
- **重构** 也扮演着重大的角色，部分由于计划文档过程中的产品开发，与敏捷开发过程相比有较少的重构。
- 在代码变得越来越难以维护时，一个替代的重新开始的方法是 **重新规划** 代码以获得一个更具可维护性的系统来降低成本。

因此一个关于敏捷开发的争论如下：如果三分之二的产品成本是在维护阶段，那为什么不在整个生命周期中使用同样维修兼容的软件开发过程？

功。

自我检查9.7.1. 对或错：维护成本经常超过开发成本。
◇ 正确。

自我检查9.7.2. 对或错：重构和重新规划是同义词。
◇ 错误。它们是相关的术语，重新规划经常依赖于自动化的工具，并在软件老化并且维护越来越困难时发生，而重构是代码改善的一个持续的过程，在开发和维护阶段都会发生。

自我检查9.7.3. 将左边的计划文档模式维护术语和右边的敏捷模式术语用线连起来。

变更请求	周期
变更请求成本评估	Icebox、Pivotal Tracker中的Active列
变更请求分类	任务点
发布	用户故事

◇ 变更请求 $\Longleftrightarrow$ 用户故事；变更请求成本评估 $\Longleftrightarrow$ 任务点；发布 $\Longleftrightarrow$ 周期；还有变更请求分类 $\Longleftrightarrow$ Icebox，Pivotal Tracker中的Active列

9.8 谬论和陷阱

陷阱：将重构与改进代码合并。

当你在重构或是创建额外的测试（例如鉴定测试）以准备改善遗留代码时，一路上一直有种很强的修改“小的事情”的诱惑：看起来有一点凌乱的方法，有点老式的实例变量，看起来从哪里都永远不会执行到的死代码，看起来你可以在做其它任务

时很快地添加“很简单的”功能。 **抵抗这些诱惑！** 首先，提早建立“基准情况”测试的原因是为了把你自己引导到一个可以做出改变并确信你没有破坏任何事情的地方。在没有好的测试覆盖的情况下试着做这样的“改进”会带来灾难。第二，正如我们之前所说的，在这里再次重复，程序员是乐观主义者：看着修复起来微不足道的任务可能将你从要重构的主要任务引开而误入歧途很长时间，甚至更糟，将代码库弄到一个不稳定的，为了继续重构你必须回滚的状态。解决方法很简单：当你重构或在为之打基础时，在试着提高代码之 **前** ，专注于完成这些步骤。

谬论：从一张白纸开始会比修复这个设计更快。

除去管理层可能会明智地禁止你这样做的实际考虑，有很多原因可以表明这个观点几乎永远是错误的。首先，如果你没有花时间来理解一个系统，你就没有能力来评估重新设计会有多难，因为程序员总是无可救药的乐观，他们很有可能严重低估工作量。第二，不管遗留代码如何丑陋，当前系统是 **可用** 的。进行短的敏捷迭代的一个主要目的是“永远保持可以工作的代码”，这样做让你从头开始你立刻就违反了这一原则。第三，如果你在重新设计中使用敏捷方法，你将不得不开发用户故事和场景来驱动你的工作，这意味着你需要按优先级给它们排序，并写下其中相当多的一部分以确保你抓住了至少和当前系统一样的功能。使用本章中的技术来为系统中需要改进的部分编写场景，并用它来从这些地方驱动新代码的开发而不是完全地重写，很可能会更快。

这是否意味着你应当 **永远不** 重新开始？不是。正如Pivotal Labs的Rob Mee指出，当前代码库不能体现原始设计意图并会成为一个负担的情况是可能出现的，这时重新开始可能恰好是能做的最好选择。（有时这种情况是由于没有及时地重构导致的！）但除了最繁琐的系统以外，这应该是当其它所有的方法都被仔细地考虑过，并确定很难满足用户需求而做出的“使用核弹的选择”。

陷阱：刻意坚持度量或“极其讨厌”回避代码味道。

在第8章中我们警告过，正确性不能通过依靠一种测试（单元、功能、综合/验收）或仅依靠作为测试彻底性程度的量化的代码覆盖率来保证。类似地，代码质量不能通过任何单独的代码度量或避免特定的代码味道来保证。因此`metric_fu`工具包通过多种度量和味道检查你的代码，所以你可以确定有多个问题的同一段代码片段，即需要重构的“热点”。

9.9 结束语：持续的重构

> 停泊在港湾中的船只是安全的，但这不是我们建造船只的目的。
>
> ——Grace Murray Hopper

正如我们在本章开头所说的，修改遗留代码并不是一件能轻松完成的任务，要求的技术必须从经验中磨练。第一次总是最难的。但像重构这样基础的技能能帮助你处理遗留代码和新代码。正如我们看到的，在遗留代码、重构、可测试性以及测试覆盖之间有很深的内在关系。我们拿到既不好看又不容易测试的代码——它的复杂性度量和代码味道得分很低，并且单元测试的孤立行为很难获得——然后将它重构成有更好的度量得分，易于阅读和理解，并且易于测试的代码。简言之，我们展示了 **好的方法就是可测试的方法，可测试的方法就是好的方法。** 我们使用重构技术来美化已有的代码，同样的技术在执行提高它们自身的操作时也可以使用。例如，如果我们需要给已有的方法添加功能，而不是简单地添加一段代码并冒着违

反SOFA指导原则中一条或多条的风险，我们可以使用提取方法来将功能放在一个新的方法中，并从已有的方法中调用它。正如你看到的，这个技术有不错的好处，我们已经知道如何使用TDD来开发新的方法。

这个调查解释了为什么TDD自然地导致了好的可测试的代码——如果首先写了测试，那么这个方法很难是不可测试的——并且阐明了在“重构”的“红色—绿色—重构”步骤背后的基本原理。如果你在编写代码的同时不断地重构，让每一个独立的改变看起来都很小并且最低限度地占用你的时间和注意力，那么你的代码将趋向于漂亮。当你从大的方法中提取小的方法时，你再确定协作者，通过选择好的名字来描述代码的目的，并插入嫁接来帮助可测试性。当叙述性地重命名了一个变量时，你就已经有文档化设计的意图了。

但是如果你持续地用新的功能来封装你的代码而 **没有** 随时用重构，等到重构最终变成必需（它会最终变成必需的）的时候，重构会更痛苦并且需要9.2节和9.3节介绍的重要的辅助工具。简言之，重构从一个使用额外时间的后台活动，以增加客户价值为代价，突然变成了一个要求你的专心和专注的前台活动。

由于程序员是乐观主义者，所以我们经常会想：“这在我身上不会发生；我写了这个代码，所以我足够了解它以致重构不会那么痛苦。”但事实上，你的代码在被部署的那一刻起就变成了遗留代码，而你继续前进关注于另一块代码。除非你有一个时光机并可以和之前的自己交谈，你可能不会记起当你写原始的代码时你在想什么。所以代码的清楚度必须不言而喻。

这个持续性重构的敏捷观点应该不会令你惊讶：就像开始、测试或需求收集一样，重构不是一个一次性的“阶段”而是一个持续的过程。在第12章中，我们将看到持续性与阶段性的观点对部署和操作同样有效。

敏捷模式的基本特点使它能极好地符合软件维护的需求，这可能是一个惊喜。实际上，我们可以把敏捷模式看成是完全没有部署阶段，但是在生命周期的一开始就进入了维护模式。

9.10 深入学习

重构并不是修正遗留代码的唯一工作，但是正如我们所看到的，重构是工作最主要一部分。把重构做得更好的最好的方法就是多做。一开始，我们建议你浏览Fowler关于重构的书，对目录中的多种重构方法有一个大致的了解。我们推荐Ruby专有的版本（Fields et al. 2009），因为并不是所有的静态类型语言中的味道或重构方法都会在Ruby中出现。版本对其它流行的语言来说也是有效的，例如Java。我们在本章中只介绍一小部分，在图9.21中列出了更多。随着你有更多的经验，你将不需要每次都查阅分类就能发现重构的机会。

代码味道由敏捷活动产生。再次强调，我们只介绍了一个更广泛的分类目录中的一小部分，图9.22列出了更多。我们也介绍了一些简单的代码度量，经过四十年发展的软件工程，很多其它的东西都被开发出来以获取代码质量，在软件维护的成本和收益方面做了很多分析和经验学习的研究。Robert Glass（Glass 2002）写了一个 **软件工程中的事实和谬论** 的简洁的集合，根据经验和学术文献并集中于维护活动中感知的和真实的成本对比。

M. Feathers. **Working Effectively with Legacy Code**. Prentice Hall, 2004. ISBN 9780131177055. URL `http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052`.

分类	重构方法		
构成方法	**提取方法**	用方法替代临时变量	引入解释变量
	用方法对象替代方法	内联临时变量	分离临时变量
	移除参数分配	替代算法	
组织数据	自包含字段	用对象替代数据值	将值改为引用
	用对象替代数组/哈希	**使用符号常量替代魔法数**	
简化条件	**分解条件**	合并条件	引入断言
	用多态替代条件	用多态替代类型代码	用监护子句替代嵌套条件
	合并重复的条件碎片	移除控制标记	引入Null对象
简化方法调用	重命名方法	添加参数	从修饰符分开查询
	用明确的方法替代参数	保存整个对象	用异常替代错误代码

图 9.21: Fowler书中更多的重构方法，其中加粗的是我们在本章中介绍过的

重复代码	临时字段	过大的类	过长参数列表
发散式变化	特性依恋	基本类型偏执	元编程疯狂
数据类	冗余类	投机通则	并行继承层次
拒绝继承	消息链	中间人	不完整类库
过多注释	选择语句	不同接口的交换类	

图 9.22: Fowler和Martin描述的一些代码味道

J. Fields, S. Harvie, M. Fowler, and K. Beck. **Refactoring: Ruby Edition** . Addison-Wesley Professional, 2009. ISBN 0321603508. URL `http://www.amazon.com/Refactoring-Ruby-Jay-Fields`.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. **Refactoring: Improving the Design of Existing Code** . Addison-Wesley Professional, 1999. ISBN 0201485672. URL `http://www.amazon.com/Refactoring-Improving-Design-Existing-Code`.

R. L. Glass. **Facts and Fallacies of Software Engineering** . Addison-Wesley Professional, 2002. ISBN 0321117425. URL `http://www.amazon.com/Facts-Fallacies-Software-Engineering-Robert/dp/0321117425`.

R. Green and H. Ledgard. Coding guidelines: Finding the art in the science. **Communications of the ACM** , 54(12):57–63, Dec 2011.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. **Communications of the ACM** , 21(6):466–471, 1978.

R. C. Martin. **Clean Code: A Handbook of Agile Software Craftsmanship** . Prentice Hall, 2008. ISBN 9780132350884.

O. Nierstrasz, S. Ducasse, and S. Demeyer. **Object-Oriented Reengineering Patterns** . Square Bracket Associates, 2009. ISBN 395233412X. URL `http://www.amazon.com/Object-Oriented-Reengineering-Patterns-Oscar-Nierstrasz/dp/395233412X`.

C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. **Software, IEEE** , 18(6):42–50, 2001.

注释

[1]http://campfirenow.com
[2]http://basecamphq.com
[3]http://en.wikibooks.org/wiki/Ruby_Programming/RubyDoc
[4]http://api.rubyonrails.org
[5]http://paulschreiber.com/blog/2010/06/15/rake-task-extracting-database-contents/
[6]http://c2.com/doc/oopsla89/paper.html
[7]https://vimeo.com/24668095

9.11 建议的项目

练习9.1. 你的任务是为一个假想的遗留的选课系统（在加州大学伯克利分校称之为TeleBears）设计一个RESTful的应用程序接口。这个系统（不包括测试和数据库架构）的一个自外而内的描述如下：

- 一个课程有一个开课系、一个课程编号、课程名称。例如，“计算机科学，169，软件工程”。
  - 学期（秋季、春季或夏季）以及开课的年份
  - 教学楼及教室号
  - 每星期授课的时间和天数
  - 每周小型的会议的天数和时间（不是所有课程都有）
  - 教员
  - 可以选的学生人数限制

  每个课程都设置有一个唯一的ID，叫做控制编号。

简要画出一个UML图来描述这样一个系统的设计，以及一组RESTful的路线（表格中或者是一个简化的文件`routes.rb`或者是一个类似于图5.18中的一个表）来支持至少以下操作：

- 一个根据系名、教员名（部分匹配也可以）以及学期的关于课程的混合搜索。
- 获得一个课程设置的会议时间（授课部分、背诵部分等等，每个都有唯一的时间槽ID、天数及时间）。
- 将一个学生登记到课程设置中。

练习9.2. 给你练习9.1的设计，评估这样一个变更请求的影响：允许一个课程的多个同时的“阶段”在一学期中进行。这个改变可能会影响架构、课程交互、如何搜索和处理等等。

练习9.3. 选一个已有的有相对直接的用户界面的外部Web服务，使用Cucumber来创建它的综合级别的鉴定测试。你可以使用`mechanize`工具包来允许Cucumber运行在一个远程站点上。

练习9.4. 确定一个你将要检查的工作的遗留软件系统。建议是，你可以使用开源的Rails项目开源Rails[1]的列表，或者选择一到两个使用这本书的学生创建的项目：ResearchMatch[2]，这会帮助匹配学生和在大学中的研究机会，以及VisitDay[3]，这能帮助你管理好学生和教员之间的会议。

从这些项目中选择一个，克隆并创建这个代码库的分支，让应用在开发环境中运行起来。这将很可能需要创建一个开发数据库，建立`config/development.rb`来匹配，并从`db/schema.rb`创建数据库模式。

练习9.5. 继续练习9.4，试着获得在开发环境中运行的测试组件。一旦测试运行起来，使用SimpleCov来评估测试的覆盖率。
提示： 如第8章所描述的那样，你可以在RSpec配置文件`spec/spec_helper.rb`中添加SimpleCov）。

练习9.6. 继续练习9.4，收集一些代码质量或代码味道的度量。你可以使用本章中介绍的工具，如`reek`和`metric_fu`，或者你可以试着使用CodeClimate[4]，它把代码评审当作一项服务提供。

练习9.7. 继续练习9.4，选取应用的一个子系统（例如，模型、视图和与一类资源相关的控制器）并管理一个设计视图。确定当前设计的一个弱点，通过重构移除它。确保你在这个地方已有测试覆盖并保证你的重构没有改变已有的功能。

练习9.8. 继续练习9.4，管理一个细节的代码检查，审阅一个重要的源代码。确认文件中一个或多个代码味道并通过重构移除它们。确保你在这个地方已有测试覆盖并保证你的重构没有改变已有的功能。

10 项目管理：Scrum、结对编程、及版本控制系统

Fred Brooks, Jr.（1931——）是经典的基于领导IBM OS/360系统经验的软件工程书籍《人月神话》的作者，他管理System/360项目和直接向IBM主席T.J. Watson Jr报告。System/360是第一代有跨产品家族的指令集兼容架构的计算机，许多人认为它是第一个真正意义上应用了"计算机架构"术语的系统。因为在计算机架构，操作系统，和软件工程上的卓越贡献，Brooks获得了1999年的图灵奖。

一个失败的团队里面没有成功者，一个成功的团队里面没有失败者。

——弗雷德·布鲁克斯（Fred Brooks），引用北卡罗篮球队主教练迪恩·史密斯（Dean Smith），1990

10.1 团队规模：两个匹萨大小的团队以及Scrum 308
10.2 结对编程 309
10.3 敏捷设计和代码评审 311
10.4 两个匹萨大小团队的版本控制：合并冲突 311
10.5 有效地使用分支 314
10.6 报告和修复代码：五个R 318
10.7 计划文档的视角 320
10.8 谬论和陷阱 327
10.9 结束语：团队、合作、以及四十年的版本控制 328
10.10深入学习 329
10.11建议的项目 331

重要概念

本章最大的概念就是团队规模和组织，以及跟踪团队开发的产品的配置管理。这些和敏捷生命周期相关的概念如下：

- **“两个匹萨（Two-pizza）”** 的团队是4~9人的规模。
- 自我管理的团队遵循 **Scrum** 模式，该模式需要一个成员充当 **产品经理** 的角色，一个充当客户的角色，以及一个来缓冲团队和外部琐事的 **敏捷教练**．团队成员所充当的角色随着时间在变换。
- **结对编程（pair programming）** 是一种用来提高团队内部沟通和代码质量的方式，通过两个人同时检查测试正在编写的代码。
- 好的 **版本控制** 实践：通过诸如 **Git** 等工具，解决4~9名工程师参与的项目中的代码管理问题。

对于计划文档生命周期，你需要从一个不同的视角来熟悉下面这些概念：

- **项目经理** 书写合同书，和客户及高层交涉，招聘和组织开发团队，解决争议以及书写项目配置的计划和项目本身。
- 大规模的团队可以通过以项目经理为首，下设不同的小组负责人的方式来组建，而每个小组的规模和敏捷开发类似。
- 通过 **视察** 让外部人员对当前的设计和未来的计划给予反馈，以及检查是否被执行了好的实践。
- **配置管理** 包括开发过程中的软件组件的 **版本控制**，通过这些组件构成的相关的工作程序的 **系统建设**。新版本程序的 **发布管理**，以及对已有产品的 **变更管理**。

不管是哪个生命周期，编程现在都是一个团体性的运动，这章所包括的技巧能够让团队变得成功。

10.1 团队规模：两个匹萨大小的团队以及Scrum

项目包括如下六个阶段：

(1) **激情**

(2) **幻灭**

(3) **痛苦**

(4) **寻找罪恶感**

(5) **惩罚无知者**

(6) **表扬未参与者**

——Dutch Holland（Holland 2004）

正如这本书中多次提过的，敏捷开发的生命周期通常以两周的迭代开始，以一个工作原型结束。每次迭代实现一些用户故事，并将其作为验收测试或者集成测试。涉众在每一次迭代之后检查产品，看是否满足了每个人的需求，以及定义接下来用户故事的优先级。

那些通过英雄式的程序员来完成任务的时代已经一去不复返了。虽然聪明的个人能够编写出具有突破性意义的软件，但是随着功能和质量的不断提升，这意味着软件开发现在首先是一个团体运动。因此，今天的成功不仅仅意味着你有很强的设计和编程能力，也意味着你要能够和其他人进行良好的合作并能够帮助你的团队取得成功。正如Fred Brooks所言：如果你的团队失败了，你不可能赢；如果你的团队胜利了，你也不可能输。

一个**scrum**在橄榄球的每个小片段中被举行。比赛停下来让队员在一起进行一个快速的"碰面"，为了重启比赛。

(1) 从昨天开始你做了什么？

(2) 你今天打算做什么？

(3) 存在什么障碍吗？

这些每日的scrum的好处就是，让大家明白团队里面的每个成员都在干什么，团队能够确认工作从而可以帮助其他人有更快的进展。

每周或是每两周去收集涉众的反馈，当把这些和敏捷迭代模型结合起来的时候，Scrum组织者更有可能使得项目快速地朝着客户想要的方向前进。Scrum使用 **冲刺（sprint）** 这个术语而不使用敏捷术语迭代。

一个Scrum有三个角色：

(1) **团队（Team）**——一个"两个匹萨"大小的团队用来完成软件交付。

(2) **敏捷教练（Scrum Master）**——一个缓存团队和外部之间分歧的团队成员。使团队专注于手上的任务，保证团队的规则，以及移除阻碍团队前进的障碍。一个例子就是保证团队的 **编码规范**，编码规范是用来保证代码的一致性和可读性的风格指导。

编码标准或 **风格表单** 对Rails[5], Python[6], JavaScript[7], C++[8], 和其他大多数编程语言。

(3) **产品经理（Product Owner）**——一个代表客户声音和定义用户故事优先级的团队成员（不能和敏捷教练是同一个人）。

Scrum依靠自我组织，其团队成员经常轮换角色。例如，我们推荐每个成员在不同迭代或者冲刺中轮流充当产品经理。

在以前工作的任何小组中，当选择技术路线的时候就会产生冲突。这部分取决于团队成员的性格，他们可能不能够快速地达成一致。一个用来解决冲突的方法就是，列出一个大家都赞成的条目，而不是大家的分歧点。这个方法能够让大家看到拥有比想象更多的共同点。这个方法使双方都能够理解对方的观点，即使他们不能全部达成一致。这一步能够减少对术语和假设的疑惑，而这些往往就是冲突的原因。

小结：
SaaS能够很好地满足两个匹萨大小的团队和Scrum，一个自组织的小团队每天都会见面。两个团队成员会充当额外的角色。一个是敏捷教练，他负责解决障碍和使团队保持专注。另外一个是产品经理，他代表用户的利益。使用结构化策略解决出现的争议是非常有帮助的。

■ **详细阐述：编码规范**

是团队里的每个成员都会去遵循的规则。它的目标是提高代码的一致性和可读性。例如，这里有一个Rails[9]的标准，以及Google提供的一个Python[10]的标准，JavaScript[11]的标准，以及其他几种编程语言的标准。

自我检查10.1.1. 对或错：当预先计划很难的时候，Scrum是最佳的选择。
◇ 正确。与以计划和命令及控制为中心的传统管理相比，Scrum更多的依靠及时的反馈。

现在给了一个团队的组织，我们已经准备好了编程。

10.2 结对编程

两个大脑好于一个。

——英国谚语

Dilbert对于结对编程
连载漫画Dilbert对结对编程幽默的评论在这两期[12] 漫画[13]里面。

极限编程（XP）是本书中所讲述的敏捷生命周期的一种变形，它提出了一种和过去软件开发不同的方式。在软件实践里面一个新的选择就是 **结对编程** 。它的目标就是通过多个人开发同样的代码来提高软件的质量。正如其名，两个软件开发者共享同一个计算机。每个人承担不同的角色：

- **驾驶者** ——编写代码并且从整体上思考如何完成当前的任务，在敲写代码的同时大声的解释其想法。
- **观察者** 或 **导航者** ——更符合汽车驾驶的术语检查敲进去的每行代码，并充当驾驶者的安全网。观察者也从整体上思考接下来需要强调的问题，并给驾驶者提供建议。

通常结对的成员会在执行任务的时候轮换角色。图10.1，显示了Pivotal Labs的工程师——Pivotal Tracker软件的制作者——他们每天大部分的时间用在了结对编程上。（Moore 2011）

图 10.1: Pivotal Labs的Sarah Mei和JR Boyens很喜欢结对编程。Sarah正在驾驶，而JR正在观察。虽然可以看到两个键盘，但是只有Sarah在编码；JR面前的计算机是用于文档和其他相关的信息，如Pivotal Tracker，正如你在右手边的图片所能见的，所有的结对工作站（在背景中可见）都是一样的，并且没有邮件和安装其他软件；另一台计算机是专用来检查邮件的。本图由Pivotal Labs提供

结对编程需要相互之间的协作，它需要许多的交流。它关注手头上的任务，两个一起工作的人增加了遵循好的开发实践的可能性。如果一个同伴静静地检查邮件，那么这就不是结对编程，只是两个人坐在一起而已。

结对编程有伙伴之间交换知识的作用，包括编程风格、工具技巧、公司进展、客户需求等。因此，为了扩展知识基础，一些团队有意在每个任务中交换一个同伴，这样使得最终所有的人都在一起工作过了。例如，对应四个人的混杂编程有六种不同的结对。

关于结对编程和单独编程的对比调查支持了结对编程有利于减少开发时间和提高软件质量的结论。例如（Cockburn and Williams 2001）发现和单独编程对比，结对编程在时间上有20%～40%的减少，且首次代码失败率为15%，而单独编程为30%。但是结对编程所花费的小时数比单独编程多15%。在微软有着10多年经验的专业程序员、测试员、和经理认可结对编程使他们工作的质量更高，并且产生了更高质量的代码（Begel and Nagappan 2008）。一份关于结对编程的研究总结到：当编程任务比较简单时——跟踪尺度上大概是一个点的任务，结对编程更加的快速，当任务比较复杂时，跟踪尺度上三个点的任务，会产生更加高质量的代码。在两种情况都比单独编程花费了更多的努力（Hannay et al. 2009）。

Pivotal Labs的经验表明这些研究还没有考虑那些对我们生产有负面影响的因素，我们现在越来越被连到一个现代世界：email、Twitter、Facebook等等。结对编程让两个程序员连续好几个小时专注于手头上的任务。事实上也确实是这样，在Pivotal Labs的新雇员回到家的时候都是精疲力尽的，因为他们还没有适应如此长时间的专注工作。

尽管结对编程需要花费更多的努力，但是大幅度提升敏捷和Rails生产力的方式就是在结对编程上进行“投资”。让两个大脑开发代码能够减少新软件上市的时间，以及提高后续产品的质量。如果你喜欢，我们推荐你尝试一些结对编程，许多开发者都喜欢结对编程。

小结：
当开始编码的时候，一个方式就是结对编程，其保证了更高质量和更少时间的开发，但是可能会带来更高的编程开销，因为两个人做同一件工作。结对被分为驾驶者和观察者，前者来完成手头上的任务，后者思考未来的挑战和给驾驶者提供建议。

自我检查10.2.1. 对或错：研究表明结对编程比单独编程更快，花费更少。
◇ 错误。虽然没有严格的能够说服所有专家的实验，而且也不清楚这些实验是否评估减少分心的影响，当前研究者一致认为结对编程比单独编程更昂贵——一个工作上要花费程序员更多的时间。

自我检查10.2.2. 对或错：一个结对最终会发现谁是最好的驾驶者，谁是最好的观察者，并且主要以这种角色进行。
◇ 错误。一个有效的结对会在两者之间转换角色，对个人而言既体会驾驶者角色，也充当观察者角色会更有效（同时也更有趣）。

对于工作在代码中有许多检查者和代码片段之间相互依赖的团队，你需要工具来帮助管理结果。

10.3 敏捷设计和代码评审

10.7节描述了使用设计检查和代码检查来提高软件产品的质量。大多数公司使用敏捷方法时不进行设计和代码审评。

例如，Pivotal Labs的传统是结对编程使得这些检查浮出在了表面，因为代码在开发的过程中被不断的检查。在GitHub中，正式的代码检查被 **pull（获取并合并）** 请求所替代，在其中开发者请求其代码整合到主代码库中，正如我们在下一节所描述的。团队中的所有人看到这些请求意见从而决定怎样更新他们自己的代码。如果任何人有疑问，可以举行一个关于该pull请求的在线讨论。我们可以使用诸如Campfire或者Github问题跟踪工具（参见10.6节），这可能会使得pull请求完成之前，对有疑问的代码进行更改。因为每天会有许多的pull请求出现，这些“小增量检查”会持续出现，因此不需要专门的会议。

10.4 两个匹萨大小团队的版本控制：合并冲突

> 在美国和全世界内，有一群有意思的人在做社交编程。最有趣的不是员工在Twitter上面做什么，而是他们在GitHub上面做什么。
>
> ——AI, Gore, 美国前副总统，2013

好的版本控制实践对一个团队比对个人更重要。代码库是怎样被管理的？如果一个团队成员在修改一些文件的时候碰巧产生了冲突怎么办？在一个给定文件中的那些行是什么时候被修改的？以及它们是被谁修改的？一个开发者在开发新功能时，怎样才不会把问题引入到稳定的代码中？当一个软件被一个团队而不是个人开发时，版本控制能够通过使用 **merging** 和 **branching** 来强调这些问题。两种任务都涉及把许多开发者的修改合并成一个单一的代码库，这其中有时候还涉及手工解决这些有冲突的改变。

这部分及下一部分假定熟悉Git的基本版本控制操作。A.6节总结了基本的用法，在10.10节提供了更深入的资源。

工作在同一个功能的小团队通常使用 **shared-repository** 模型来管理代码库：代码库有一个特殊的副本（ **origin** ）被指定为为权威，所有的开发者都同意把他们

代码文件：ch_teams/code/git_conflict_markers.txt

```
1 Roses are red,
2 Violets are blue.
3 <<<<<<< HEAD:poem.txt
4 I love GitHub,
5 =======
6 ProjectLocker rocks,
7 >>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:poem.txt
8 and so do you.
```

图 10.2：当Bob试图合并Amy的改变时，Git在`poem.txt`中插入冲突标记来显示合并冲突。行3用<<<显示了冲突区域的开始；直到===（第5行）中的任何事显示了`HEAD`（Bob本机代码库中的最后一个提交）中的文件的内容，以及直到冲突结束标记tg>>（第7行）中的任何事，显示了Amy的改变（文件作为出现在Amy的冲突提交中，其提交ID在第7行）。第1、2、8行没有被Git改变或自动合并

许多早期的VCSs 例如Subversion只支持shared-repository模型，以及"one true repo"经常被称为 **master**，一个在Git中表示很不同的术语。

的修改提交到origin以及定期地pull origin来得到其他人的修改。特别地，Git本身不关心哪个副本是权威的——如果他们拥有代码库的权限，任何人都能够通过 **pull** 得到origin的修改以及通过 **push** （上传）来改变origin的代码。但是对于小团队，通常的做法（也是惯例）是把origin代码库放在云端，例如在Github和ProjectLocker上。团队成员把代码库 **复制** 到他们的工作机器上，做他们的工作，进行他们的提交，并且把这些提交push到origin上。

我们已经知道`git push`和`git pull`能够将你的代码库副本备份到云端，但是在一个团队环境中，这些操作有另外重要的意义：如果Amy把她的修改提交到了她的代码库，只有当她执行了一个push，并且她的同事Bob执行了一个pull，Bob才可以看到这些改变。这就可能引起了 **合并冲突**：

(1) Amy和Bob每个人都有origin代码库的最新的一份副本。

(2) Amy对文件A进行了一系列修改并且提交。

(3) Amy对文件B进行了修改并且提交了一份单独的修改。

(4) Amy把她的提交推送到origin。

(5) Bob对文件A进行了修改并提交了，但是没有对文件B进行任何改动。

(6) Bob试着push他的提交，但是他被阻止了。因为自从Bob最后一个pull之后，origin代码库出现来另外的commit，Bob在能够push他的改变之前，必须把他的origin代码库副本更新。

`git pull` 实际上包含了单独的命令`git fetch`，其被用来从origin代码库复制最新的副本。以及`git merge`，其试着使origin代码库和本机上的代码库一致。

所展示的额外提交并不意味着一定产生了有冲突的改变——比如，Bob不受Amy对文件B的提交的影响（步骤3）。在我们的例子中，虽然2和5步骤确实代表了对同一个文件有冲突的改变。当Bob执行`git pull`的时候，Git将要 **merge** Bob和Amy对文件A的改变。如果Amy和Bob修改了文件A的不同部分，Git将会自动整合文件A不同部分的修改并且提交整合到Bob的代码库中。但是，如果Amy和Bob对文件A的修改有重合的部分，正如图10.2所示，Git将认为自动整合是不安全的，它将会把一个有冲突但是没有提交的版本留在Bob的代码库中，让Bob手动修改和提交。

在任何一种情况情况下，一旦Bob提交了文件A，他就能够提交他的修改，只要origin代码库成功地反映了Amy和Bob的修改。在Amy下次pull的时候，她将会得到她和Bob修改之后的合并版本。

这个例子显示了一个重要的规则：**总是让提交优先于合并**（因此在pull之前隐含了执行一次merge）。提交确保在试着合并其他人的修改之前，你有你自己工作的

一个稳定的镜像保存。Git警告你如果你试着合并你没有提交的文件，同时提供了不顾警告仍然进行的情形中恢复的机制（见图10.3）。但是如果你 **经常尽早提交**，你的生活将会变得更加轻松，这比你从你的错误中恢复更加的简单。

因为在一个团队里面工作意味着许多开发者都在修改文件的内容，图10.4列出了一些有用的Git命令以用来帮助跟踪谁在什么时间做了什么修改。图10.5显示了一些有用的标记符表示用来代替40位的冗长的Git提交ID。

Git的高级特性像 **rebasing** 以及 **commit squashing** 能够把许多小的提交变成更少但是更大的提交，以使得公开可见的修改历史更加清晰。

- `git reset --hard ORIG_HEAD`
  在进行合并之前，恢复仓库到最后提交的状态。
- `git reset --hard HEAD`
  恢复仓库到最后提交的状态。
- `git checkout` **commit** `--` **[file]**
  恢复一个文件，或者如果忽略了整个仓库，恢复到它 **提交** 的状态（参见图10.5中有关一个提交后40位的哈希值的方法。）能够用来恢复之前使用`git rm`删除的文件。
- `git revert` **commit**

  恢复由 **提交** 引入的改变。如果这个提交是一个合并的结果，有效地恢复合并，并且把当前的分支保持在合并之前的状态。Git试图恢复由这个提交引入的改变，而不影响这个提交之后的其他改变，如果提交发生在很久以前，可能需要手动解决冲突。

图 10.3: 当合并出现错误的时候，这些命令能够帮助你恢复、撤销所有的或者部分的冲突

`git blame` **[file]**	标注一个文件的每一行来显示谁最后修改和何时修改的。
`git diff` **[file]**	显示当前工作版本的 **file** 和最后提交版本之间的区别。
`git diff` **branch [file]**	显示当前版本的 **file** 和它最近在这个 **分支** 上提交的不同（见10.5节）
`git log` **[ref .. ref] [files]**	显示通过 **ref** 指定的两个提交之间的影响所有 **files** 的日志实体，如果参数忽略了，就显示影响这些文件的整个日志历史。
`git log --since="` **date** `"` **files**	显示自给定的日期开始的影响所有 **files** 的日志实体（例子：`"25-Dec-2011"`，`"2 weeks ago"`）。

图 10.4: 用来跟踪什么文件什么时候被修改了的命令。许多命令接受`--oneline`选项来产生一个紧促形式的报告。如果一个可选参数 file 被忽视了，默认是所有被跟踪的文件。注意所有的这些命令都有 许多 的选项，你能够通过`git help`来看到这些选项

HEAD	当前分支上最近的提交版本。
HEAD~	当前分支上最高优先权的提交（HEAD~ **n** 指第 **n** 优先的提交）。
ORIG_HEAD	当一个合并被执行时，HEAD更新到最新合并的版本，并且ORIG_HEAD指代合并之前的提交状态。如果你使用git diff查看每个文件作为合并结果是怎样被改变，是很有用的。
1dfb2c~2	2个提交优先于ID前缀是1dfb2c的提交。
" **branch** @{ **date** }"	分支上最后一个提交优先于指定的 **date** 的提交（见图10.4中的日期格式），其中HEAD指代当前分支。

图 10.5: Git命令中一些方便的方法来指代一些提交，而不是使用一个40位长的提交ID或者一个特殊的前缀。git rev-parse expr 把上面任何一种形式的表达式转换成一个完整的提交ID

小团队的冲突管理小结：

(1) 小团队通常使用"shared-repo"模型，这种模型中push和pull使用单一的权威的代码库副本。在Git中，权威副本用origin来指代，并且其通常存储在GitHub的云端上或者公司内部的一台服务器上。

(2) 在你修改文件前，**提交** 你本机的修改并且 **合并** 你不在时由其他人做的修改。在Git中，从origin代码库中合并修改的方法是git-pull。

(3) 如果合并不能够自动进行，你必须查找冲突文件中的 **冲突标记**，并且手动修改，然后提交并把修复之后的版本推送到服务器上。使用Git，当冲突文件被再次提交时就意味着冲突解决了。

■ **详细阐述：远程协作：公共的代码库的fork-and-pull**

Git是被设计来支持许多开发者的超大型项目的，例如Linux kernel。**fork-and-pull** 管理模型让各个小组能够独立工作，以及能够分开不同的修改从而不影响其他人的结果。一个远程的小组能够fork你的代码库[14]，这样能够在Github上创建自己的副本来接受他们的push。当他们准备把稳定的代码贡献到代码库的时候，各个小组创建一个pull请求[15]，要求你去合并回其代码库中被选择的提交到你的代码库中，pull请求允许合并两个选中的代码库，而其他代码库保持分离。

自我检查10.4.1. 对或错：如果你试图进行git push，但是失败了。传回来一个这样的错误消息：例如"Non-fast-forward (error): failed to push some refs，"这意味着你的代码库版本和origin代码库版本之间的某些文件存在着一个合并冲突。
◇ 错误。这只是意味着代码库版本缺失一些origin版本中的提交，除非你合并了那些缺失的提交，否则你的推送将不会被允许。这些缺失提交的合并 **也许** 会导致一个合并冲突，但是通常情况下不会。

10.5 有效地使用分支

除了对你的工作采取镜像和备份，版本控制也让你同时管理一个项目的代码库的多个版本，例如，让你的团队的一部分去实验一些新的功能而不影响正在工作的代码，或者修复先前一个许多用户还在使用的版本的代码。

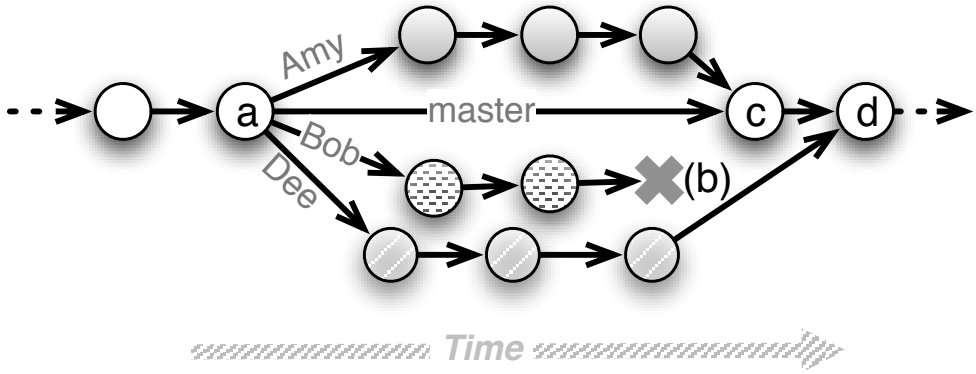


图 10.6：每一个圆圈代表一个提交，Amy、Bob和Dee每个人都是从同一个提交（a）节点开始分支的，工作在不同RottenPotatoes功能上。一些提交之后，Bob觉得不值得继续开发了，因此他删除了它的分支（b）；与此同时，Amy完成了她的测试和代码，并且把她的功能分支合并回了主分支，创建了合并-提交（c）。最后，Dee完成了他的特征，但是因为Amy的合并—提交（c）导致主分支已经被改变了，Dee不得不做一些手工冲突解决来完成他的合并-提交（d）

分支 是针对如下一些情形所设计的。不是仅仅将提交认为是一个快照的序列，我们应该把它们看成是一个提交 **图**：一个分支是由创建这个代码树的一个基于某个特定提交的。和真正的树分支不一样，一个代码库分支不仅仅从主干分开来，也会被最终合并回主干。

从那一点开始，新的分支和它分离的那个是分开的：虽然取决于项目的需要，对一个分支的提交可能会合并回另一个，但是对一个分支的提交不会影响另一个分支。事实上，分支甚至能够从其他分支分开，但是过度复杂的分支结构没有带来很多的好处，相反会导致维护的困难。

我们提供了两种通用的既能够同时使用也能够分开使用的分支管理策略，这两种都试着使主分支一直包含一个稳定工作版本的代码。图10.6 显示了一个 **功能分支**，它允许一个开发者或者一个小团队去完成一个特殊的特征，而这些改变在没有被完成和测试之前是不会影响到主分支的。如果这个特征被合并到主分支，并且随后决定删除掉它（也许是它没有达到预期的客户价值），与合并分支相关的特定提交有时能够被撤销，只要主分支上没有太多依赖新特征的修改。

Flickr 开发者现在使用[16] 完全没有功能分支的repository——提交总是发生在主分支上。

图10.7 显示 **发行分支** 在一个特定发行版本中如何被用来修复问题。它们被广泛用于处理非SaaS产品，如库，或其发行远离主分支，可能大幅偏离最近的发行分支的Ruby gem。例如，Linux kernel，其开发者每天检查上千行的代码，使用分支版本来标出稳定和长期支持的发行版本kernel。发行版本通常会接受来自开发或者主分支的多次合并，并且会贡献多次合并到其中。SaaS很少使用发行版本，因为趋势是持续整合/持续部署（参见1.6节）：如果你一周部署多次，那么部署的版本在时间上没有主分支同步，因此你也许会直接用主分支来部署。我们在第12章中更进一步地讨论持续部署。

GitFlow[17]，一个获得了许多好的实践的分支管理策略，也许在更大项目的长期支持分支中很有用。

图10.8 显示了一些用来管理Git分支的命令。最新创建的Git代码库以一个叫做`master`的单一分支开始。在任何给定的时间， **current branch** 指代你当前正在工作的代码库的副本。因为通常每个副本包含所有的分支，你能够在同一个代码库的分支中快速地来回切换（参照视频10.5.1关于这样做的警告）。

当有多个分支的时候，如何指定哪个分支接受push或者pull呢？正如图10.8所示，我们到目前为止一直使用的`git push`和`git pull`命令实际上省略了特殊的情形——这些命令在使用分支的时候也能够很好地处理push和pull。

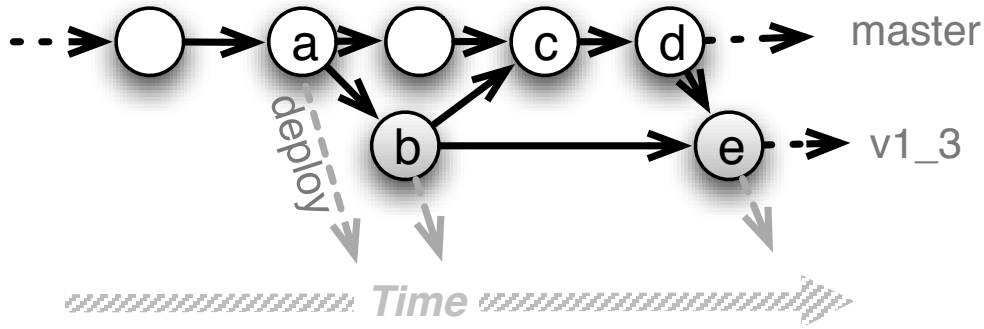


图 10.7: （a）一个新发行分支被用来"快照"RottenPotatoes的1.3版本，发行版本中发现一个错误，并且这个错误的修复提交在了发行分支中（b）；应用程序从发行分支中被重新部署了。包含修复的提交被合并到来主分支中（c），但是来自发行版代码的主分支中的代码已经大量进化了，因此需要手动调整来修复代码。与此同时，开发团队在主分支上找到了一个特别严重的安全缺陷并且在提交（d)中修复了。因为我们不想把主分支中除这个修复以外的任何修改整合到发行版本中，使用`git cherry-pick`可以把包括这个安全修复的特殊提交合并到发行版本（e)中

视频10.5.1: 用Git使用分支.
光盘文件：ch_teams/mov/branches.mp4
这个视频段展示了如何在Git中创建和管理一个新的分支（例如，去开发一个新的特征），如何把分支的修改合并到先前它分开的主干中，以及在一些情况出现问题的时候，如何来撤销分支的合并（例如，后来发现特征有错误需要回滚）。它也强调了一个很重要的警告，及展示了为什么你需要在切换到一个不同的分支之前，总是需要提交当前分支的修改。

分支小结：

- 分支允许一个代码库有不同的变化。例如，功能分支使得在开发一个新特征的时候不影响正在工作代码的稳定性，以及发行分支对从主开发中分离出来的先前发行版本进行错误修复。
- 从一个版本到另一个版本的归并修改（例如，一个功能分支回到主分支中），可能导致某些文件产生合并冲突，因此请在合并和转到一个不同的分支工作之前进行提交。
- 利用敏捷开发和SaaS，功能分支通常是短期存在而发行分支是不常见的。

- `git branch`
  列出仓库中存在的分支，用*指示当前的分支。如果你在一个类Unix系统中使用`sh`或者继承于`bash`的shell，把this code[18]放在`~/.profile`中，当你在一个Git仓库目录下，这个shell会及时显示当前的Git分支。

  代码文件：ch_teams/code/git_prompt.txt

```
export PS1="[`git branch ---no-color 2>/dev/null | \
  sed -e '/^[^*]/d' -e 's/* \(.*\)/\1/'`]% "
```

- `git checkout` **name**
  转到已存在的 **name** 分支中。
- `git branch` **name**
  如果 **name** 分支存在，转到它中去；否则会创建一个叫 **name** 的分支，但是不会跳转到它中。快捷方式`git checkout -b` **name [commit-id]** 创建和跳转到一个基于 **commit-id** 的新分支，默认是当前分支中最近的提交。
- `git push` **[repo] [branch]**
  把 **branch** 上的改变（提交）推送到远处的代码库 **repo** 中。（你对一个给定的分支进行首次操作时，它在远处的 **repo** 上创建分支。）没有参数，把当前本机的分支推送到当前分支的远程，或者默认叫做`origin`的远端。
- `git pull` **[repo] [branch]**
  把远端的 **repo** 中的 **branch** 分支获取并合并提交到本机 **当前** 分支 **（即使当前分支的名字和你正在获取的分支的名字不匹配——当心！）** 。为了获取你本机没有的相应的远端分支 **foo** ，首先通过名字使用`git checkout -b foo`创建一个本机分支，并且跳转到它，然后使用`git pull origin foo`。没有参数， **repo** 和 **branch** 默认相应是`git config branch.` **currentbranch** `.remote`和`git config branch.` **currentbranch** `.merge`的值，它是由某些Git命令自动设置的，可以通过`git branch --track`改变。如果你以常见的方式设置一个新的仓库， **repo** 默认是`origin`， **branch** 默认是`master`。
- `git remote show` **[repo]**
  如果省略了 **repo** ，就显示一个已经存在的远端仓库的列表。如果 **repo** 是一个已经存在的远端仓库的名字，显示 **repo** 中的分支，和哪个本机分支被设置跟踪它们。也显示哪一个本机分支没有随 **repo** 进行更新。
- `git merge` **branch**
  合并所有 **branch** 中的改变到当前分支。
- `git cherry-pick` **commits**
  不是合并来自于某个给定的分支的所有改变（提交）， **只** 是执行当前分支中叫做 **commits** 引入的改变。
- `git checkout` **branch file1 file2...**
  对于每一个文件，合并 **branch** 版本中的文件的不同处到当前分支版本中的文件中。

图 10.8: 处理分支和合并的常用Git命令。分支管理涉及合并；图10.3 告诉了当合并出现问题的时候如何撤销

■ **详细阐述：长期存在的分支和衍合（Rebasing)**

当你在功能分支上工作时，其提交将会同主干分开；如果你在上面长期工作，“大的合并”可能会因为有许多的冲突而使得你非常痛苦。这种痛苦可以通过频繁的 **衍合（rebasing）** 来缓解，一个你从其他分支逐步合并最近修改的操作，然后告诉Git去重新安排，使得你的分支看起来是从最近的一个提交分支过来的。当你使用长期存在的分支、如发行分支或长期存在的实验分支时，衍合非常有用，如果把用户故事分解成可管理的大小（参见7.3节）并且进行频繁的部署（参见12.3节），衍合应该在敏捷的SaaS开发中很少必要。

自我检查10.5.1. 描述一个需要往两个方向进行的合并情形——一个功能分支中的修改合并到主分支中，以及一个主分支中的修改合并到功能分支中（在Git中，这被称为 **交叉合并**）。

◇ Diana创建了一个新的分支用于一个功能上的工作。在她完成这个功能之前，一个很重要的错误被修复了，并且这个修复被合并到了主分支中。因为这个错误是Diana特征交互代码的一部分，她把主分支中的这个修复合并到了她自己的功能分支中。最后，当她完成这个分支的时候，她的功能分支被合并回了主分支中。

10.6 报告和修复代码：五个R

错误的出现是不可避免的。如果足够幸运，你会在软件投入使用之前发现，但是产品错误总是会出现的。团队里面的每个人都会同意管理错误生命周期的进程：

(1) 报告（**R**eporting)一个错误。

(2) 重现（**R**eproducing)一个问题，或者重定义（**R**eclassifying)它“不是一个错误”，或者“不可修复”。

(3) 建造一个回归（**R**egression)测试来验证这是一个错误。

(4) 修复（**R**epairing) 这个错误。

(5) 重新发布（**R**eleasing)修复的代码。

任何涉众都可能在服务器端或者客户端发现（**R**eport)SaaS代码中的错误。然后开发或者QA团队中的一个成员就必须重现（**r**eproduce)这个错误，文档记录下运行环境和引发这个错误的必要步骤。这个过程可能会因为各种原因导致重定义（**r**eclassifying)这个错误不是一个错误：

- 这不是一个错误，但是是设计中定义了的需求，对它进行提高或者改变一个动作的工作正在进行。
- 这个错误是属于没有部署或者不再支持产品中的代码。
- 这个错误只会出现在不支持的用户环境中，例如一个非常旧的浏览器缺乏这个SaaS应用的必要功能。
- 这个错误已经在最新版本中被修复了（在SaaS中，用户总是使用最新版本并不常见）。

一旦一个错误是真实可再现的，它就进入了一个错误管理系统。目前已经有很多这样的系统，小规模和中等规模的需求可以被一个工具满足：Pivotal Tracker允许把一个错误当作一个故事而不是一个功能，它给这个故事评0分，但是仍然允许它像一个普通的用户故事一样被完成。这个工具的优势就在于Tracker为你管理这个错误的生命周期，因此已有的实现用户故事的进程也适用于错误修复。例如，修复这个错误必须比相关的其他工作有更高的优先权；在一个瀑布模型过程中，这个也许意味着和维护过程中相关的重要错误同一个优先级，但是在敏捷过程中，它通常意味着和用户需求的相关新功能一个优先级。使用Tracker, 产品经理可以根据错误的重要性和对用户的影响程度来调整关于错误的故事在其他故事之上或者之下。例如，产品中可能会导致数据丢失的错误将会得到非常高的优先级。

"1级严重性"
在Amazon.com的错误需要负责的工程师发起一个控制在15min内的关于学习这个错误的电话会议——一个更加严格的相应要求而不仅仅是一个随叫随到的修复者（Bodík et al. 2006）。

下一步工作就是修复（**R**epair), 通常 **第一步** 是编写 **最简单可能性** 的自动测试，这些测试在有错误存在的情况下不会通过， **然后** 就是修改代码以使得测试通过。你现在作为一个TDD 参与者对这个应该很熟悉了，但是即使在非TDD环境中这个惯例也是正确的： **没有错误在不经过测试的情况下能够得到解决。** 根据错误的情况，可能需要单元测试、功能测试、集成测试或者这些测试的综合测试。 **最简单** 意味着这些测试尽可能取决于预先条件，和错误相关性很大。例如，简化一个RSpec单元测试意味着在测试行动之前或者在阻塞（block)**之前**减少设置，且简化一个黄瓜场景（cucumber scenario）意味着减少**假定**或者**背景**步骤。这些测试通常会添加到常规的回归测试单元中以确保这个错误不会在再次出现的时候没有被检测。一个复杂的错误可能会需要许多提交来修复；在BDD+TDD项目中一个常见的策略是，有失败或者遗失测试的提交应该只有在这些测试通过之后才应该合并到主开发分支中。

许多错误跟踪系统能够自动的交叉引用错误报告，这些报告的提交ID中含有相关的修复和回归测试。例如，使用GitHub的钩子服务[19]，一个提交能够自动地标上Tracker中相应的错误或者特性故事ID，并且当提交被推送到GitHub时，故事会自动地标为已处理了。取决于团队协议及使用的错误管理系统，错误可以在找到包含对应修复的发行版本后，或者在发行版本真正发布后立即"关闭"。

正如在第12章将会看到的，在许多敏捷团队的发行将会十分频繁，缩短了错误的生命周期。

错误修复的5R小结：

- 一个错误在被解决之前，必须要被报告（**r**eported)、可重现（**r**eproduced)、被回归测试验证（**r**egression)以及被修复（**r**epaired)，所有在修复错误之前的都是可发布（**r**eleased）的版本。
- 只有在一个自动测试让我们了解了导致错误的原因之后，错误才可以被关闭。
- 对于仅仅是增强功能的请求，或只出现在过时版本中，或只出现在不支持的环境中的有些错误报告，此时也许需要重新分类将它们分类成不会被修复的错误。

自我检查10.6.1. 你为什么认为"修复错误"故事在Tracker中值0分，即便它们和普通的用户故事遵循相同的生命周期?

◇ 一个团队的速度将会通过修复错误被人为地提高，因为他们将会首先实现功能得到分数，然后通过让其正确实现得到更多的分数。

自我检查10.6.2. 对或错：一个和其他服务（例如，通过Twitter验证)交互的错误在回归测试中不能够被捕获，因为需要控制Twitter的行为的必要条件。
◇ 错误。集成级别的mocking和stubbing，例如使用FakeWeb gem[20] 或者8.7节中所描述的技术，几乎总是能够被用在一个自动测试中来尽量减少外部条件以重现这个错误。

自我检查10.6.3. 对或错：一个由浏览器渲染了错误的内容或者JavaScript问题导致的布局的错误，也许可以通过人来手动重现，但是它不能够在一个自动回归测试中被捕获。
◇ 错误。如Jasmine和Webdriver（6.7节）的类似工具能够被用来开发这些测试。

10.7 计划文档的视角

在计划文档进程中，项目管理从项目经理开始。项目经理是项目的负责人：

- 他们书写合同以从客户那儿获得项目。
- 他们把已有的员工和新招进来的员工组成开发团队。
- 他们通常会写团队成员表现的绩效评价，这通常会大幅的影响工资。
- 从一个Scrum的角度（10.1节), 他们充当产品的拥有者——主要接触客户—— **并且** 他们也充当敏捷教练，因为他们和上层管理团队联系，而这些管理人员给团队提供资源。
- 正如7.10节所述，项目经理也估计开销，制定和维护计划书，强调可能遇到的风险，以及怎样克服或避免这些风险。
- 正如你对计划文档进程所预想的一样，项目经理必须书写他们的项目管理计划。图10.9 根据相应的IEEE标准，给出了一个项目管理计划的大纲。

所有这些责任的结果就是：如果项目有问题，项目经理要为这些错误负责任。引用一个教科书作者在其引言部分对项目管理的描述：

> 然而，如果对每个[有问题]的项目进行事后检查，很可能会得出一个一致性的结果：项目管理非常差。
>
> ——Pressman 2010

关于增加项目成功的可能性，我们讨论了项目管理的四个主要任务：

(1) 团队的大小、角色、空间及沟通

(2) 管理人员和冲突

(3) 检验和度量

(4) 配置管理

1、团队大小、角色、空间及沟通

计划文档进程能够扩展到更大的团队规模，其每一个小组的组长向项目经理报告。然而，每一个小组通常会保持两个匹萨大小的规模，正如我们在10.1节中所见的。我们推荐的规模大小是3∼7个人（Braude and Berstein 2011)，至多10个人

1. 项目概览	5.2 项目工作计划
1.1 项目总结	5.2.1 工作活动区
1.1.1 目的，范围和对象	5.2.2 调度分配
1.1.2 假设和约束	5.2.3 资源分配
1.1.3 项目交付	5.2.4 预算分配
1.1.4 计划和预算总结	5.2.5 采购计划
1.2 计划进展	6. 项目评估和控制
2. 参考	6.1 需求管理计划
3. 定义	6.2 范围变更控制计划
4. 项目背景	6.3 调度控制计划
4.1 进程模型	6.4 预算控制计划
4.2 进程提高计划	6.5 质量确保计划
4.3 设施计划	6.6 分包管理计划
4.4 方法，工具和技巧	6.7 项目交付计划
4.5 产品验收计划	7. 项目交付
4.6 项目组织	8. 支持过程计划
4.6.1 外部接口	8.1 项目监督和工作环境
4.6.2 内部接口	8.2 决策管理
4.6.3 授权和责任	8.3 风险管理
5. 项目计划	8.4 配置管理
5.1 项目启动	8.5 信息管理
5.1.1 评估计划	8.5.1 文档化
5.1.2 人员配置计划	8.5.2 沟通和宣传
5.1.3 资源收购计划	8.6 质量保证
5.1.4 项目职员训练计划	8.7 测量
	8.8 回顾和评审
	8.9 检验和确认

图 10.9: 一个来自于IEEE 16326-2009 ISO/IEC/IEEE的系统和软件工程——生命周期过程——计划管理标准的项目管理计划格式

（Sommerville 2010)。Fred Brooks在第1章中给出了解释：当你把更多的人添加到团队中的时候，能够并行工作的人也就越多，但是它也增加了每个人在沟通中花费的时间。这些团队的规模应该要合理地考虑花费到沟通中的时间比例。

在我们知道了团队的规模之后，在计划文档进程中，小组成员能够给予他们预期的角色。例如（Pressman 2010):

- 配置管理负责人
- 质量保证负责人
- 需求管理负责人
- 设计负责人
- 实现负责人

一个令人惊讶的发现就是团队的空间类型会影响项目管理。一个研究发现开放空间的团队搭配能够让效率增加一倍。（Teasley et al. 2000)其中的原因包括团队成员能够方便地和其他人沟通，以便于协调他们的工作和学习，并且他们能够把自己的工作成果发布在所有人都能够看到的公共空间中。另外一份关于关于开放空间的研究有如下结论：

> 成功的一个主要驱动是团队成员能够随叫随到，准备好一个自发性的会议，对一个问题的建议，以及教导或者学习一些新的东西等等。我们从早期的工作中知道了人们不在视野范围内时，效率会大幅下降，最严重的就是当他们相距超过30米的时候。
>
> —— Allen and Henn 2006

咖喱角 是一种来自于印度很流行的油炸小吃。

当一个团队需要邮件和在类似于wiki上面的文字进行沟通的时候，通常也需要一个周会来帮助协调项目。回想一下我们的目标是减少那些不必要的沟通，因此保持高效的开会是很重要的。下面是我们从网上找到的关于怎样有效开会的建议的摘要。我们使用SAMOSAS作为对我们的提醒，在开会的时候带来一份肯定会让我们的会议更加有效！

- **S**tart 准时开始和结束会议。
- **A**genda 在会议开始之前就指定好会议日程；如果没有日程，就取消会议。
- **M**inutes 每个时间都需要记录下来，这样随后每个人都能够回忆其结果；会议日程的第一项就是找到一个会议记录者。
- **O**ne 每次一个发言者；当另外一个人说话的时候不要打断。
- **S**end 提前分发资料，因为与说话者的语速相比，人们的阅读速度更快。
- **A**ction 在会议结束的时候写下来要做的事，这样作为会议的结果人们就知道他们应该做什么。
- **S**et 指定好下一次会议的日期和时间。

2、管理人员和冲突

关于如何管理人的书籍多达上千本，但是我们发现的最有用的是 **《一分钟管理》** 和 **《如何获得朋友和影响人们》** （Blanchard and Johnson 1982; Carnegie 1998)。我们喜欢第一本书的原因是它提供了简短快速的建议。明确你想达到的目标和应该怎样很好地完成它，但是把应该如何做留给团队成员以鼓励创新。当和个人见面检查进展的时候，一个好的开始是帮助他们建立自信，及给予他们时间去学习手边的任务。与此同时，你应该对他们进展的不好的地方保持坦诚，以及告诉他们为了修复问题应该做什么。我们喜欢第二本书的原因是它教授了说服的艺术，让人们去做你认为应该做，但是又不会让他们感到是被命令着去做的。这些技巧也能够让你在不能够施加命令的地方去说服人们，比如对你的客户和经理。

这两本书在一个团队内解决冲突的时候都非常有用。冲突并不一定就是坏事，因为有冲突总比让项目崩溃掉更好。英特尔公司把这种态度标为 **建设性的冲突** 。如果你强烈地意识到某个人正在提出错误的事，那么你有义务把这个问题提出来，即便他是你的老板。英特尔公司的文化就是把你的想法表达出来，即便你不同意房间里面最高级别的人的观点。

如果冲突仍旧存在，把这个留给计划文档进程中的项目经理，他会做出最后的决定。美国能够在20世纪60年代成功登月的一个原因就是，NASA的一个领导者Wernher von Braun有快速解决冲突的诀窍，他的观点就是随机地做出一个选择，通常越快越好，因为选择正确的概率大概是50%，这样项目就能够进展而不是花费时间来仔细考量所有的证据之后做出一个只是好一点的选择。

然而当一个决定做出之后，团队就需要拥抱这个选择并且开始前进。英特尔关于这项决议的座右铭就是 **不同意但是会负责** ：“我不同意，但是我会继续做贡献，即便我不同意。”

3、检验和度量

诸如 **设计评审** 和 **代码评审** 能够在任何东西都正式开始之前允许对系统的反馈。这个观念是一旦你有了一个设计最初的实现计划，你就做好了接受不仅仅是团队开发者的反馈。设计和代码评审遵循瀑布生命周期，每一个阶段在进入到下一个阶段之前被完成，或者至少是Sprial或RUP开发单个迭代阶段完成。

一个设计评审就是程序的作者展示其设计的会议，设计评审会议的目标是借鉴参会人员的经验来提升软件设计的质量。代码评审在设计被实现之后进行。这种同行之间的反馈也有利于帮助在组织内交换知识，并且能够提供有利于展示者职业发展的机会。

Shalloway认为正式的设计和代码评审因为在开发过程中发生的太晚，而不能显著地影响开发结果。（Shalloway 2002)他倡议使用称为“方法评审”的更早、更小会议的方法来代替现行的方法。这个观点就是让少数高级的开发者来帮助团队想出解决问题的方法。组内进行关于不同方法的头脑风暴来帮助找到一个很好的方法。

如果你计划做一个正常的设计评审，Shalloway建议你在选择一个方法和设计快接近完成时首先进行一个“小的设计评审”。它和之前一样涉及到同样的人，但是目标是为正式的检查做准备。

一个正式的检查本身应该以一个对用户需求的概述性描述开始。然后给出软件的架构，显示组件的API。突出在不同抽象层所使用的设计模式是很重要的（参见第11章）。人们期望解释你 **为什么** 做出这个决定，你是否考虑过其他可行方案。根据会议的时间和兴趣点，最后的阶段将是浏览实现方法的代码。在所有这些阶段，如果你有一个人们想听到的问题或疑问的具体清单，你将会在评审中收获更多。

代码评审的一个好处就是鼓励人们跳出团队来视察评论和代码。因为我们没有一个工具能够强制执行像第9章的建议，以确保评论会提升抽象层次，唯一的强制策略就是代码评审。

除了评审代码和评论外，视察能够给计划文档进程项目的任何部分的反馈：项目计划、调度、需求、测试计划等等。这些反馈能够帮助 **验证和确认** 整个项目，以确保其处于良好的过程中。这里甚至还有一份关于怎样书写核查和验证项目计划的IEEE标准，如图10.10所示。

1. 目的	5.4 硬件V&V 进程，活动区和任务
2. 参考文档	5.4.1硬件概念
3. 定义	5.4.2 硬件需求
4. V&V 概览	5.4.3 硬件设计
4.1 组织	5.4.4 硬件制造
4.2 主调度	5.4.5 硬件集成测试
4.3 完整性级别方案	5.4.6 硬件量化测试
4.4 资源总结	5.4.7 硬件验证测试
4.5 责任性	5.4.8 硬件转换
4.6 工具，技巧和方法	5.4.9 硬件操作
5. V&V 进程	5.4.10 硬件维护
5.1 通用体V&V 进程，活动区和任务	5.4.11 硬件处理
5.2 系统V&V 进程，活动区和任务	6. V&V 报告需求
5.2.1 保障支持	6.1 任务报告
5.2.2 供给规划	6.2 异常报告
5.2.3 项目计划	6.3 V&V 最终报告
5.2.4 配置管理	6.4 特殊的学习报告（可选）
5.2.5 涉众需求定义	6.5 其他报告（可选）
5.2.6 需求分析	7. V&V 管理需求
5.2.7 架构设计	7.1 异常决议和报告
5.2.8 实现	7.2 任务迭代策略
5.2.9 综合	7.3 偏差策略
5.2.10 转换	7.4 控制程序
5.2.11 操作	7.5 标准，实践和惯例
5.2.12 维护	8. V&V 测试文档化需求
5.2.13 处理	
5.3 软件V&V 进程，活动区和任务	
5.3.1 软件概念	
5.3.2 软件需求	
5.3.3 软件设计	
5.3.4 软件构建	
5.3.5 软件集成测试	
5.3.6 软件量化测试	
5.3.7 软件验证测试	
5.3.8 软件安装和检测（转换）	
5.3.9 软件操作	
5.3.10 软件维护	
5.3.11 软件处理	

图 10.10: 来自于IEEE1012-2012标准的关于系统和软件核查于验证计划的大纲

正如预测花费的算法模型（参见7.10节），一些研究人员提倡使用软件度量取代关于项目的质量和进展的视察和评审。这个观点就是收集组织内所有时间范围内的许多项目的度量，给新项目建立一个参考标准，然后查看这个项目和参考标准比起来怎样。这句引言描述来度量的观点：

> 没有度量，就难以去知晓一个项目是怎样被执行的，以及软件的质量层级。
>
> —— Braude and Berstein 2011

下面是可以被自动收集的度量：

- 代码规模，通过千行来计算代码（**KLOC**）或者功能点（参见7.10节）
- 工作量，通过花费在项目上的人月量来计算
- 项目里程碑计划和实现
- 通过的测试用例
- 缺陷发现率，通过每月测试中发现的缺陷来衡量。
- 缺陷修复率，以每月修复的缺陷来衡量。

只要规范化不同项目的比较结果的数字，其他度量可以从这中得到，如每人月的KLOC、KLOC的缺陷率等。

这个方法的问题就是，现在没有什么证据表明自动收集的这些指标和项目结果有什么关联。理想状态下，这些指标是有关联的，并且我们能够比偶尔和费时的评审获得更深入细致的理解。下面这个引言表达了对这些度量的观点不看好：

> 然而，我们离理想情况还有很长一段距离，并且没有信息显示在可遇见的未来内，自动品质评价能成为可能。
>
> ——Sommerville 2010

4 配置管理

配置管理包括四种形式，其中的三种我们在之前已经看到过了。第一种就是**版本管理**，我们在10.4节和10.5节见过。这种形式跟踪组件版本的改变。第二种，**系统构建**，和第一种很相近。像`make`这样的工具装配兼容版本的组件到一个目标系统的可执行程序中。第三种就是**发行管理**，我们将在第12章讲述。最后一种是**变更管理**，它来自于客户和其他涉众对于错误修复和提高功能性的变更请求（参见9.7节）。

正如你目前所期待的，IEEE有一个关于配置管理的标准，图10.11显示了表格的内容。

表格的内容
1. 概述
1.1 范围
1.2 目的
2. 定义，首字母缩写和缩写词
2.1 定义
2.2 首字母缩写和缩写词
3. 裁剪
4. 观众
5. 配置和管理进程
6. CM计划低层级进程
6.1 目的
6.2 活动区和任务
7. CM管理管理低层级进程
7.1 目的
7.2 活动区和任务
8. 配置鉴定低层级进程
8.1 目的
8.2 活动区和任务
9. 配置变更控制低层级进程
9.1 目的
9.2 活动区和任务
10. 配置状态审计低层级进程
10.1 目的
10.2 活动区和任务
11. CM配置审计低层级进程
11.1 目的
11.2 活动区和任务
12. 接口控制低层级进程
12.1 目的
12.2 活动区和任务
13. 供应商配置项控制底层过程
13.1 目的
13.2 活动区和任务
14. 发行管理低层级进程
14.1 目的
14.2 活动区和任务

图 10.11: 一个IEEE 828-2012关于系统和软件工程中的配置管理标准的内容表格

小结：
在计划文档进程中：

- 项目经理负责书写合同，组织团队，以及和用户及高层管理沟通。
- 项目经理书写项目计划和配置计划，同时也书写用来确保计划被跟进的核实和检查计划。
- 为了控制沟通上的时间，小组保持在3~10人的规模。他们能够通过层级关系组成向项目经理汇报的更大团队，每一个小组都有他们自己的负责人。
- 指导包括给他们清晰的目标，但是不强制他们，以鼓励性的评审开始，但是对缺点保持坦诚且帮助他们克服缺点。
- 虽然冲突需要解决，但解决这些冲突对项目找到最后的前进道路是非常有帮助的。
- 像设计评审和代码评审这样的核查能够让外面的人给当前的设计和未来的计划反馈，这能够让团队从其他人的经验中获益。这也是一个用来检验良好的规范是否被执行，以及计划和文档是否清晰的好方法。
- 配置管理是一个广泛的领域，它包括维护产品时候的变更管理，软件组件的版本控制，那些和工作程序有关联的组件的系统构建，以及把产品交付给用户的发行管理。

自我检查10.7.1. 比较计划文档过程和敏捷过程中的团队大小。
◇ 计划文档过程能够通过层级关系来组成一个更大的项目，但是每个小组的规模和敏捷中的“两个匹萨”规模是一样的。

自我检查10.7.2. 对或错：设计评审是一个为了提高软件产品质量的会议，它不仅仅利用了参会者的智慧，也增进了技术信息的交流，而这能够很好地教导组织内的初级成员，无论是展示者还是参会者。
◇ 对。

10.8 谬论和陷阱

陷阱：结对编程时总是观察更有经验的另一方。

如果结对中的一个人更具经验，一个倾向就是让更有经验的人做所有的开发工作，这样更少经验的人就相应地变成了永久的观察者。这种关系不是健康的，很有可能让缺乏经验的人离开。

陷阱：根据软件栈而不是根据功能划分工作。

把团队划分成前端专家、后端专家、客户联络员等现在变得更不常见，但是仍旧在发生。作者和其他一些人相信更好的结果来自于这样一种模式：即让团队里面的每一个成员了解一个选择的特征或者故事的 **所有** 方面——Cucumber场景、RSpec测试、视图、控制器行为、模型逻辑等等。特别是结合结对编程时，让团队里面的每一个开发者维护一个关于产品各个架构的“全局堆栈”视图。

陷阱：在合并或者改变分支之后意外地丢失了这些改变。

如果你做了一个pull或者merge, 或者你转向了一个不同的分支，一些在磁盘上的文件可能突然有不同的内容。如果任何这些文件已经被加载到了编辑器中，在编辑的版本将会 **失效** ，更坏的情况是，如果你没有保存这些文件，你或者会覆盖合并后的变更或者保存到一个你认为是在你的分支中，但是实际上却不在的文件。解决的方法非常简单：在你进行pull、merge或改变分支之 **前** ，确保你提交了当前所有的变更；在你进行pull、meger或者改变分支之 **后** ，重新加载任何可能受影响的文件到你的编辑器中——或者为了真正的安全，请在进行提交之前关闭编辑器。正如“Gitster”Scott Chacon翔实具体的博客[21]中指出的，也要对某些带有潜在毁坏性的Git命令保持警惕性，比如`git reset`。

陷阱：让代码库副本和原始（权威的）版本长期没有进行同步。

最好不让代码库和原始版本保持太长时间的分歧，否则合并（10.5节）将会非常的痛苦。你应该总是在你开始工作之前进行`git pull`，并且只要你本机的提交更改足够稳定，就进行`git push`以便对团队成员产生影响。如果你正工作在一个长期存在的，有和主版本不保持同步风险的功能分支时，参见`git rebase`文档让分支和主分支进行周期性的同步，但是只要在必要的时候才合并回主分支。

谬论：在主分支上保持简单的变更是很好的。

程序员是乐观的。当我们打算修改的代码时，我们总认为只会更改一行。然后发现需要更改5行；然后我们又会发现修改会影响其他的文件，而这些文件又不得不修改；然后我们会发现需要去添加或修改一些已经存在的和旧代码相关的测试，等等。因为这个原因，在开始一个新工作的时候， **总是** 创建一个功能分支。用Git创建分支几乎瞬时就能够完成，如果变更确实是非常小，为了避免在合并之后弄乱分支命名空间，你可以将分支删掉。

10.9 结束语：团队、合作、以及四十年的版本控制

开始90%的代码花费整个开发时间的10%。剩下10%的代码花费开发时间的90%。

——Tom Cargill，引自 **编程珠玑，1985**

伴随着两个匹萨大小的团队突然变成一种流行的协作单元，版本控制系统的历史反映了“团队中的团队”的面向分布式协作的运动。大约从1970——1985年，原始的Unix的 **源代码控制系统** （SCCS）和它的长期存在的继承者 **修订控制系统** （RCS)一次只允许有一个开发者，通过锁定某个特殊的文件——其他人只能够读，只有第一个开发者将文件送回，释放了锁之后，其他人才有修改文件的权利。SCCS和RCS也要求所有的开发者使用同一台计算机（通过分时），该计算机的文件系统存储了代码库。在一个有许多文件的系统中，这种锁机制很快就会成为一个瓶颈，因此在1986年， **并发版本控制系统** （CVS)通过自动合并允许对同一个文件进行同时编辑，以及允许主分支代码库可以在非开发者副本的机器上，促进了分布式开发。2001年开始的 **Subversion** ，对分支有更好的支持，允许开发者在一个项目不同版本上进行独立的工作，但是仍旧假定所有的开发者在一个特殊的代码树上进行工作，将会把他们的更改推送到同一个主代码库中。Git通过允许任何代码库的副本可以和其他代码库进行push和pull，完成了去中心化，并且通过比它的前辈更容易、更快地创建分支和合并，实现了完全去中心化的“团队中的团队”。分布式协作在

今天已经算是常态了：一个非大型的分布式团队，push和pull让许多两个匹萨规模的敏捷团队来进行独立地开发，Git的使用使得这些努力无处不在了。比之于可能采用计划文档模型的大型团队，这种新的两个匹萨规模的团队让组织更加容易。

尽管有这些改进，软件项目仍旧因为其延期交付和超出预算而名声不好。本章的技巧能够帮助一个敏捷团队避免这些陷阱。每一次迭代和涉众讨论能够让团队尽可能地利用资源，在预算范围内尽最大可能且及时地做出让用户满意的软件。Scrum团队组织与敏捷生命周期及开发SaaS的挑战很吻合。版本控制的原则让许多不同端的开发者同时进行，而又彼此不影响他人的工作，也允许有严格管理风格和系统的错误生命周期管理。

计划文档进程依靠项目经理进行时间和开销的估计、风险的评估以及项目的执行，这样能够在给定的时间和与功能对应的预算内交付产品。这种方法和平等的Scrum方法相比更加的专制，Scrum中敏捷教练和产品经理角色在敏捷团队中不断地由不同的成员充当。在计划文档生命周期中任何事情都是被书写规定的，包括项目管理计划，配置管理计划，以及怎样确认和验证产品是否走在正确道路上的计划。团队外面的开发者的视察对计划和代码给予反馈，并帮助评估项目进展。

一旦一个项目的每个生命周期都完成了，在你快速进入到下一个项目之前，花费时间去思考你从这个项目中学到了什么是很重要的。反省什么做得好，什么做得不好，以及你接下来将会采取什么不同的措施。只要你能够从中学到一些东西，犯错就不是一件丢人的事；重复地犯同样的错误才是丢人的。

10.10 深入学习

- 在 **《用Git进行版本控制》** 中（Loeliger 2009），你能够找到Git强大特征的详细描述，其采用了一个更加易于入门的方法，并且是一本免费Git社区书籍[22]，它也是一个有关Git非常有用的参考书。对于一个特定命令的具体帮助，使用`git help` **命令**，例如，`git help branch`；但是这些解释只是用来参考的，不是教程。

- 许多中等规模的项目不使用Pivotal Tracker，或者那些错误管理需求超出了Tracker提供的项目也不使用Tracker，而是依靠每一个GitHub代码库内建的问题特征。问题系统允许每个团队为不同的错误类型创建合适的标签，并且创建他们自己的“错误生命周期”进度和设置优先级。广泛分布的大型项目通常会考虑更高级（也更复杂）的错误跟踪系统，例如开源的Bugzilla[23]。

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

T. J. Allen and G. Henn. **The Organization and Architecture of Innovation: Managing the Flow of Technology** . Butterworth-heinemann, 2006.

A. Begel and N. Nagappan. Pair programming: What's in it for me? In **Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement** , pages 120–128, Kaiserslautern, Germany, October 2008.

K. H. Blanchard and S. Johnson. **The One Minute Manager** . William Morrow, Cambridge, MA, 1982.

P. Bodík, A. Fox, M. I. Jordan, D. Patterson, A. Banerjee, R. Jagannathan, T. Su, S. Tenginakai, B. Turner, and J. Ingalls. Advanced tools for operators at Amazon.com. In **First Workshop on Hot Topics in Autonomic Computing (HotAC'06)** , Dublin, Ireland, June 2006.

E. Braude and M. Berstein. **Software Engineering:Modern Approaches, Second Edition** . John Wiley and Sons, 2011. ISBN 9780471692089.

D. Carnegie. **How to Win Friends and Influence People** . Pocket, 1998.

A. Cockburn and L. Williams. The costs and benefits of pair programming. **Extreme Programming Examined** , pages 223–248, 2001.

J. Hannay, T. Dyba, E. Arisholm, and D. Sjoberg. The effectiveness of pair programming: A meta-analysis. **Information and Software Technology** , 51(7):1110–1122, July 2009.

D. Holland. **Red Zone Management** . WinHope Press, 2004. ISBN 0967140188. URL `http://www.amazon.com/Red-Zone-Management-Dutch-Holland/dp/0967140188`.

J. Loeliger. **Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development** . O'Reilly Media, 2009. ISBN 0596520123. URL `http://www.amazon.com/Version-Control-Git-Collaborative-Development/dp/0596520123`.

J. Moore. ipad 2 as a remote presence device? **Pivotal Blabs** , 2011. URL `http://pivotallabs.com/blabs/categories/pair-programming`.

R. Pressman. **Software Engineering: A Practitioner's Approach, Seventh Edition** . McGraw Hill, 2010. ISBN 0073375977.

A. Shalloway. **Agile Design and Code Reviews** . 2002. URL `http://www.netobjectives.com/download/designreviews.pdf`.

I. Sommerville. **Software Engineering, Ninth Edition** . Addison-Wesley, 2010. ISBN 0137035152.

S. Teasley, L. Covi, M. S.Krishnan, and J. S. Olson. How does radical collocation help a team succeed? In **Proceedings of the 2000 ACM conference on Computer supported cooperative work** , pages 339–346, Philadelphia, Pennsylvania, December 2000.

注释

[1] `http://www.opensourcerails.com/`
[2] `http://github.com/ucberkeley/researchmatch`
[3] `http://github.com/vinsonchuong/meetinglibs`
[4] `http://codeclimate.com`
[5] `https://github.com/bbatsov/rails-style-guide`
[6] `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`
[7] `http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml`
[8] `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`
[9] `https://github.com/bbatsov/rails-style-guide`
[10] `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

[11]http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml
[12]http://dilbert.com/strips/comic/2003-01-09/
[13]http://dilbert.com/strips/comic/2003-01-11/
[14]http://help.github.com/fork-a-repo/
[15]http://help.github.com/send-pull-requests/
[16]http://code.flickr.com/blog/2009/12/02/flipping-out/
[17]http://nvie.com/posts/a-successful-git-branching-model/
[18]http://pastebin.com/8JkAQDx0
[19]http://github.com/
[20]http://fakeweb.rubyforge.org
[21]http://progit.org/2011/07/11/reset.html
[22]http://book.git-scm.com
[23]http://mozilla.org

10.11 建议的项目

练习10.1. 从本书中挑选几个练习，给它们评分，然后测试你完成它们的速度。

练习10.2. 考虑一个你经常访问的网站，或者一个你经常使用的网络应用程序；列出一些用户故事，它能够指导你从头创建一个类似的应用程序。

练习10.3. （讨论）考虑你最近的一个以小组形式工作的项目。你和你的小组使用了多少本章讨论过的观点和建议？在那些你使用的方法中，你觉得哪一种是最有用的？在你未使用的方法中，哪种方法你认为会是最重要的？

练习10.4. （讨论）在第1章中一个建议的项目，就是列出10个最重要的应用。给出了这样一个清单，哪个最有可能使用一个Scrum组织的两个匹萨规模的团队进行开发和维护？哪一种将不会？列出你的选择的理由。

练习10.5. （讨论）你认为Pivotal为什么要将Epics添加到Tracker中？他们使用这个新功能解决什么问题呢？

练习10.6. （讨论）找到一个附近的编程同事来尝试几天的结对编程。上一章几个建议的项目上结对编程是一个很好的选择。找到一个能够坐在一起的地方有多难？你发现它强制让你们更加专注来写出更高质量的代码，部分因为减少了注意力的分歧，或者它似乎效率更低，因为本质上只有一个人在进行工作？

练习10.7. 和一个编程伙伴，做出一个在你能力范围内的网站或者应用程序。理想情况下，它应该不会比RottenPotatoes更复杂！使用本章描述的方法和工具——结对编程、速度（velocity）、版本控制来工作并且完成项目。哪一个工具或方法是最有用的？

练习10.8. （讨论）下次你去参加一个会议，记录有多少SAMOSA原则被违背了。如果有一些，建议你把试图跟随SAMOSA作为一个实验。你注意到了这两个会议有什么不同了吗？SAMOSA是对会议有帮助，还是对会议有弊？

练习10.9. 作为团队活动的一部分，对一个中等规模的代码片段进行审视。注意：边上的图标表明项目来自于ACM/IEEE 2013软件工程标准（ACM 2013）。

练习10.10. （讨论）在Linus Torvalds创建Git五年前，由CollabNet创建的Subversion（svn）是一个非常流行的版本控制系统。Torvalds用Git试图去解决svn的什么问题？Torvalds是如何取得成功的？你认为svn和git相比的优缺点是什么？

练习10.11. 描述中心化和分布式软件配置管理的不同。

练习10.12. 识别配置项和使用一个源码管理工具，诸如基于小团队的GitHub。

练习10.13. 给一个会议创建一个日程并遵循它。

练习10.14. 挑选和使用在一个小型软件项目中定义的编码标准。

练习10.15. 确定和判别用计划文档进度的软件开发团队中必要的角色。

练习10.16. 列出团队冲突的原因，危害和潜在的好处。

练习10.17. 在一个团队设置中应用一个冲突解决策略。

练习10.18. 遵循一个计划文档生命周期，为一个软件项目准备一个项目计划，包括对规模和努力的估计、一个调度计划、资源分配、配置管理、变更管理以及项目风险鉴别和管理。

11 SaaS类的设计模式

William Kahan
(1933——) 因其在数值分析领域的基础贡献，获得了1989年的图灵奖。Kahan把他的一生都致力于“让全世界更加安全地进行数值计算”。

当你能够正确地思考接下来将会怎样，且没有许多外来或让人困惑的观点强加于你的时候，事情就真正简单了。思考一下爱因斯坦的格言，“一切都应该足够简单，但不能够更简单。”

——“和William Kahan的一次谈话”，**Dr. Dobbs期刊**，1997

11.1 模式、反模式及SOLID类架构 336
11.2 正好够用的UML 340
11.3 单一责任原则 342
11.4 开闭原则 345
11.5 里氏代换原则 348
11.6 依赖注入原则 350
11.7 迪米特法则 354
11.8 计划文档的视角 359
11.9 谬论和陷阱 360
11.10结束语：框架包含了设计模式 361
11.11深入学习 361
11.12建议的项目 363

重要概念

本章最大的概念就是 **设计模式** 能够提高类的质量。一个设计模式就是通过分离可变与不可变的东西来得到问题的已被证明的解决之道。

由首字母缩写组成的五个面向对象的SOLID设计原则描述了类之间的交互设计。一个 **反模式** 表明了糟糕的类设计，其可以通过 **设计味道** 来鉴别。因此，使用 **设计味道** 来检测对 **SOLID原则** 这种好的设计模式的违背，类似于使用 **代码味道** 来检测对 **SOFA原则** 这种好的方法设计的违背（参见9.7节）。

SOLID的五个缩写字母分别代表：

(1) **单一功能原则（Single Responsibility Principle）**：一个类应该有且只有一个责任；也就是，只有一个改变的原因。 **方法缺乏内聚性（Lack of Cohesion Of Methods）** 这一度量表示了一个庞大的类这种反模式。

(2) **开闭原则（Open/Closed Principle）**：一个类应该对扩展保持开放性，但是对修改保持封闭性。 **Case语句** 设计味道意味着对该原则的违背。

(3) **里氏代换原则（Liskov Substitution Principle）**：一个为类型T设计的方法应该对T的子类起作用。也就是说，类型T的所有子类都应该保持T的"功能"。 **拒绝的遗产(refused bequest)** 设计味道指示了对该原则的违背。

(4) **依赖注入原则（Dependency Injection Principle）**：如果两个类彼此依赖，但是其实现可能会改变，那么依赖于一个可以"注入"到它们中的一个单独抽象层会更好。

(5) **迪米特法则（Demeter Principle）**：一个方法能够调用自己类内的其他方法，以及该类实例变量的类中的方法；其他的都是被禁止的。违背此原则的一个设计味道就是 **不合适的亲密** 。

对于敏捷开发， **重构** 是提高类设计与方法设计的前进动力；在一些情况中的重构可能会允许你使用一个合适的 **设计模式** 。作为对比，对于计划文档模型的生命周期：

- 早期的设计阶段让软件最初架构和类的设计选择更加简单。
- 条款被划分到各个问题及其子问题中，开发者试图在这些问题中使用模式来解决它们。
- 正如设计指导代码编写， **设计评审** 能够提供早期的反馈。
- 一个担心就是一旦编码开始之后，设计是否需要更改。

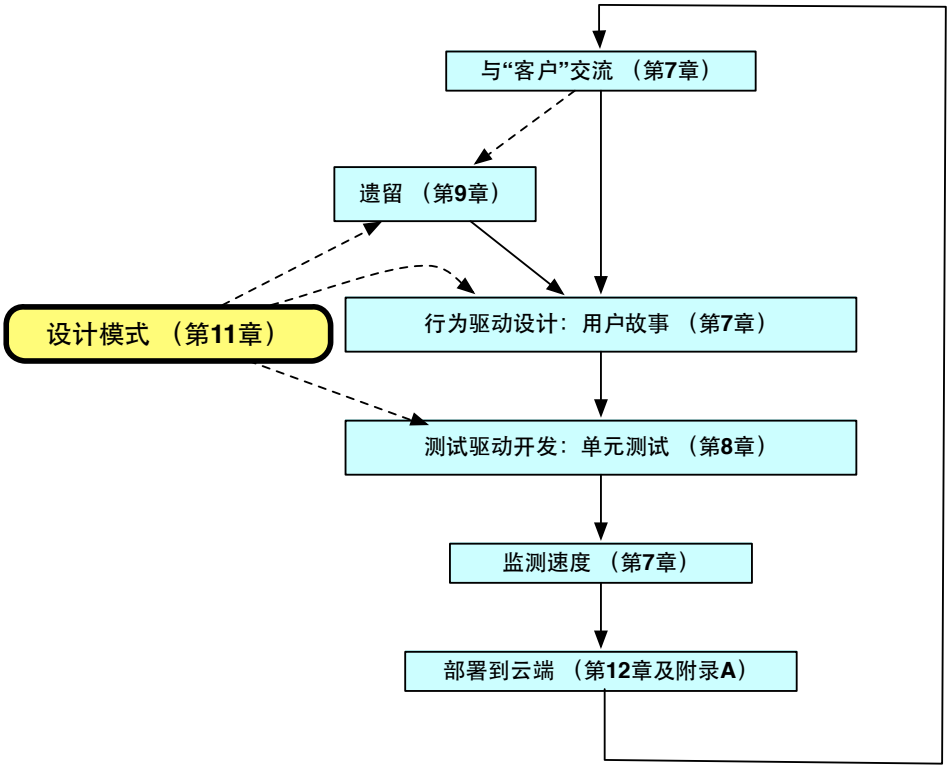


图 11.1: 本书这章中的敏捷软件生命周期及其关系。本章涉及设计模式，这些模式影响了BDD和TDD对新应用程序 及遗留代码的使用

11.1 模式、反模式及SOLID类架构

第2章介绍了设计模式：一个可重用的结构、行为、策略或技巧，它通过 **从相同中分离出不同** 来让一个可行的解决方案应用到一系列类似问题中。在本书中，模式帮助我们在实现目标中扮演了一个主要的角色：产生的代码不仅仅是正确（TDD）和满足用户需求（BDD），也是精确、可读、无重复和漂亮的。图11.1强调了本章所讨论的设计模式在敏捷开发中的角色。

正如我们已经见过客户-服务器架构模式和MVC结构模式，本章讨论类和类架构的设计模式。如图11.2所示，我们将会使用类似于第9章中的方法。不是简单地列出一系列的设计模式，我们将会通过一些关于是什么让一个类架构变好或变坏的指导方针，可能存在的问题的气味和症状，以及怎样通过显式重构来修复这种问题——通过在类内和类间移动代码来消除问题，来具体化它们的使用。在一些情况下，我们能通过重构让代码和一个已存在和被证明了的设计模式相匹配。在另一些情况中，重构并不一定导致类架构中有大的结构改变。

对于方法层级的重构，使用设计模式的最好方法是一边做，一边学，设计模式的数量超出了我们在本书这章覆盖的范围。实际上，一些全书都在讲述设计模式的书，包括最初的 **《设计模式：面向对象软件的可重用元素》** （Gamma et al. 1994），其作者以“四人组（Gang of Four）”或者Gof出名，书中的条目被称为“ **四人**

第9章	第11章
警告一个类内的方法的问题的代码味道	警告类间问题的设计味道
许多有关代码味道和重构的条目；我们使用Fowler的作为定义	许多有关设计味道和设计模式的条目；我们使用四人组的设计模式的Ruby版本作为定义
ABC，循环复杂度度量用量化的警告来衡量代码补充代码味道	LCOM（方法缺乏内聚性）度量用量化的警告来补充设计味道
通过提取方法和在一个类内移动代码来进行重构	通过提取类和在类间移动代码来进行重构
好的方法的SOFA原则（简短（**S**hort）、只做一件事（do **O**ne thing）、少量参数（**F**ew arguments）、单一抽象层（single **A**bstraction level））	好的类架构的SOLID原则（单一责任原则（**S**ingle responsibility）、开闭原则（**O**pen/Closed）、里氏代换原则（**L**iskov substitution）、依赖注入原则（dependency **I**njection）、迪米特法则（**D**emeter））
一些不适用于Ruby的代码味道	一些不适用于Ruby或者SaaS的设计味道

图 11.2: 在第9章中介绍的单独类和方法的警告症状和补救措施和本章介绍的类之间的关系的对比。对于文本中解释的原因，在大多数书在SOLID中用I表示接口隔离（一个在Ruby中不会引起的味道），以及D表示注入依赖性。我们使用I指代注入独立性（Injecting dependencies)和D指代迪米特法则（Demeter principle），这在Ruby中很常见

组设计模式”。如图11.3所示，四人组的23条设计模式被划分为创建型、结构型及行为型。正如Fowler的关于重构的原始著作，四人组的设计模式引发了其他书用某种特定的编程语言作为例子来教导设计模式，包括Ruby(Olsen 2007)。

因为四人组的设计模式最初是关于静态语言的，一些人认为在Ruby中不会出现这些问题。例如，Ruby中就很少出现因消除类型签名导致重新编译的情况，因为其不需要编译且不使用类型来强制签名。

四人组的作者阐述了面向对象设计中广为传播的两条原则，这两条原则对大多数模式都有启发：

- 相比于继承，更倾向于组合和委托。
- 针对接口而非实现进行编程。

随着对一些具体的设计模式的了解，我们将会理解这些包罗万象的词语意思。

一个理想的世界中，所有的程序员都会有品味地使用设计模式，如第9章中所建议的持续重构其代码，并且所有的代码都将会非常优美。很显然事情并不总是这样的。一个 **反模式** 的代码片段看起来使用了一个很著名的设计模式，但实际上不是这样的——一般因为最初的代码（好的）为了满足新的需求而进行了更新，可并没有同时进行重构。 **设计味道** 和我们在第9章中见到的代码味道一样，是你的代码朝着反设计模式方向发展的警告信号。和代码味道通常使用在一个类内的方法不一样，设计味道是在类间使用，以及负责它们之间怎样划分。因此，重构方法是在 **类内** 移动代码，重构设计是在 **类间** 移动代码，创造新的类或者模块（也许通过从已存在的类中提取共性），或者移去那些不再发挥其作用的类。

“Uncle Bob” Martin是一个美国软件工程师，自1970年就作为一个顾问[1]，是敏捷/极限的创立者，也是 **软件工艺（Software Craftsmanship）** 运动的领导成员，它鼓励程序员把他们自己看成是在学徒制模式下学习专门技巧的创造性专业人士。

类似于第9章中的SOFA，缩写词SOLID（由Robert C. Martin所创立）代表了整洁代码应该遵循的五个设计原则。和第9章一样，设计味道和定量指标能够在我们违背SOLID原则的时候进行提示；修复通常是一个通过把代码和设计模式联系起来以解决问题的重构。

图11.4 显示了缩写词SOLID，及告诉了我们什么是好的类组合。在讨论所选择的设计模式中，我们将会看到对每一个原则的违背，和怎样通过重构坏的代码能够修复这些违背（在某些情况下，是为了使用某个设计模式）。一般而言，SOLID原则能够解决类架构中各种影响效率的问题：

(1) 粘度（Viscosity）：使用一个快速的hack可以非常容易地修复一个问题，即便你认为这不是正确的事情。

创建性模式（Creational）
抽象工厂模式（Abstract Factory），工厂方法模式：给一系列相关或独立的对象创建一个接口，而不指定它们具体的类 单例模式（Singleton）：确保一个类有且仅有一个实例，并且提供一个全局的访问点。 原型模式（Prototype）：使用一个原型性的实例来指定要创建的对象的类型，并且通过复制这个原型来创建新的对象。正如我们将在第6章见的，基于原型的继承是JavaScript语言的一部分。 生成器模式（Builder）：把一个复杂对象的构建从其展示中分开，使得同样的构建过程可以创建不同的展示。
结构性模式（Structural Patterns）
适配器模式（Adapter），代理器模式（Proxy），外观模式（Façade），桥接器模式（Bridge）：把一个类的编程接口装换到另一个客户期望的接口（有时更简单），或者把一个抽象层的接口从其实现中去耦合，为了依赖注入或者性能。 **装饰器模式（Decorator）**：动态地把额外的责任附加到一个对象，保持同样的接口。以便"更倾向于组合或者使得委托优于继承。" 组成模式（Composite）：提供在一个单独的对象和一些列对象上都起作用的操作。 享元模式（Flyweight）：使用共享来高效地支持大量的同类型对象。
行为模式（Behavioral patterns）
模板方法模式（Template Method），策略模式（Strategy）：给同一个任务统一地封装多个不同的策略。 **观察者模式（Observer）**：当一个对象发生改变时，一个或多个实体需要被通知。 **迭代器模式（Iterator），访问者模式（Visitor）**：把一个数据结构的遍历操作从对数据结构的每一个元素访问的操作分离开来。 **空对象模式（Null Object）**：（没有出现在四人组作者的条目中）提供一个定义了能够安全调用的中立方法的对象，来取代条件保护方法的调用。 状态模式（State）：包装这样一个对象，它的行为依赖于少量的内部状态。 职责链模式（Chain of Responsibility）：为避免请求的发送者与其接受者耦合在一起，给予多个对象都有处理这个请求的机会，并沿着这条链传递该请求，直到有一个对象处理它为止。 中介模式（Mediator）：定义一个对象来封装一组对象交互的方式，中介者使各对象不需要显式地相互引用，从而使其耦合松散。 解释器模式（Interpreter）：给一个语言定义一个展示语法，并定义一个解释器来执行展示语法。 命令者模式（Command）：封装一个操作请求为一个对象，这样使得你用不同的请求、队列或者日志请求以及支持来参数化用户。

图 11.3: 23条 四人组作者的设计模式 原则分为三个类型，粗体显示了我们会在阐明或修复违背SOLID原则的情况下遇到的设计模式子集，而且我们把相近的模式，例如抽象工厂和工厂方法，合并到一个实体里面。无论何时介绍一个设计模式，我们将会解释这个模式的目的，用统一建模语言（在下一节介绍）表示重构前后的类架构，给出一个模式是如何在Rails或者Ruby gem中广泛应用的例子

(2) 不动性（Immobility）：不重复会非常难，因为你想重用的功能很难以提取的方式连接到应用程序。

(3) 不必要的重复（Needless repetition）：以一种连续的不动性，应用程序中或许可在多个地方重复相似的功能，因此功能性中的一点小变化都会导致代码和测试中有许多小改变，这个过程有时被称为 **霰弹式修改（shotgun surgery）** 。

(4) 不必要的复杂（Needless complexity）：应用程序的设计通常反映了许多不必要的添加。

原则	意思	警告味道	重构修复
单一责任原则	一个类有且仅有一个修改的理由	庞大的类，低的LCOM得分，数据泥团	提取类，移动方法
开闭原则	类应该对扩展保持开发，而对修改保持关闭	条件复杂度，基于**case**的分发器	使用策略或者模板方法，可能结合其他抽象工厂模式；使用装饰器来避免子类数量爆炸
里氏代换原则	替换一个类的子类应该保持正确的程序行为	拒绝继承：子类破坏性地覆盖一个继承的方法	用委托来替代继承
依赖注入原则	那些它的实现在运行时可能会改变的协同类应该依赖一个中间状态的“注入的”依赖	单元测试需要 **特别设计的** 桩来创建嫁接；构造器硬编码到另一个构造器的调用，而不是允许运行时决定使用哪个其他的类	注入一个依赖到一个共享的接口，以分离类；根据需要使用使用适配器模式，外观模式，或者代理器模式来使跨变量的接口统一
迪米特法则	仅和你的朋友交谈，把你的朋友的朋友当作陌生人	不合适的亲密，特性依恋，连环残骸模仿（参见8.11节）	委托行为和调用委托方法

图 11.4: SOLID设计原则和一些表明你的代码可能违背了其中某条的设计味道。我们和标准的SOLID原则有一点不同：我们用I代表注入依赖性，D代表迪米特法则，而在大多数书中用I代表接口隔离（我们在Ruby中不使用），D代表注入依赖(Ddependencies)

对于重构和遗留代码，寻找设计风格和用正确的设计模式重构它们，是一个通过实践学习的技巧。因此，不是列出设计味道、重构和设计模式的清单，而是关注于对SOLID原则的讨论，并且给出一些全面鉴定设计味道和解决这些味道的替代设计的典型例子。当处理自己程序的时候，使用11.11节中列出的更详细的资源。

模式、反模式及SOLID小结：

- 好的代码应该能很优雅地适应相应的改变。设计模式被证明是通过解决常见问题实现这个目标的正确途径。它们通过提供一个分离在进化中可能变化或者保持相同的整洁的方法，及适应这些变化的方法来实现。
- 和独立的方法一样，重构是一个提高类架构的过程，其通过在类间和类内移动代码来让代码更易维护和进化。在一些情况下，这些重构使用了“四人组”（GoF）的23条设计模式中的某个。
- 和独立方法一样，设计味道和度量能够作为一个早期的 **反模式** 警告——一小段遵循设计模式的代码片段，它可以变得有更好的架构。

■ **详细阐述：其他类型的模式**

自本书开始我们一直在强调，正确使用模式遍及好的软件工程项目。为了完成类级别设计模式，其他人已经发展了一系列企业应用的架构模式[2]（我们已经在第2章见过）、并行的编程模式[3]、计算模式（为了支持特殊的算法家族，例如图算法、线性代数、环、网格等）、**并发模式** 和用户接口模式[4]。

Grady Booch（1955——），因为其在软件工程和协作开发环境中的工作而广为人知，他和Ivar Jacobson、James Rumbaugh发明了UML。

自我检查11.1.1. 对或错：一个测量软件质量的方法就是其使用的设计模式数目。

◇ 错误。虽然设计模式为一些常见问题提供了已被证明的解决方法，不属于这些问题的代码可能不需要那些模式，但这也不会让代码变坏。四人组的作者特别警告通过设计模式的使用来度量代码质量。

11.2 正好够用的UML

统一建模语言 或者UML不是一个文本语言，而是一系列图像标注技巧，用来“指定、可视化、修改、构建及文档化一个开发中的面向对象密集型软件系统的部件[5]。”通过统一之前不同的建模型言标准和图11.5列出的图解类型，UML从1995年发展到了今天。

虽然本书更多地关注轻量级敏捷建模——事实上，基于UML的建模被指责太“浮肿”和重量级——即便一些UML关系图被广泛应用在敏捷建模中。图11.6显示了一个UML **类图**，它描绘了应用程序中每一个真实的类，它最重要的类，实例变量和方法，及与其他类的关系，例如有许多（has-many）和属于（belongs-to）关联。线端点连接的两个相关联的类被标有了可参加到关联的最小和最大数目实例，被称为关联 **多重性**，使用符号***** 表示“无限”。例如，一个多重性**1..*** 表示“一个或更多”, **0..*** 表示“零个或更多”，及**1** 表示“仅仅一个”。UML区分两种类型的“拥有”（有一个或者有许多）关联。在一个 **聚合（aggregation）** 中，被拥有者在拥有者销毁之后继续存在。例如，**课程有许多学生** 是一个聚合，因为当课程销毁的时候，学生不会被销毁！在一个 **组合（composition）** 中，被拥有者通常会随拥有者销毁。例如 **电影有许多评论** 是一个组合，因为删除一个电影会导致其所有的评论都被删除。

即便在不使用UML的其他软件工程师中，类图也非常流行。基于上述这一点点的UML的介绍，后面使用SOLID原则和设计模式改进代码时，我们能够使用类图来阐明类图的“在修改前和修改后的区别”。

结构图	
类	通过显示系统的类，它们的属性，以及类间的关系来描述一个系统的结构。
组件	描述一个软件系统是怎样被划分进组件，以及显示这些组件之间的依赖。
组合结构	描述一个类的内部结构，以及这个结构可能的协作。
部署	描述系统实现所使用的硬件，及运行环境和部署在硬件上的物品。
对象	显示一个示例模型系统在一个指定时间的全部或部分视图结构。
包	通过显示相互之间的依赖性，描述了一个系统是如何被划分成逻辑组的。
简况	通过哪些特定的对象类型能够在一个特定的应用程序中能继承使用的方法，描述了可重用的特定领域的模板对象。
交互图	
交流	显示对象之间的交互或。
交互概览	提供一个每个节点代表的交流图表的概览。
序列	在一系列信息中，每个对象是怎样和其他交互的。也只是了每个对象和关联信息的生命周期。
时间图表	一种特殊类型的交互图表，主要关注于时间约束。
活动图	
活动	描述一个系统中的业务和组件中的一步步的操作流。一个活动图显示了整个控制流。
状态机	描述系统的状态和状态转移。
使用案例	描述了一个系统的行为体提供的功能性，它们作为用例的目标，以及这些用例的依赖性。

图 11.5: UML 2.2定义的14种图解类型用来描述一个软件系统。这些描述基于优秀的Wikipedia上对UML的总结[7]，在上面还给出了每种图解类型的一个例子。使用案例图解和敏捷用户故事类型，但是其缺少具体的层次允许像Cucumber这样的工具来建立用户故事和整合/接受测试之间的桥梁

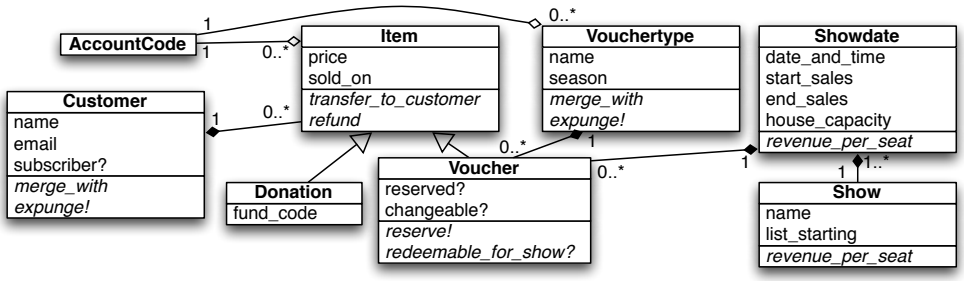


图 11.6: 这个UML 类图 显示了一个剧院票房应用程序中的一部分类，这和图9.4及图9.5是同一个应用。每个方框代表一个类及它最重要的方法和属性（或者说是类的责任）。继承通过箭头来表示。有关联的类通过线相连，其端点被标记为用单独或者范围表示的多重性标记（multiplicity），可选地加上一个菱形——空心的菱形表示聚合（aggregation），实心的菱形表示组合（composition），没有菱形的表示其他形式的关联

统一建模语言(UML)的小结：

- UML包括一个家族的图表类型，用来阐明软件设计和实现的不同方面。
- 即便是在不使用其他UML特性的工程师中，UML类图也被广泛使用了。它们显示了一个类名，该类最重要的公有和私有方法、属性及和其他类的关系。

■ **详细阐述：何时使用UML?**

尽管是一个重量级的工具，对于被划分成各个子系统，每个子系统由地域上被广泛分布的团队开发的非常大型的应用，UML非常有用。也因为UML标记是与语言无关的，它对于协调国际团队也是非常有帮助的。因为UML的成熟，许多工具支持其使用；挑战就是使图表和代码及设计"保持同步"，这是为什么大多数这种工具试着在这两个方向进行尝试，从UML同步代码框架，和从代码提取UML图表。对学习UML有帮助的一个这样工具就是UMPLE[8]，一个渥太华大学开发的拥有表达类关系的领域特定语言。Try Umple[9]网站能够从UMPLE代码生成UML类图，你自己画的图表中生成UMPLE代码，或者根据UMPLE代码或UML图表来生成不同编程语言的可执行代码。它是一个探索UML和类图的好工具，但是我们不推荐使用其生成的Ruby代码，因为生成的代码不是非重复的（non-DRY）且有点不符合Ruby的语言习惯。

自我检查11.2.1. 在一个描述"大学有许多部门"关系的UML类图中，关联的每一边应该允许怎样的多重性（multiplicities）？
◇ 大学这端重数为**1**, 因为一个部门必须仅属于一个大学。部门这段重数为**1..***, 因为一个大学可以有一个或多个部门。

自我检查11.2.2. "大学有许多部门"，这一关系应该被建模为一个聚合还是组合?
◇ 它应该是一个组合，因为大学关闭之后，部门不应该存在。

11.3 单一责任原则

SOLID中的 **单一责任原则** （SRP）强调一个类应该有且仅有一个责任——也是就说，只有一个修改的原因。例如，在5.2节中，当我们给RottenPotatoes添加单一的登录需求，我们创建了一个新的**SessionsController**来处理登录交互。一个看起来可行的替代方法是增大**MoviegoersController**，因为登录也是一个和moviegoers有关联的行动。事实上，在第5章描述的单一登录方法出现之前，该方法是早期版本的Rails中推荐的基于密码的验证方法。但是只要我们想改变验证策略，即便Moviegoer本质上不取决于用户怎样登录，这样一个方案将要改变**Moviegoer**模型和控制器，在MVC中，每一个控制器应该专门处理一个资源；验证了的用户会话是一个和用户本身不同的资源，它应该有自己的RESTful行为和模型方法。一般而言，如果你不能用25个以内的单词描述一个类的责任，它很可能不止一个责任，新的责任应该被划分到它们各自的类中。

在静态类型的编译语言中，违背SRP的代价是明显的：一个类的任何改变都需要重新编译，并且也可能触发那些依赖于它的类的重新编译和链接。因为我们在解释性的动态语言中不会付出这些代价，这会非常容易让类变的太大和违背SRP。一个警告就是缺乏 **内聚性** 。内举行表示了单一的逻辑实体中相关元素的关联程度。如果两个方法访问同一个实例或者类变量的子集，或者一个方法调用另一个方法，这两

个方法就是相关联的。缩写词 **LCOM** 代表 **方法缺乏内聚性（Lack of Cohesion Of Methods）**，用来测量一个类的内聚性：特别地，如果一个类包含多个"簇"，同一个簇中方法是相关联的，但是不同簇中的方法没有很强的关联性，你应该警惕这个类的方法是否缺乏内聚性。图11.7显示了LCOM中两个最常被使用的变体。

LCOM变体	得分	解释
修订过的Henderson-Sellers LCOM	0（最好的）～1（最坏的）	0意味着所有的实例方法访问了所有的实例变量。1意味着任何给定的实例变量都只仅被一个实例方法使用，也就是说，实例方法之间是公平独立地。
LCOM-4	0（最好的）～1（最坏的）	在用图表示中，方法节点通过一条边相连的组件是认为互连的，通过图中互连的组件来估计你的类中责任的数目。得分n大于1表示n-1个责任可以被提取到它们自己的类中。

在9.6节中，在成功地重构**convert**之后，`reek`报告在类**TimeSetter**中的"低内聚性"，因为我们使用了类变量而不是实例变量来维护实际的实例状态。正如本节所描述的。

图 11.7: "推荐的"缺乏内聚性的方法（LCOM）得分严重取决于使用何种LCOM变体。表格显示了最广泛使用的两种变体

数据泥团（Data Clumps） 设计味道是对一个好的类在逐渐进化成"多责任"反模式的一个警告信号。一个数据泥团是这样的变量或值，它们总是一起作为一个方法的传递参数，或一起作为某个方法结果的返回值。这种"一起传递（traveling together）"是值可能传递了一个信号，即它们可能需要自己的类。另外一个信号就是作为一个"简单"的数据值需要新的行为。例如，假设一个**Moviegoer**有电话号码**phone_number**和邮政编码**zipcode**两个属性，且你需要添加邮政编码准确性和规范手机号码格式的能力。如果把这些方法添加到**Moviegoer**中，它们将会减少内聚性，因为它们形成了一个仅处理特殊实例变量的"小集团"方法。正确的做法就是使用**提取类** 的重构方式将这些方法放到一个新**Address**类中，如图11.8所示。

单一责任原则的小结：

- 一个类应该有且仅有一个改变的理由，也就是单一责任。
- 一个糟糕的LCOM（Lack of Cohesion Of Methods）得分和数据泥团设计味道都是对可能违背SRP的警告。提取类的重构方式能帮助移除和打包额外的责任到一个单独的类中。

和SRP相关的是 **接口隔离原则**（ISP，以及SOLID中最初版本中的**I**），它强调了如果一个类的API被许多类型差异很大的客户所使用，这个API大概应该要被隔离到真对每种类型的客户有用的子集中。例如，**Movie**类可能会提供movie元数据（MPAA评分，发行日期等等）和一个搜索TMDb的接口，但是一个使用这两种服务中的客户不太可能关心另一种服务。ISP解决的问题是，在编译性语言中，对一个接口的改变需要重新编译这个类，从而引发所有使用这一接口的类的重新编译和链接。虽然通过文档来隔离不同功能的接口是一个好的风格，但是ISP很少在Ruby中引起这些重编译问题，是因为Ruby没有编译的类，因此我们也不进一步讨论了。

自我检查11.3.1. 画出在图11.8重构前后的类架构UML类图。

◇ 图11.9 显示了UML图。

代码文件：ch_designpatterns/code/srp_example.rb

```
class Moviegoer
  attr_accessor :name, :street, :phone_number, :zipcode
  validates :phone_number, # ...
  validates :zipcode, # ...
  def format_phone_number ; ... ; end
  def verify_zipcode ; ... ; end
  def format_address(street, phone_number, zipcode) # data clump
    # do formatting, calling format_phone_number and verify_zipcode
  end
end
# After applying Extract Class:
class Moviegoer
  attr_accessor :name
  has_one :address
end
class Address
  belongs_to :moviegoer
  attr_accessor :phone_number, :zipcode
  validates :phone_number, # ...
  validates :zipcode, # ...
  def format_address ; ... ; end # no arguments - operates on 'self'
  private  # no need to expose these now:
  def format_phone_number ; ... ; end
  def verify_zipcode ; ... ; end
end
```

图 11.8: 为了提取类，我们把共享一个责任的方法组区别于类中的其他，把这些方法移到一个新类中，使“共同传递”数据项操作在一个类的实例变量上，并且以一个类的实例而不是一个独立的项来传递

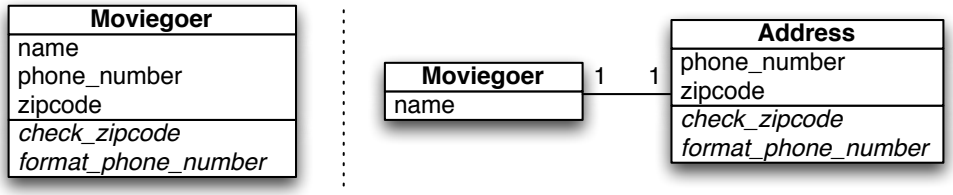


图 11.9: 重构前的UML类图(左)和从Moviegoer中提取了Address之后的类图(右)

代码文件：ch_designpatterns/code/ocp_violation.rb

```
class Report
  def output
    formatter =
      case @format
      when :html
        HtmlFormatter.new(self)
      when :pdf
        PdfFormatter.new(self)
        # ...etc
      end
  end
end
```

图 11.10: Report 类依赖基类Formatter和子类HtmlFormatter及PdfFormatter。因为基于report格式进行显式的分发，添加一个新类型的report输出需要修改Report#output，并且可能需要修改Report的其他有同样逻辑的方法——造成了所谓的 霰弹式修改

代码文件：ch_designpatterns/code/ocp_fix_abstract_factory.rb

```
class Report
  def output
    formatter_class =
      begin
        @format.to_s.classify.constantize
      rescue NameError
        # ...handle 'invalid formatter type'
      end
    formatter = formatter_class.send(:new, self)
    # etc
  end
end
```

图 11.11: Ruby的元编程和鸭子类型可以让有一个使用抽象工程模式的优雅方案。Rails提供的classify用来转换`snake_case`为`UpperCamelCase`。Rails提供的语法糖constraints是用来在接受者上调用Ruby的自省方法Object#const_get。我们也处理了formatter类的无效值的情况，而不好的代码是不处理的

11.4 开闭原则

SOLID中的开闭原则(OCP)强调了类应该“对扩展保持开放，但是对修改保持关闭”。也就是说，应该可以扩展类的行为，而不会修改其他类或者应用程序依赖的已存在代码。

虽然添加一个从基类继承的子类是一种扩展类的方法，但是通过该类它自己往往是不够的。图11.10显示了为什么基于case的分发逻辑——一种Case语句设计味道——可能会违背OCP。

取决于具体的案例，许多设计模式能够起作用。在图11.10中的有味道的代码试图解决的问题就是，当它存储在@format实例变量中时，Formatter期望的子类直到运行时才知道。 **抽象工厂模式** 提供了一个通用的接口来实例化对象，这些对象具体是哪个子类通常在运行时才知道。Ruby的鸭子类型和元编程（metaprogramming）使得这种模式能够有一个非常优雅的实现，如图11.11所示（相比而言，在静态类型语言中，为了“绕过”类型系统的要求，我们不得不对每一个子类都创造一个工厂方法，并且让其都实现一个通用的接口——工厂模式由此而得名）。

另一种方法就是利用 **策略模式** 或者 **模板方法模式** 。两种模式都支持这样的情形：虽然有一个通用的途径来做一个任务，但是这个途径有许多可能的变种。这两种模式之间的区别是在哪个层次来表示通用性。对于模板方法模式，虽然每一步的实现是不同的，但是所有变种的步骤集合是一样的，因此模板方法模式通常使用继

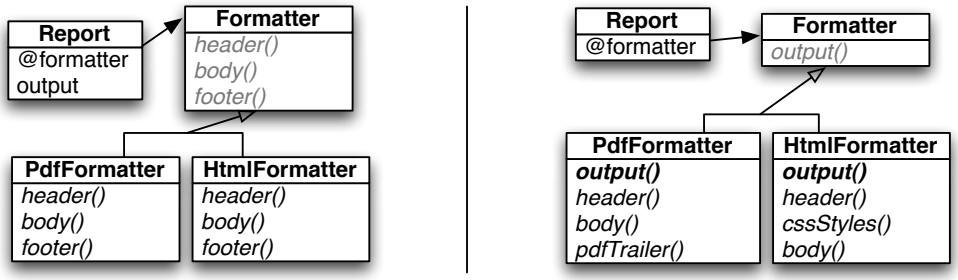


图 11.12: 在模板方法中（左图），扩展点是header, body及footer，因为Report#output方法调用了@formatter.header，@formatter.body等，每一个都委托到合适的子类的特殊副本中。（浅灰色指示方法仅仅委托到一个子类中。）在策略方法中（右图），扩展点是在output方法中，它把整个任务委托给一个子类。委托是组合中如此常见的一种方式，以至于一些人把它称为 委托模式

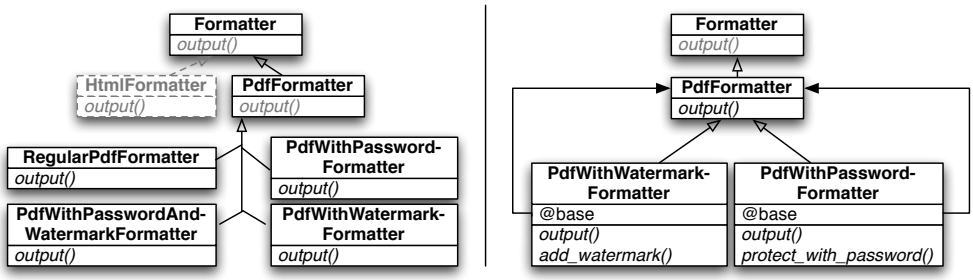


图 11.13: （左图）试着使用继承来解决Formatte问题导致的子类成倍增加。左图中的问题显示了为什么你的类设计应该“让组合优先于继承”。（右图）一个使用装饰器模式的更优雅的解决方法

承来实现。对于策略模式，全局的任务是一样的，但是在每个变种中的具体步骤集合可能不同，因而策略模式通常用组合来实现。图11.12显示了每一个模式如何被应用到报告格式生成的例子。如果每一种模式都遵循同样高层次的步骤——例如，产生头部，产生报告主体，然后产生脚部——我们能够使用模板方法模式。在另一方面，如果这些步骤本身是相当不同的，使用策略模式应该更明智。

策略模式一个广泛使用的例子就是OmniAuth（参见5.2节）：许多应用程序需要第三方的认证，步骤会因认证提供者而有许多的不同，但是API对它们来说全都一样。事实上，OmniAuth甚至把它的插件称为“策略”。

当我们想给一个已存在的类 **添加** 方法，但又发现如果不修改它就做不到的时候，一个不同种类的违背开闭原则的情况出现了。例如，生成的PDF文件，可以带有或不带有密码，可以在背景中有水印或没有。两种特征都会给已经存在的PdfFormatter“带来”一些额外的行为。如果你进行了许多的面向对象编程，解决这个问题的第一反应可能是使用继承，如图11.13（左图）中显示的UML类图，但是这有四种排列的特征，因此你将会复制四个子类——这四个子类几乎是重复的。幸运地，**装饰器模式** 能够起作用：我们可以把类或方法包装在一个有着同样API，但是经过“增强”的“装饰”类或者方法中，这样做可以允许我们根据需要组装多个装饰器。图11.14显示与图11.13（右图）中PDF格式器相对应的基于装饰器模式的更优雅的设计。

Python的不幸的是，Python中的“decorators”[10] 和装饰器设计模式完全没有关系。

在现实中，Rails的ActiveSupport模块通过**alias_method_chain**提供了方法级别的装饰，如图11.15所示，它和Ruby的开放类联系起来的时候很有用。通常情况

代码文件：ch_designpatterns/code/ocp_decorator.rb

```
class PdfFormatter
  def initialize ; ... ; end
  def output ; ... ; end
end
class PdfWithPasswordFormatter < PdfFormatter
  def initialize(base) ; @base = base ; end
  def protect_with_password ; ... ; end
  def output
    @base.output.protect_with_password
  end
end
class PdfWithWatermarkFormatter < PdfFormatter
  def initialize(base) ; @base = base ; end
  def add_watermark ; ... ; end
  def output
    @base.output.add_watermark
  end
end
# If we just want a plain PDF
formatter = PdfFormatter.new
# If we want a "draft" watermark
formatter = PdfWithWatermarkFormatter.new(PdfFormatter.new)
# Both password protection and watermark
formatter = PdfWithWatermarkFormatter.new(
  PdfWithPasswordFormatter.new(PdfFormatter.new))
```

图 11.14: 为了给一个类应用装饰器，我们通过创建一个子类来“包装”（为了遵循里氏代换原则，正如11.5节所述）。这一装饰器子类，将功能上没有改变的部分委托给原有的方法或类，只是实现了添加扩展功能的额外方法。这样做使得我们然后能够通过“堆叠”装饰器来简单地“构建起”我们需要的PdfFormatter版本

代码文件：ch_designpatterns/code/decorator_alias_method.rb

```
# reopen Mailer class and decorate its send_email method.
class Mailer
  alias_method_chain :send_email, :cc
  def send_email_with_cc(recipient,body) # this is our new method
    send_email_without_cc(recipient,body) # will call original method
    copy_sender(body)
  end
end
# now we have two methods:
send_email(...)            # calls send_email_with_cc
send_email_with_cc(...)    # same thing
send_email_without_cc(...) # call (renamed) original method
```

图 11.15: 为了装饰一个已经存在的方法Mailer#send_email，我们重新打开了它的类，并且使用了alias_method_chain来装饰它。我们不用更改任何调用了send_email的类，所有的调用现在使用经过装饰的版本来发送邮件和复制发送者

下，一个更有趣的装饰器例子是我们在第2章使用过的Rack应用服务器。Rack的核心是“中间件” 模块，这一模块接受HTTP请求和返回包括HTTP响应码、HTTP头部及HTTP正文的一个三元素的数组。一个基于Rack的应用程序指定了所有请求都会经过的一组中间件组件：例如为了给一个HTTP请求添加行为）例如，拦截一些请求作为OmniAuth去初始化认证流），我们可以通过装饰基本的HTTP请求行为来实现。通过其他的其他的装饰我们添加了对SSL，性能的测量和一些类型的HTTP缓存的支持。

开闭原则的小结：

- 让一个类对扩展保持开放，但是对修改保持关闭，我们需要一些策略在将会需要的地方放置扩展点。Case语句设计味道是一个可能违背OCP的症状。
- 如果扩展点采取了一个步骤实现可能会改变的形式，策略模式和模板方法模式可能会起作用。它们两者通常和抽象工厂模式联合使用，因为要创建的变量的类型直到运行时才会知道。
- 如果扩展点采取了选择不同特征子集的形式，这些特征“添加在”已存在的行为中，装饰器模式可能会起作用。Rack应用服务器就是这样设计的。

■ **详细阐述：关闭什么？**

“对扩展保持开放，但是对修改保持关闭”是假定你预先知道将来会有什么样的扩展，因此你让类对最可能的个性开放，并对可能破坏依赖性的修改战略性地保持关闭。在我们的例子中，因为我们已有不止一个方法去做事情（格式化一个报告），允许稍后添加额外的格式化方法看起来似乎很合理，但是你并不总预先知道你将会添加什么样扩展。做最好的估计，并且在变化发生时再进行处理。

自我检查11.4.1. 这里有两条关于委托的表述：

(1) 子类把一个行为委托给一个祖先类。

(2) 一个类把某个行为委托给一个子类。

查看模板方法模式、策略模式、装饰器模式（图11.12和图11.13），看看哪一条最恰当地描述了每个模式是怎样使用委托的?

◇ 在模板方法模式和策略模式中，祖先类提供了“基本的游戏计划”，这些计划将会通过委托指定给不同的子类。在装饰器模式中，每一个子类提供了自己具体的功能，但是委托给祖先类来实现基本的功能。

Barbara Liskov
于1968年获得学位，成为美国第一个计算机科学博士之一。

11.5 里氏代换原则

里氏代换原则（Liskov Substitution Principle）（LSP）是以图灵奖获得者Barbara Liskov命名的。她在子类型中的开创性工作极大地影响了面向对象编程。非正式而言，LSP表明如果一个方法对类型T的对象起作用，也应该对类型T的子类型起作用。也就是说，所有T的子类型都应该保留T的“合约”。

代码文件：ch_designpatterns/code/lsp_square_rect.rb

```
class Rectangle
  attr_accessor :width, :height, :top_left_corner
  def new(width,height,top_left) ... ; end
  def area ... ; end
  def perimeter ... ; end
end
# A square is just a special case of rectangle...right?
class Square < Rectangle
  # ooops...a square has to have width and height equal
  attr_reader :width, :height, :side
  def width=(w)  ; @width = @height = w ; end
  def height=(w) ; @width = @height = w ; end
  def side=(w)   ; @width = @height = w ; end
end
# But is a Square really a kind of Rectangle?
class Rectangle
  def make_twice_as_wide_as_high(dim)
    self.width = 2*dim
    self.height = dim            # doesn't work!
  end
end
```

图 11.16: 行为上，rectangles有一些squares没有的能力——例如，独立设置它们的边长的能力，正如Rectangle#make_twice_as_wide_as_high

这似乎是常识，但是它会不经意间就出错。图11.16中的代码违背了LSP原则。你也许会认为**Square**是一个特殊的**Rectangle**，并且因此应继承于**Rectangle**。但是 **行为上**，当指定一个边的长度时，一个正方形 **不** 像一个矩形！当发现这个问题时，你也许会尝试在**Square**中重写**Rectangle#make_twice_as_wide_as_high**方法，因为这个方法对于**Square**是不合理的，也许会引发一个异常。但是这应该是一个 **拒绝的遗赠** ——一个经常违背LSP的设计味道。该症状就是一个子类或破坏性地重写从超类继承的某个行为，或强制地改变超类来避免这个方法（它自己可能会违背开闭原则）。这个问题在于，继承关系的目的就是共享实现，但是如果一个子类不会利用父类的实现，它也许就不值得成为一个子类。

因此，如图11.17所示，我们可以不使用继承，而使用组合和委托来修复问题。很高兴因为Ruby的鸭子类型特性，大多数情况下，使用组合和委托仍旧允许我们把

代码文件：ch_designpatterns/code/lsp_square_rect_fix.rb

```
# LSP-compliant solution: replace inheritance with delegation
# Ruby's duck typing still lets you use a square in most places where
#  rectangle would be used - but no longer a subclass in LSP sense.
class Square
  attr_accessor :rect
  def initialize(side, top_left)
    @rect = Rectangle.new(side, side, top_left)
  end
  def area      ; rect.area      ; end
  def perimeter ; rect.perimeter ; end
  # A more concise way to delegate, if using ActiveSupport (see text):
  #  delegate :area, :perimeter, :to => :rectangle
  def side=(s) ; rect.width = rect.height = s ; end
end
```

图 11.17: 与一些违背开闭原则的情况类似，错误使用继承也会导致问题。如图所示，优先选择组合和委托来替代继承，可以修复这个问题。12行显示对于应用程序使用ActiveSupport，可以使用一个简洁的语法来使用委托（所有的Rails应用程序都这样做）；非Rails的Ruby应用程序的相似功能是通过Ruby的标准库中的Forwardable模块提供的

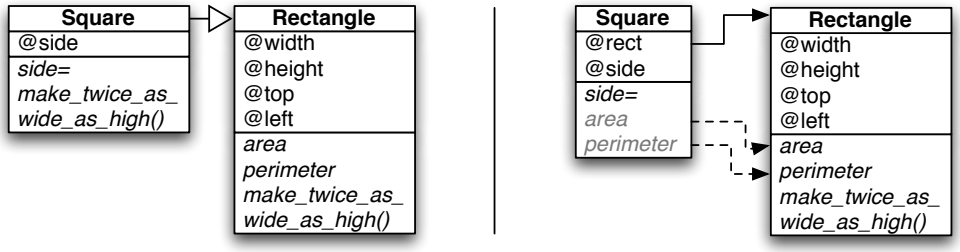


图 11.18：左图：UML类图代表了图11.16中最初的违背里氏代换原则的代码，它破坏性地覆盖了Rectangle#make_twice_as_wide_as_high。右图：重构了图11.17中的符合里氏代换原则的代码的类图

一个**Square**的实例传到一个预期**Rectangle**的地方，即便它不再是**Rectangle**的一个子类；静态类型语言将不得不引入一个显式的接口来捕获**Square**和**Rectangle**共同的操作。

里氏代换原则的小结：

- 里氏代换原则强调，起作用在某个类的对象的方法也应该对其子类的对象中起作用。当子类和其父类的行为不一样时，会违背里氏代换原则。
- **拒绝的遗赠** 设计味道，就是子类破坏性地重写父类的行为，或者强制改变父类的行为以使这个行为不会被继承——这些通常都是违背里氏代换原则的信号。
- 许多违背里氏代换原则的情况，可以通过使用类的组合而非继承来修复，即通过委托而不是继承来实现重用。

自我检查11.5.1. 为什么Ruby标准库中的**Forwardable**是以一个模块而不是一个类提供呢？

◇ 模块允许继承机制融入到任何想使用它们的类中，如果**Forwardable**是一个类将会很笨拙。换言之，**Forwardable**本身就是一个组合优先于继承的例子。

11.6 依赖注入原则

依赖注入原则（DIP，Dependency Injection Principle），有时候也被称为依赖反转，是说如果两个类彼此依赖，但是它们的实现可能发生改变，那么让它们两者依靠“注入”在两者之间的一个独立抽象接口更好。

假设RottenPotatoes现在添加了通过邮件进行市场推广——感兴趣的观众能够收到关于他们喜欢电影的折扣邮件。RottenPotatoes通过整合外部的邮件服务MailerMonkey来做这件工作：

代码文件：ch_designpatterns/code/dip_example_1.rb

```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize
5      @mailer = MailerMonkey.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10   moviegoers = Moviegoer.interested_in params[:movie_id]
11   EmailList.new.send_email_to moviegoers
12 end
```

假设这个特征是如此地成功，以至于你决定扩展这个机制，使得在Amiko社交网络的观众也可以选择将这些邮件转给他们Amiko上的朋友，使用新的包装了Amiko的RESTful API的Amiko gem来得到朋友列表，在墙上张贴消息，发送消息等等。然而，这里有两个问题。

首先，**EmailList#initialize** 对**MailerMonkey**有一个硬编码依赖。然而，我们现在有时会需要使用Amiko。这个运行时的差异，正是依赖注入原则所解决的——既然我们直到运行的时候才知道我们需要哪种类型的邮件程序，我们可以修改**EmailList#initialize**，这样才能在运行时候“注入”正确的值。

代码文件：ch_designpatterns/code/dip_example_2.rb

```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize(mailer_type)
5      @mailer = mailer_type.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10   moviegoers = Moviegoer.interested_in params[:movie_id]
11   mailer = if Config.has_amiko? then Amiko else MailerMonkey end
12   EmailList.new(mailer).send_email_to moviegoers
13 end
```

ActiveRecord被批评使用`database.yml`而不是依赖注入来配置数据库的启动，设计者大概是认为数据库在应用程序运行的时候不会发生改变。虽然依赖注入引入的嫁接也能帮助帮助实现替身或者桩，第8章中显示的Ruby的开放类和元编程让你能够在任何需要的地方插入测试嫁接。

你可以认为依赖注入原则是在两个类之间注入额外的嫁接。事实上，依赖注入原则有助于在静态编译语言中进行测试。因为通过Ruby的动态语言特性，我们可在运行时侯的任何地方创造出嫁接，实现替身或者桩等测试方法，因此这种好处在Ruby中不是很明显。

第二个问题是和**MailerMonkey**提供的简单**send_email**方法相比，**Amiko**暴露了一个不同且更复杂的API(**EmailList#send_email**委托，如第3行所示)，我们的控制方法已经被设置为在mailer对象上调用**send_email**。**适配器模式（Adapter Pattern）**能够在这里帮助我们：它被设计用来将一个已存在的API转成和一个已存在的调用者相匹配的API。在这种情况下，我们能够定义一个新的类**AmikoAdapter**将更复杂的**Amiko** API转为到们的控制器预期的简单方法，来提供和**MailerMonkey**中同样的**send_email**方法：

代码文件：ch_designpatterns/code/null_object.rb

```
class Config
  def self.email_enabled? ; ... ; end
  def self.emailer ; if has_amiko? then Amiko else MailerMonkey end ; end
end
def advertise_discount_for_movie
  if Config.email_enabled?
    moviegoers = Moviegoer.interested_in(params[:movie_id])
    EmailList.new(Config.emailer).send_email_to(moviegoers)
  end
end
```

代码文件：ch_designpatterns/code/null_object_fix.rb

```
class Config
  def self.emailer
    if email_disabled? then NullMailer else
      if has_amiko? then AmikoAdapter else MailerMonkey end
    end
  end
end
class NullMailer
  def initialize ; end
  def send_email ; true ; end
end
def advertise_discount_for_movie
  moviegoers = Moviegoer.interested_in(params[:movie_id])
  EmailList.new(Config.emailer).send_email_to(moviegoers)
  end
end
```

图 11.19: 上图：禁止一个行为的一个天真的方法是，通过在所有地方检查一个条件变量，来禁止这个操作。下图：空对象模式通过提供安全调用，但不会做任何事情的“哑巴”方法来禁止一个行为，而不必在所有的地方都检查这个条件变量

代码文件：ch_designpatterns/code/dip_example_3.rb

```
class AmikoAdapter
  def initialize ; @amiko = Amiko.new(...) ; end
  def send_email
    @amiko.authenticate(...)
    @amiko.send_message(...)
  end
end
# Change the controller method to use the adapter:
def advertise_discount_for_movie
  moviegoers = Moviegoer.interested_in params[:movie_id]
  mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
  EmailList.new(mailer).send_email_to moviegoers
end
```

适配器模式不仅是转换一个已经存在的API，而且可以简化它——例如，**Amiko** gem也提供了其他许多和email不相关的Amiko函数，但是**AmikoAdapter**只“匹配”API中与邮件相关的部分——它有时候被称为 **外观模式(Façade pattern)** 。

最后，即便是应用程序启动时就知道邮件策略的情况下，若我们有时想禁止发送邮件该怎么办呢？图11.19（上）显示了一个很天真的方法：我们把使用邮件程序的逻辑放到一个新的**Config**类中，但是如果邮件被禁止，我们仍不得不在控制器方法中通过一个条件判断来选择邮件发送的逻辑。如果在应用的其他地方都必须执行同样的逻辑，同样的条件逻辑将不得不在所有地方复制（霰弹式修改）。一个更好的选择是 **空对象模式** ，在其中我们创建了一个“空的”对象，该对象和一个实体有相同的行为，但是当这些行为被调用的时候，不会做任何事。图11.19（下）把空对象模式应用于这个例子，避免了代码中的条件判断在代码中扩散。

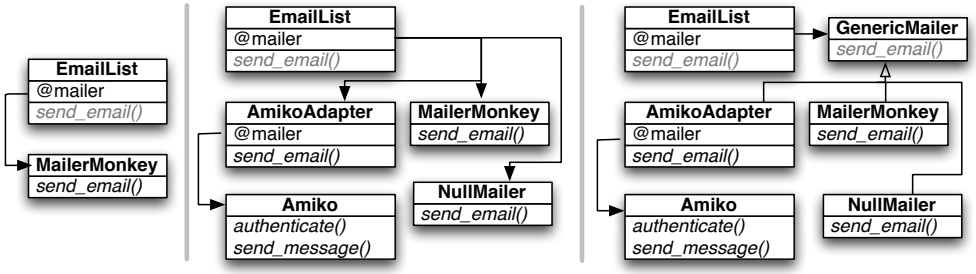


图 11.20：左图：没有依赖注入，EmailList直接依赖MailerMonkey。中图：有依赖注入，@mailer能在运行的时候使用任何的MailerMonkey、NullMailer（这实现了空对象模式来禁止邮件），或者AmikoAdapter（这在Amiko)上实现了适配器模式），所有的都有相同的API。右图：在静态类型语言中，但是在Ruby中，如果它的所有方法都是抽象的，那么这个超类在Ruby中通常会被忽略（正如这个案例），因为抽象方法和抽象类都不是Ruby语言的一部分

图11.20显示了不同版本的依赖注入例子对应的UML类图。

和适配器及外观模式有关联的模式是 **代理模式**，在其中有一个同样API对象“代替着”原有对象。客户端和代理器沟通，而不是原始的对象沟通；代理器也许会直接把一些请求转给原始对象（也就是说，代理它们），但是也可能针对不同的请求采取其他的行为，大多数情况下这么做是由于性能和效率的原因。

代理模式的经典例子是ActiveRecord中的关联（参见5.3节）。注意在 **电影有许多评论** 的关系下，可以写作**r=@movie.reviews**。**r**是哪种类型的对象？虽然我们已经知道了我们可以把**r**当成一个枚举集合，但是它实际上是一个响应所有集合方法的代理对象（**length**，**<<** 等等），只有在必要的时候才会查询数据库。另外一个使用代理模式的例子就是在无法连到Internet情况下发送邮件。如果基于Internet的邮件服务是通过**send_email**方法，一个代理对象能够提供仅仅把邮件存储在本机硬盘上的**send_email**方法，直到下次计算机连接到了Internet才发送。代理隐藏了客户（邮件程序）在没有连接到Internet的时候必须修改行为的细节。

■ **详细阐述：依赖注入违背了开闭原则吗？**

你也许会怀疑我们通过增加第二个类型的邮件服务的“修复”行为是否违背了开闭原则，因为添加第三个邮件发送程序将会要修改**advertise_discount_for_movie**。如果有理由相信你随后确实会添加另外的邮件发送程序，那么你可以将其和11.4节介绍的抽象工厂模式联合使用。这个场景是一个需要做出这样判断的例子，我们是否应该把另外的发邮件程序作为你想保持开放的扩展点，还是你觉得应用程序不会很好地适应修改，因而选择关闭。

依赖注入原则的小结：

- 依赖注入在两个类之间插入一个嫁接，通过传递（注入）一个取值在运行时候才知晓的依赖，而不是一个在源码中的硬编码。
- 因为依赖注入经常被用来确定在运行时使用哪种集合的实现，它经常和适配器模式一起使用，适配器模式是一个将API转换成客户希望使用的类。
- 适配器模式的变种包括外观模式和代理模式，在外观模式中API不仅仅是被适配，也被简化。代理器模式中API是类似的，但是其行为是在不改变客户（API的调用者）行为的情况下，根据不同的使用条件进行改变。
- 空对象模式是另外一种模式，它用安全的无效行为代替笨拙的条件来禁止某种特征。

自我检查11.6.1. 为什么正确使用依赖注入对静态类型的语言有更大的影响?
◇ 在这些语言中，你不能够创造像Ruby这样的动态语言在运行时能改写一个硬编码的行为的嫁接，因此嫁接必须预先通过注入依赖来实现。

11.7 迪米特法则

这个名字来自于适应性和面向方面编程方面的迪米特项目，它也是希腊神话中的农业女神，表示以"自底向上"的方式来编程。

非正式地说，迪米特法则（Demeter Principle）或者 **迪米特定律** 就是："和你的朋友交谈——不要和陌生人保持亲密。"详细地说，一个方法能够调用它自己类中的其他方法，和自己类的实例变量中的方法，其他的都是禁止的。迪米特法则最开始并不是SOLID原则中的一部分，正如图11.4所解释的，但是因为其在Ruby和SaaS中被大量使用，我们把它包含在这里了。并且我们使用SOLID中的**D**代表它。

迪米特法则很容易用例子来阐述。假设RottenPotatoes和电影院有合约，这样观众只要在RottenPotatoes中有余额（例如，通过接受电影院的礼品卡）就能够直接购买电影票。

图11.21显示了一个违背迪米特法则的关于行为的实现。如果改变Wallet的实现，就会引发一个问题——例如，如果我们把**credit_balance**改为**cash_balance**，或者添加**points_balance**来允许观众通过收集土豆点成为顶级评论人。突然间，与**Wallet**关系隔两层的**MovieTheater**类将不得不随之改变。

两个设计味道能够警告我们可能违背了迪米特法则。一个就是 **不合适的亲密关系**：**collect_money**方法直接操作**Wallet**的**credit_balance** 属性，即使管理属性是**Wallet**自己的责任（当同类型的不合适亲密关系在一个类中反复出现的时候，它有时候被称为 **特性依恋（feature envy）**，因为**Moviegoer**希望它有**Wallet**管理特征的"访问权限"。）另外一个在测试中引发的气味就是 **连环残骸模仿（mock trainwreck）**，其出现在图11.21的25~27行：为了测试违背迪米特法则的代码，我们发现当在测试中调用方法时，我们自己将会设置一个具有"连锁"反应的模仿。

委托再一次帮了我们的忙。如图11.22（上图）所示，一个简单的改进就是委托**credit_balance**属性。但是最好的委托在图11.22（下图），因为在这个委托方式中，支付行为完全被包装在**Wallet**中，包括决定何时对失败支付引发一个错误。

在任何你感觉"费劲地通过"一个接口来完成某些事情的时候，都能够引发不合适的亲密关系和对迪米特法则的违背。因此把自己暴露给依赖类的具体实现实际上应该不关你的事。三个设计模式可以帮助避免经常会导致对迪米特法则的违背的场景。一个是访问者模式，在其中一个数据结构被遍历，并且提供回调函数来对数

代码文件：ch_designpatterns/code/demeter_example.rb

```
1  # This example is adapted from Dan Manges's blog, dcmanges.com
2  class Wallet ; attr_accessor :credit_balance ; end
3  class Moviegoer
4    attr_accessor :wallet
5    def initialize
6      # ...setup wallet attribute with correct credit balance
7    end
8  end
9  class MovieTheater
10   def collect_money(moviegoer, amount)
11     # VIOLATION OF DEMETER (see text)
12     if moviegoer.wallet.credit_balance < amount
13       raise InsufficientFundsError
14     else
15       moviegoer.wallet.credit_balance -= due_amount
16       @collected_amount += due_amount
17     end
18   end
19 end
20 # Imagine testing the above code:
21 describe MovieTheater do
22   describe "collecting money" do
23     it "should raise error if moviegoer can't pay" do
24       # "Mock trainwreck" is a warning of a Demeter violation
25       wallet = mock('wallet', :credit_balance => 5.00)
26       moviegoer = mock('moviegoer', :wallet => wallet)
27       lambda { @theater.collect_money(moviegoer, 10.00) }.
28         should raise_error(...)
29     end
30   end
31 end
```

图 11.21: 12行包含了一个违反迪米特法则的情况：虽然 MovieTheater了解Moviegoer很合理，但它也知道Wallet的实现，因为它“通过”wallet属性来操作wallet的credit_balance。除此之外，我们也在MovieTheater中处理“资金不够”的问题，尽管看起来这一操作逻辑上应该属于Wallet

代码文件：ch_designpatterns/code/demeter_delegation_1.rb

```
# Better: delegate credit_balance so MovieTheater only accesses Moviegoer
class Moviegoer
  def credit_balance
    self.wallet.credit_balance  # delegation
  end
end
class MovieTheater
  def collect_money(moviegoer,amount)
    if moviegoer.credit_balance >= amount
      moviegoer.credit_balance -= due_amount
      @collected_amount += due_amount
    else
      raise InsufficientFundsError
    end
  end
end
```

代码文件：ch_designpatterns/code/demeter_delegation_2.rb

```
class Wallet
  attr_reader :credit_balance # no longer attr_accessor!
  def withdraw(amount)
     raise InsufficientFundsError if amount > @credit_balance
     @credit_balance -= amount
     amount
  end
end
class Moviegoer
  # behavior delegation
  def pay(amount)
    wallet.withdraw(amount)
  end
end
class MovieTheater
  def collect_money(moviegoer, amount)
    @collected_amount += moviegoer.pay(amount)
  end
end
```

图 11.22: 上图：如果Moviegoer把credit_balance给他的wallet，MovieTheater 不再需要知道wallet的实现。然而，当它应该仅仅是Moviegoer或者Wallet的责任时，把支付 行为 （从信用余额减去支付）暴露给Moviegoer仍旧是不可取的。下图：委托支付行为，而不是通过它完成的属性，解决了这个问题并且消除了迪米特法则的违背

代码文件：ch_designpatterns/code/observer_example.rb

```
class EmailList
  observe Review
  def after_create(review)
    moviegoers = review.moviegoers # from has_many :through, remember?
    self.email(moviegoers, "A new review for #{review.movie} is up.")
  end
  observe Moviegoer
  def after_create(moviegoer)
    self.email([moviegoer], "Welcome, #{moviegoer.name}!")
  end
  def self.email ; ... ; end
end
```

图 11.23: 一个邮件列表子系统观察其他模型，因此它能根据某些事件生成邮件。观察者模式完美胜任这一任务，因为它把所有关于什么时候需要发送邮件的考虑都收集到了同一个地方

据结构的每一个成员执行操作，允许你“访问”每个元素而又忽视数据结构的组织方式。事实上，“数据结构”能够在你访问不同的节点的时候被懒惰的进行实例化，而不是一次性地全部静态存在。这个模式的一个广泛应用的例子就是Nokogiri[11] gem，这一Ruby gem支持HTML和XML被遍历组织成一棵树，并支持在文档中搜索某个特定的元素，能够使用Nokogiri遍历文档和调用每个文档节点提供的访问方法。

访问者模式的一个特殊类型就是 **迭代器模式** ，其在Ruby中是如此的流行，以至于Ruby使用者基本不把它看成一个模式（当你任何时候使用**each**时）。迭代器分离了遍历一个集合的实现和你想对每一个元素实施的操作。没有迭代器，操作可能要“深入”到集合中，这样就会产生关于集合是怎样组织的细节的这种不合适的亲密关系。

观察者模式首先是在**Smalltalk** 的MVC框架中实现的，Ruby从其中继承了它的对象模型。

最后一个可能会帮助避免违反迪米特法则的设计模式是 **观察者模式** 。观察者模式用在当一个类（观察者）想知道其它类（对象）在做什么，但是又不知晓每个对象实现的细节的情况下。观察者模式为对象提供了一个标准化的方式，让它们维护其观察者列表，当任何它们有兴趣的状态发生改变时，会自动通知观察者，其使用一个狭小的接口来分离两个不同的概念：观察和每个观察者将会对观察到的信息采取什么行为。

Ruby的标准库包括了一个称为**可观察的**mixin[12]，Rails的ActiveSupport提供了一个更简洁的观察者，让你可以观察任何模型的ActiveRecord生命周期钩子（**after_save**等等），在5.1节曾介绍过。图11.23显示了给RottenPotatoes添加一个“订阅”两种状态变化的**EmailList**是多么简单：

(1) 当一个新的评论被添加时，它邮件通知那些所有评论了该电影的观众。

(2) 当一个新的观众注册时，给他发送一份“欢迎”邮件。

除了ActiveRecord生命周期的钩子外，我们将会在第12章中遇到的Rails 缓存，这是另一个被实际使用的观察者模式例子：为了知道什么时候模型的实例变得过期及应该从缓存中移除，每种ActiveRecord模型的缓存都在观察模型的实例。观察者不需要知道被观察者的具体实现——它仅仅需要在正确的时候被调用，像迭代器和访问者一样。

在结束本节前，我们需要指出一个看起来像是违背了迪米特法则，但是实际上却没有的例子。在Rails的视图（例如，用于**评论**）中会经常看到如下的代码：

代码文件：ch_designpatterns/code/demeter_in_view.html.haml

```
%p Review of:   #{@review.movie.title}
%p Written by:  #{@review.moviegoer.name}
```

这些违背了迪米特法则吗？这是一个主观判断：严格地讲，一个review不应该知道movie的具体实现，但是很难说创建委托方法**Review#movie_title**和**Review#moviegoer_name**会在这个例子中提高可读性。Rails社区的主流观点是这样的view是可接受的，其目的是为了展示对象的关系，也展示那些在视图代码中的关系，像这样的例子通常是允许存在的。

迪米特法则的小结：

- 迪米特法则陈述了一个类不应该知道与此类相隔一层以上的协作类的细节，也就是说，它能够访问自己的和与其协作的类中的实例方法，但是不能在**协作类**的协作者中访问。
- 不恰当的亲密关系是一种设计味道，它有时在单元测试中表现为一个连环残骸式的模仿(mock trainwreck)，可能是一个违背迪米特法则的信号。如果一个类显示了许多和其他类不合适亲密关系的实例，它有时候被说成和其他类有特性依恋。
- 委托是解决这些违背的关键机制。
- 设计模式包括了一些没有违背迪米特法则的常见类操作，包括迭代器模式和访问者模式（让行为从一个集合的遍历分离），以及观察者模式（分离有趣的事件通知和被观察者的细节）。

■ 详细阐述：观察者、访问者、迭代器及Mixins

因为鸭子类型和mixins，对于许多设计模式，Ruby比静态类型语言能够用更少的代码表达。Wikipedia上的**观察者模式**、**迭代器模式**和**访问者模式**都是明确基于Java的例子来阐述的。和Ruby内部基于each的迭代器对比，静态类型语言通常提供外部的迭代器和访问者。在其中你设置一个集合的迭代器和显式地问集合是否有更多的元素，有时候需要许多控制来对这个类型系统起作用。同样地，观察者通常需要修改对象类，这样它们能够实现一个**可观察**的接口，但是Ruby的开放类允许我们跳过这些步骤，如图11.23所示：从程序员的角度，所有的逻辑都在观察类中，不是在对象中。

自我检查11.7.1. Ben Bitdiddle是一个迪米特规则违背的纯化论者，他反对电影具体视图中的**@movie.reviews.average_rating**表达，该表达显示了一个电影的评价影评得分。你将会怎样安慰Ben Bitdiddle并解决这个违背？

◇

代码文件：ch_designpatterns/code/demeter_exercise.rb

```
# naive way:
class Movie
  has_many :reviews
  def average_rating
    self.reviews.average_rating # delegate to Review#average_rating
  end
end
# Rails shortcut:
class Movie
  has_many :reviews
  delegate :average_rating, :to => :review
end
```

自我检查11.7.2. 尽管对于解决迪米特法则违背，“委托是关键的解决问题的机制”，当你发现自己从A类到B类委托了许多方法，仅仅是为了解决C类中的迪米特法则违背，你为什么需要觉得有问题呢？
◇ 你可能会问自己在C类和B类中是否应该有一个直接的关系，或者A类是否对B类有一个“特性依恋”，表明A和B之间的责任分离可能需要再设计。

11.8 计划文档的视角

计划文档的好处就是仔细的先期计划能够得到一个拥有好的软件架构的产品，能够很好地使用了设计模式。正如第1章提到的，这些先期计划反映在了 **大的预先设计** 的可选择阶段中。

一个计划文档开发团队以 **软件需求条款（SRS）**（参见7.10节）开始，在其中团队把任务划分成一系列的问题。对于每个问题，团队寻找能够解决问题的一个或多个架构模式。然后团队进入到下一个层级的子问题，并且选择匹配它们的设计模式。这里的哲学是从其他人的经历学习作为模式，来避免重复前人已经犯过的错误。对于更有经验的工程师而言，另外一个获得反馈的方法就是进行 **设计评审** (参见10.7节)。注意在计划文档中，设计评审可以在任何代码都没有写之前进行。

和敏捷开发相比，在计划文档中以一个好的设计开始需要更多的努力。正如Martin Fowler在其 **《设计死了吗？》**[13]一书中所指出的，对敏捷开发的常见批评就是它鼓励开发者在没有进行任何设计之前就开始编码，并且太依赖重构和随后的修复。有时候批评者会这样说，你能够通过把一些东西聚在一起建造一个狗窝，但是无法以那种方式建造摩天大楼。

敏捷开发的支持者认为计划文档模式也不好：它直到设计完成才允许编码，很难有自信说设计将会被实现或者它是否真正的满足了客户的需求。当架构师/设计师不会编写代码或者可能不会接触当前的编码惯例和工具，这种批评尤其成立。这个问题的结果是，正如敏捷开发的拥护者所说，就是当编码开始后，设计反正是不得不改变的。

双方都有自己的道理，但是批评能够通过另外一种更微妙的方式表达“进行多少预先的设计是最好的呢？”例如，敏捷开发者计划把持久性存储作为其SaaS应用的一部分，即便他们写的第一个BDD和TDD不会和数据库有关系。一个更加微妙的问题是横向扩展性问题。正如我们在第2章提到和接下来会在第12章中进行更全面讨论的，成功的SaaS设计者 **必须** 在早期考虑横向扩展性。即便扩展性问题要在几个月之后才会遇到，项目中的早期设计决定会损坏扩展性，如果不通过大量改写和重构，很难去改变这些错误的设计。

解决这个难题的一个可能的方法就是Folwer的文章中提到的一个粗略方法。如果你先前完成的一个项目中遇到了设计限制和元素，在接下来类似的新项目中计划考虑这个就很好，因为你先前的经验很可能在这次带来合理的设计决定。

小结：
计划文档过程在软件开发过程中有一个显式的设计阶段，在其中会很自然地使用设计模式。一个潜在的缺点就是不确定在开始编写代码和系统进化之后，先前的架构和设计模式是否需要改变。相比之下，敏捷过程在编码开始之后，是依靠重构来和设计模式相配合，虽然有经验的开发者会更依靠他们先前类似的项目，来规划他们可能需要的软件架构和设计模式。

自我检查11.8.1. 对或错：敏捷设计这种提法本身就是矛盾的。
◇ 错，虽然在敏捷开发中没有明显单独的设计阶段，在敏捷开发中成为常态的重构，可以充分融合设计模式。

11.9 谬论和陷阱

陷阱：过分依赖模式或依赖模式不足。

对于每一个我们看到过的工具和方法论，盲目地遵循设计模式是一个陷阱：当我们的问题可以利用一个已经被证明的解决方案时，使用模式能够帮我们找到方法，但是它们不能确保产生漂亮的代码。事实上，四人组作者特别警告和 **反对** 以使用的设计模式数量来衡量一个设计。另外，如果在设计周期中太早地使用设计模式，即便你可能不需要一个通用的方法来解决当前的问题，你也会试着去实现一个完全通用的模式。这将使设计更复杂，因为和不需要通用级别的代码相比，大多数设计模式需要 **更多** 的类、方法和非直接的层次。相比之下，如果太迟地应用设计模式，你可能会冒着掉进反模式和过度重构的风险。

怎么去做呢？通过边做边学来提高品味和判断。在你前进的过程中你将会犯错，但是你关于如何处理可工作和可维护的代码的判断能力将会迅速提高。

陷阱：过分依赖UML图表或其他图表。

图表的目的是沟通。阅读一个UML图表并不会比阅读用户故事或良好分解的TDD测试容易。在有助于清晰化一个类架构的时候创造一个图表；不要把它们作为一个拐杖来依靠。

谬论：动态语言不需要SOLID原则

正如在本章所见的，SOLID所强调的一些问题并不会出现在像Ruby这样的动态类型语言中。然而，SOLID原则仍旧代表了好的设计；在静态语言中，有一个更具体的预期开销来忽略它们。在动态语言中，尽管存在着一些不需要SOLID原则的机制，而是使用动态特性让代码更优雅和无重复。使用动态语言特性带来的相应风险是代码风格更易变得懒散，从而带来丑陋的反模式的代码。

陷阱：一个类中有大量的私有方法。

你也许已发现声明为**private**的方法很难测试，因为根据定义，它们只能被那个类的方法实例调用——意味着它们不能被RSpec测试直接调用。虽然你能够通过一种方法来临时让方法变成public（**MyClass.send(:public,:some_private_method)**），但是private方法足够复杂，以至于它们自己的测试应该考虑设计味道：它们自己的方法可能太长，违背了SOFA中的**S**hort原则，包含这些类的方法可能会违

背 **单一责任原则** 。在这种情况下，考虑提取一个只会被原有类调用的协同类（collaborator class），并且让这个协同类的方法都是公共的，从而提高可维护性和测试性。

陷阱：在initialize中实现工厂模式。

11.4节经给出了一个抽象工厂模式的例子，在其中正确的子类构造器可以直接被调用。另外一个常见的场景就是你有一个类**A**和两个子类**A1**和**A2**，你想调用**A**的构造器返回一个新的正确子类对象。你一般不能把工厂逻辑放在**A**的**initialize**方法中，因为按照定义，这个方法必须返回一个类**A**的实例，可以代替的是，给出一个不同名字的工厂方法，比如**create**，让它成为一个类方法，并且在**A**的构造器中调用。

代码文件：ch_designpatterns/code/factory_pattern_pitfall.rb

```
class A
  def self.create(subclass, *args) # subclass must be either 'A1' or 'A2'
    return Object.const_get(subclass).send(:new, *args)
  end
end
```

11.10 结束语：框架包含了设计模式

> 给一个数字计算机准备程序的过程相当有吸引力，不仅仅是因为经济和科学上的回报，也是因为它是一种如创造诗歌或歌曲般的美的体验。
>
> ——Donald Knuth

四人组最开始的23条设计模式在其书本发行之后就快速地传开了。有许多的设计模式资料（Cunningham 2013; Noble and Johnson 2013），以及一些被裁减为针对特定问题领域的，比如用户界面（Griffiths 2013; Toxboe 2013）。

一个问题就是新手开发者即便读了四人组的书籍或者学习了这些资料，也很难知晓应该使用哪种模式。如果你对给定的一个设计模式之前没有经验，你试着以预期的方式来设计，你可能会弄错，因此不应该等到以后再添加，而是在真正需要的地方使用。

好的消息就是像Rails之类的框架包装了其他人的设计经验，来提供抽象和已经被证明是可重用的设计约束。例如，你在设计应用行为时可能不会采取REST，但是这种方法被证明会让Web故事的扩展性设计更具连续性。四人组以他们自己的方式区分设计模式和框架，是为了让什么是设计模式更清晰——更抽象，减少关注点，以及不针对具体的问题领域——今天的框架是新手开始学习设计模式很好的方法。通过查看一个框架中的模式的代码，你能够获得如何创建你自己的基于设计模式的代码经验。

11.11 深入学习

《设计模式》（Gamma et al. 1994）是四人组关于设计模式的经典之作，虽然权威，但是它会比其他书读起来慢一点，并且其中的例子是基于C++的。**《Ruby中的设计模式》**（Olsen 2007）可以看成是四人组模式关于Ruby例子的详细子集。它也讨论了Ruby语言中不需要的模式。**《代码整洁之道》**（Martin 2008）全面地探讨了SOFA和SOLID原则对使用设计模式的促进。

不是列出一系列的模式，而是通过展示他们修复的设计味道来激发模式的一个子集。**《Rails反模式》**（Pytel and Saleh 2010）给出了很好的例子，其例子是真实工作的代码是如何以一个好的设计开始，随着时间逐渐变得混乱，然后怎样通过重构来美化和流水化，这通常是通过一个或多个合适的设计模式。图11.24显示了那些重构的一些例子，主要是从Martin Fowler的重构方法在线目录[14]和综合性书籍（Fields et al. 2009）和中摘录的。

气味	描述	修复
注释除臭剂，不合适的命名	含糊的变量或方法命名需要大量的注释。	通过描述性名字和（如果有必要）通过在对应的代码中强调其他味道，来减少所需要的注释。
惰性类，数据类	一个做很少事的类，例如，除了给对象提供getter和setter外，没有其他逻辑的类。	合并把数据对象包装到其他类中的方法。
重复代码，组合爆炸	同一个类中，几乎同样的代码在多个方法中只有细微改变的重复。	使用如块或者**yield**（参见3.8节）这些DRY策略来提取公共的部分，使用模板方法模式或者策略模式（参见11.4节）来提取帮助方法（参见9.6节）。
平行的继承层次结构	几乎一样的代码在在继承自不同祖先的不同类中只有细微的改变；例如，大量的使用稍微不同的数据或行为组合的代码片段。	提取共同部分到自己的类中和委托给那个类（参见11.7节）。如果有不同祖先的类需要这个功能，尝试提取它到一个可以混合的模块中。

图 11.24: 一些通过一个局部修改就能简单修复的味道。这些是摘录在Fowler的 重构，Ruby版本 （Fields et al. 2009）

最后，M.V. Mäntyllä 和C. Lassenius创建了一个代码味道和设计味道的在线分类系统[15]，并且分组在“The Bloaters”、“The Change Preventers”等具有叙述性名字的条目中。这个网站总结在了他们在这个方向上于2006年发表的杂志文章[16]。

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

W. Cunningham. Portland pattern repository, 2013. URL `http://c2.com/ppr/`.

J. Fields, S. Harvie, M. Fowler, and K. Beck. **Refactoring: Ruby Edition** . Addison-Wesley Professional, 2009. ISBN 0321603508. URL `http://www.amazon.com/Refactoring-Ruby-Jay-Fields`.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software** . Addison-Wesley Professional, 1994. ISBN 0201633612.

R. Griffiths. HCI design patterns, 2013. URL `http://www.hcipatterns.org/patterns`.

R. C. Martin. **Clean Code: A Handbook of Agile Software Craftsmanship** . Prentice Hall, 2008. ISBN 9780132350884.

J. Noble and R. Johnson. Design patterns library, 2013. URL `http://hillside.net/patterns`.

R. Olsen. **Design Patterns in Ruby**. Addison-Wesley Professional, 2007. ISBN 9780321490452.

C. Pytel and T. Saleh. **Rails AntiPatterns: Best Practice Ruby on Rails Refactoring (Addison-Wesley Professional Ruby Series)**. Addison-Wesley Professional, 2010. ISBN 9780321604811.

A. Toxboe. UI patterns, 2013. URL `http://ui-patterns.com/`.

注释

[1] http://butunclebob.com
[2] http://martinfowler.com/eaaCatalog
[3] http://www.cs.uiuc.edu/homes/snir/PPP/
[4] http://ui-patterns.com
[5] http://foldoc.org/index.cgi?query=UML&action=Search
[6] http://en.wikipedia.org/wiki/Unified_Modeling_Language
[7] http://en.wikipedia.org/wiki/Unified_Modeling_Language
[8] http://cruise.site.uottawa.ca/umple/
[9] http://try.umple.org
[10] http://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators
[11] http://nokogiri.org
[12] http://www.ruby-doc.org/stdlib-1.9.3/libdoc/observer/rdoc/Observable.html
[13] http://www.martinfowler.com/articles/designDead.html
[14] http://martinfowler.com/refactoring/catalog
[15] http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
[16] http://www.soberit.hut.fi/~mmantyla/ESE_2006.pdf

11.12 建议的项目

练习11.1. 描述这样一个设计，给RottenPotatoes添加一个新功能，使得顾客可以在其中通过在线购买电影票。为了保持简单，假设顾客通过RottenPotatoes 购买了某部电影的电影券，并且那个票券可以在任何上映该电影的电影院中使用（RottenPotatoes不会知道某个具体的电影院）。假设你预先不知道将会使用哪种支付服务来处理信用卡收费，但是将会有某种RESTful API。在为新功能创建了用户故事和lo-fi模型后，你的设计至少需要强调下面的需求：

- 决定如何来建模资源和它们之间的关系（相关性）来支持新功能。
- 决定如何包装支付服务的交互，即使你还不知道将会使用何种服务。

对于接下来的项目，你将需要鉴别一个你将会视察的运行中的遗留软件系统。作为建议，你可以使用在Open Source Rails[1]中的开源Rails项目列表，或者选择使用本书中学生创建的两个项目中的一个：一个是ResearchMatch[2]，帮助匹配学生和他们大学的研究机会；另一个是VisitDay[3]，帮助安排学生和教员之间的见面。

练习11.2. 描述一个或多个能够应用到软件系统中的设计模式。注意：边缘图标表明来自于ACM/IEEE 2013软件工程标准的项目（ACM 2013）。

练习11.3. 对于适合一个给定的用户故事的一个简单系统，讨论和选择一个合适的设计模式。

练习11.4. 在软件设计中使用简单的模式例子。

练习11.5. 讨论和选择在系统中针对一个给定的用户故事的适当的软件架构。那个用户故事的系统设计反映了你自己中意的架构吗?

练习11.6. 以有意义的外部品质属性角度来分析软件设计，诸如可维护性和缺乏黏度。

12 SaaS非功能性需求：性能、发行、可靠性和实用安全

Barbara Liskov（**1939——**）由于对编程语言设计的基础性创新而获得2008年图灵奖。她的发明包括抽象数据类型和迭代器，两者都是Ruby语言的核心。

你永远不需要最优的性能，你只需要足够好的性能……程序员对于性能的追求太高了。

——Barbara Liskov，2011

12.1 从开发到部署 368
12.2 量化响应性 370
12.3 持续集成和持续部署 372
12.4 发行和特征标记 373
12.5 量化可访问性 376
12.6 监控和查找瓶颈 378
12.7 用缓存来提高渲染和数据库性能 379
12.8 避免滥用数据库查询 383
12.9 安全：在应用程序中保护用户数据 386
12.10计划文档的视角 390
12.11谬论和陷阱 392
12.12结束语：性能、可靠性、安全和有漏洞的抽象层 394
12.13深入学习 395
12.14建议的项目 398

重要概念

本章中，最主要的理念是当部署应用程序时如何去避免以下棘手的问题：系统崩溃，访问量激增时的响应过慢，泄露用户数据。这些非功能型特征比产品功能性特征更重要，因为这些令人头疼的问题会使顾客离去。

在敏捷生命周期中：

- 对SaaS应用最大的 **性能** 挑战是 **延迟** ，这个在有限的情况下可以通过 **过度供应** 来解决。 **Apdex** 度量标准是一种衡量应用程序是否满足 **服务水平目标**（**Service Level Objective，SLO** ） 的标准手段。
- 可以通过在一个 **平台即服务**（**Platform as a Service，PaaS**） 上运行应用程序来增加满足SLO标准的机会。 **平台即服务** 能帮你处理大部分管理和扩展。
- 后端数据库是大多数应用程序不得不放弃PaaS方案的原因，但是你可以通过使用缓存、创建索引及避免非必要的且费时的数据库查询来延长连接数据库的时间。
- **发行** 在SaaS中更具挑战性，因为你通常需要在没有停止旧版本的情况下部署新的版本。 **特征标志** 使得快速部署更简单。
- **安全性** 可以通过遵从 **最小权限原则**（**Principle of least privilege**） 和 **预设安全模式** 来加强，它们限制访问在"须知"基础及 **心理可接受原则** 下，这说明用户接口在有保护的情况下应该更简单。
- **防御编程**（**Defensive programming**） 可以在缺陷发生之前预知，这使得系统更可靠安全。

对于计划文档的生命周期：

- 性能只是一个可能的非功能性要求。
- 与敏捷生命周期相比，计划文档周期中发行的频率更低，但每一次发行更大。
- **平均失效时间**（**Mean Time To Failure，MTTF**） 是一个整体衡量标准，包括硬件错误、软件错误和操作错误。减少 **平均修复时间**（**Mean Time To Repair，MTTR**） 跟试图提高MTTF一样有效，并且它比MTTF更容易衡量。
- 安全性可以通过使系统对软件缺陷更健壮来加强，软件缺陷使得系统易于遭受攻击，比如缓冲区溢出、算术溢出和数据竞争等。

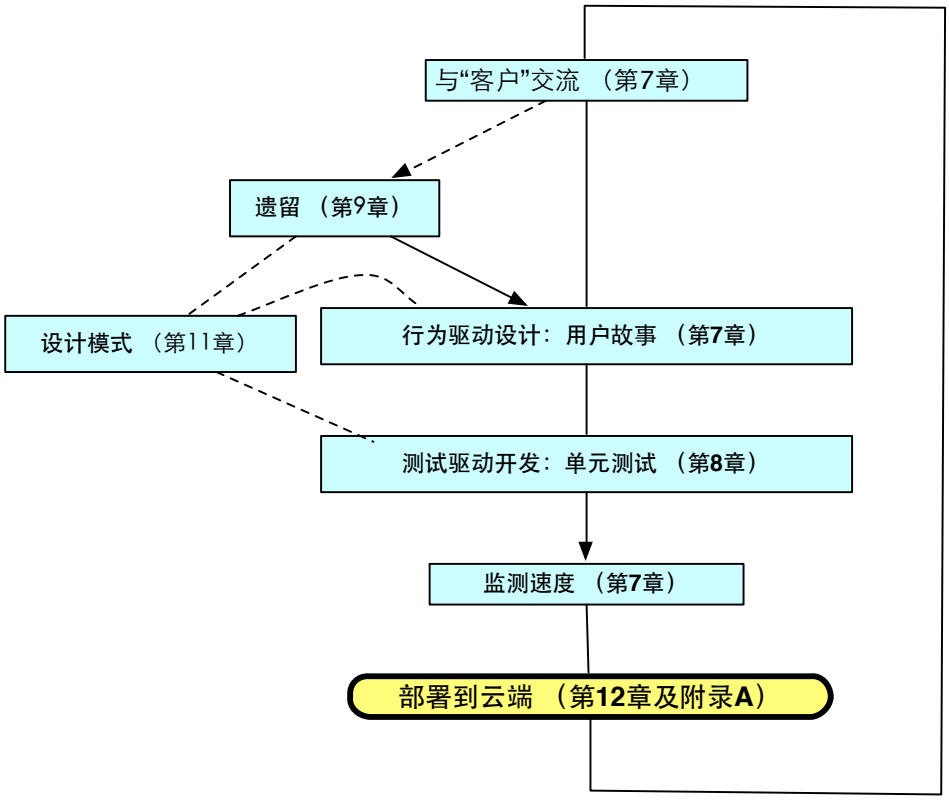


图 12.1: 敏捷软件生命周期和其与本书各章之间的关系。本章覆盖了在云端部署应用来使用户可以评估敏捷迭代

12.1 从开发到部署

用户是一种可怕的生物。没有他们，系统会无限稳定。

——Michael Nygard，**Release It!** （Nygard 2007）

从一个SaaS应用部署的那一刻起，它的行为就改变了，因为它有了真实的用户。如果它是一个面向公众的应用，它就存在着被恶意攻击和这些攻击有意外成功的可能性，但是即使是私有应用，比如内部计费系统，为了保证顺利的部署和操作，也应该设计成易于部署且易于操作的。幸运的是，正如图12.1所提醒我们的一样，可部署性是敏捷生命周期每一步迭代的一部分。很多敏捷SaaS公司每天都部署好几次，因此你可以很快地对“例行”的部署变得熟练。

SaaS部署比以前更容易了。就在几年前，SaaS开发者必须学习一些关于系统管理的知识来管理自己的产品服务器。对于小的网站，他们通常部署在共享的互联网服务提供商上（“托管主机ISP”），或者部署在运行在共享软件上的虚拟机器上（虚拟专用服务器或者VPS），或者部署在ISP的数据中心的一个或多个专业计算机上（“托管服务”）。今天，通过云计算提供横向扩展性（参见2.4节），新兴的像Heroku这样的公司提供了**平台即服务**（PaaS）：一个精心部署的软件栈已经对部署应用程序做好了准备，它为你提供很多的管理和扩展性责任，使得部署对

于开发者更加友好。PaaS提供者或者运行它们自己的数据中心，或者逐渐依靠更低级别的，如Amazon公有云的IaaS提供者，Heroku就是这样做的。另外一个涌现的PaaS是CloudFoundry，一个既可以部署在一个公司已有的服务器上或部署在一个公有云上的PaaS软件层。还有Microsoft Azure，是一系列基于Windows Server和运行在Microsoft云上的管理服务。

对于早期阶段和许多成熟的SaaS应用，PaaS现在是更加流行的部署方式：基本的扩展性问题和性能调优问题被专业的SaaS管理者解决了，他们通常比大多数开发者更有经验。当然，当一个站点变得足够大或者足够流行时，它的技术要求可能超出PaaS所能提供的，此时把运维带回"家里"会更加经济，我们将会看到这是一个主要的考虑。因此本章的目标就是帮助应用程序可能地待在PaaS友好的使用层上。确实，如果应用程序是面向内部的，那么它的最大用户数是有限的，并且和面向公有的相比，它运行在一个具有保护和不易攻击的环境中，那么你停留在这样的环境中是绝对有好处的。

正如我们将要看到的，管理应用程序的增长的一个关键是控制数据库上的需求，因为数据库的横向扩展更加的困难。本章的一个观点是，你将会面对的性能和安全问题对于小型和大型SaaS应用上是一样的，但是解决方案是不同的，因为PaaS提供者在解决部分这种问题是非常有帮助的，节省自行内建解决方案的工作。

尽管本章的标题，**性能**和**安全**术语经常被过度使用和错误地定义。以下是一个我们将会强调的更集中的关键性操作列表。

- 响应性：大多数用户等待应用程序响应需要多长时间。
- 发行管理：你怎样"原地"部署或升级应用程序而不降低其可用性和响应性（参见12.3节和12.4节）？
- 可访问性：应用程序正常的服务请求所占的比例为多少?
- 可扩展性：随着用户数量的增加，不管是持续性增长还是一次性的人气飙升，应用程序可以在不增加每个用户的运营成本的情况下，维持稳定的可用性和响应速度吗？第2章指出，在云计算上的三层SaaS应用有着非常棒的 **潜在** 横向扩展性，但是仅仅有好的设计并不能保证应用程序能够扩展（坏的设计可以保证它不会）。但是缓存（参见12.7节）和避免滥用数据库（参见12.8节）能帮得上忙。
- 隐私性：重要数据只能被授权方访问吗？例如数据只能被数据的拥有者和应用程序的管理者访问。
- 可验证性：应用程序能够保证给定的用户是他/她所宣称的那个人么？通过验证密码或使用第三方认证，如Facebook的连接或者OpenID，冒充者在没有获得用户凭证的情况下不能够成功冒充用户。
- 数据完整性：应用程序能够防止用户数据被篡，或至少能够察觉到数据已经被篡改或者破坏吗?

上述列表中的前三个项目统称为 **性能稳定** ，后三项组合在一起统称为 **安全** ，对于后三项我们将在12.9节将对此进行讨论。

小结：

- 高可访问性和高响应性、不停机下的发行管理、不增加每个用户开销下的可扩展性是一个SaaS应用的 **稳定性能** 的三个关键，而保护用户的数据是应用程序的主要安全问题。
- 优秀的PaaS供应商可以提供基础实施机制来自动处理一些维护性能稳定性和安全性的细节，但作为一名开发人员，必须在应用程序的设计与实现的各方面关注这些问题，并且使用我们在本章将会讨论的机制来实现。
- 与传统盒装软件相比，SaaS开发—操作者通常更多地涉及部署、发行以及升级应用程序，以及监控性能或安全有关的问题。

自我检查12.1.1. 应用程序扩展性的哪个方面 **不能** 在PaaS环境下自动地处理?

◇ 如果应用程序"超出"PaaS提供者提供的数据库的最大限度，你将需要手动建立一个解决方案，把数据库分割到多个不同的数据库中。这个任务是高度应用程序相关的，所以PaaS提供商不能提供一个通用的机制来做到这一点。

12.2 量化响应性

> 性能是一个功能。
>
> ——Jeff Atwood, StackOverflow联合创始人

> 速度是一个功能。
>
> ——Adam De Boor, 谷歌公司Gmail软件工程师

响应性 是一个用户采取了一个动作（如点击一个链接），和用户得到响应（如新的内容出现在页面上）之间有可察觉的延时。从严格意义上说，响应性有两个组成部分： **延时** ，开始接受新的内容的最初延时，和 **吞吐量** ，传递所有内容的时间。早在20世纪90年代中期，许多的家庭用户使用调试解调器连接网络，这种网络传递第一个包的信息花费100ms。它们可以维持一个最高56Kb（56×10^3bit/s）速度来传递剩下的内容，所以说一个大小为50KB的Web页面或大小为400Kb的图片会花费8s多时间来进行交付。然而，如今的家庭用户越来越多的使用吞吐量为1∼20Mb宽带连接，所以Web页面的响应性以延迟为主导而不是吞吐量。

由于响应性对用户行为有如此大的影响，SaaS运营商仔细地监视他们网站的响应性。当然，实际上，不是与网页交互的每个用户都需要同样的响应时间，所以评估性能需要恰当地以一系列响应时间作为特征。考虑这样一个网站，10个请求中的8个需要在100ms内完成，1个需要250ms内完成，剩下的一个在850 ms内完成即可。如果用户对于这个网站的满意阈值T是200ms，那么事实上它的 **平均** 响应时间$(8(100) + 1(250) + 1(850))/10 = 190$ms低于满意阈值。但在另一方面，20%的请求（因此，20%的用户）得到不满意的服务。我们使用以下两个定义来测量延迟，使得我们不会忽视那些用户体验很差的小部分用户。

SLA vs. SLO： **服务水平协议** （SLA）是一个在服务提供者与用户之间的合同，它为SLO是否没达到要求提供一个参考。

- 一个 **服务等级目标** （SLO）通常采用这样一种形式，一个在给定宽度的时间窗口上的延时的分位点的量化形式。例如，"在5分钟窗口内应该有一个95%的请求低于100ms的延迟。"从统计学的角度来看，延迟分布的第95个分位点必须不得超过100ms。

- **Apdex** 分数（应用程序的性能指标）是一个开放标准[4]，它计算一个简化的SLO：用一个在0和1之间（包括0和1）的数字来代表满意用户的满意度。给定一个由应用程序操作员所选定的用户满意延时T的阈值，如果响应在T时间之内完成，一个请求就是 **令人满意的** ，如果延时在T跟$4T$之间则是 **可以忍受的** ，大于$4T$就是 **令人不满的** 。Apdex分数公式是（令人满意的数目+0.5%×可以忍受的数目）/（样本的数目）。在上面的例子中，Apdex分数将是$(8 + 0.5 \times (1))/10 = 0.85$。

当然，用户所感知的总响应时间包括许多超出SaaS应用所能控制的因素。它包括DNS查找、TCP连接的设置时间和发送HTTP请求给服务器，浏览器在接受一个包含足够多内容的响应，在渲染这些内容之前所引起的延时（所谓的“到屏幕的延迟（time to glass）”，一个和“逆时针”一样奇怪的术语）。特别是在使用PaaS，SaaS时，开发者/操作者对于他们自己的应用程序中的代码路径有大部分的控制权：路由和分发、控制器动作、模型方法以及数据库访问。因此我们将会专注于这些组件的响应性的测量和提高。

Google 相信[5]这个事实对于响应性给了他们更大的压力，因此，从任何Google服务中得到一个响应不比开始联系服务慢。

对于小型的网站，一个减缓延迟的完全合理方法是在一个或多个层中进行 **过度供应（overprovision）** （提高和稳定状态相关的额外资源），正如2.4节对展示层和逻辑层的建议。几年前，过度配置意味着购买一些可能闲置的额外硬件，如今支付你所使用的模式的云计算让你只在需要的时候花钱“租用”的额外服务器。事实上，像RightScale[6]的公司提供的访问仅仅是在亚马逊EC2上包装了一层。

正如我们将会看到的， **推动我们不能做到“PaaS友好”的那些同样的问题，也会阻止我们寻找“后PaaS”的解决方案中，因为在这种方案中它们同样会造成可扩展性问题** 。因此理解它们是哪种问题和怎样解决它们，无论是否使了PaaS模式，都会帮助你。

用户对于响应性的满意度的阈值是什么？1968年一个来自于人机交互协会的经典研究发现三个有趣的阈值：如果一个计算机系统对于一个用户动作在100 ms内响应，将会被接受为即时的；在1s内，用户仍会感到一个他的动作及其响应之间是一个因果连接，但是会认为系统是迟缓的；然而在8s之后，在等待的同时，用户的注意力会转移。令人惊奇的，在30多年之后，一个2000年的学术发现和另一个独立的公司Zona2001年的研究再次确认了“8s准则。”尽管许多人认为更快的互联网和更快的计算机提升了人们的预期，8s准则仍旧作为一个通用的指导。New Relic，我们接下来会介绍其监控服务，在2012年3月报告了他们在全世界范围内监控的所有网页的平均加载时间是5.3秒，平均Apdex得分是0.86。

小结：

- 响应性测量了一个交互应用程序对用户感觉起来有多“迟缓”。在当今高速的互联网连接和快速的计算机下，响应性主要由延时决定。服务水平目标（SLO）量化了响应性目标的声明，如在一个5分钟的窗口中的99%的请求的延迟低于100ms。
- Apdex得分是一个在0.0和1.0之间的简化的SLO测量，一个网站对于请求满足一个站点相关的延时阈值T时，会得到“满分”，请求在$4T$内完成得到“一半的分数”，对于请求花费比这更长，将得不到分数。
- 不管我们是否使用PaaS，威胁到可访问性和响应性的问题都是一样的，但是试着待在PaaS层中是值得的，因为其提供的机制会缓和这些问题。

自我检查12.2.1. 对或错：从响应性的角度，更快就更好。
◇ 错，比100ms更快对于人们来说是不可感知的，只有当响应性比8s更慢或者更坏时，人们才会放弃该网站。

12.3 持续集成和持续部署

正如在1.8讨论的，在SaaS之前，软件发行一个主要而不经常的里程碑式的事件。在产品发行后，产品维护责任大部分会转移到质量保障或客户服务部门。相比之下，许多敏捷企业频繁地部署新版本（有时 **一天** 几次），且开发人员同运维人员及用户需求保持密切联系。

在敏捷开发中，既然部署已经成为了常态，因此部署需要完全自动化。我们需要做到敲入一个命令就能触发部署一个新版本软件的所有动作，包括如果某些步骤失败，在不修改发行版本情况下能够清除失败的部署。对于基于迭代的TDD和BDD，通过频繁部署你变得擅长这项任务，且通过自动部署确保每次部署都保持连续性。如前所示，Heroku还是对自动部署提供支持。类似于Capistrano[7]这样的自动工具能够在非PaaS环境中自动部署你的Rails应用。

当然，只有在应用程序经过良好测试，并且在开发模式下保持稳定，部署才能成功。虽然在本书中我们高度关注测试，但在部署过程中有两件事发生变化了。首先，由于开发和生产环环节的不同，或者用户浏览器的不同（特别是对于Javascript密集型的应用程序），部署版本和开发版本之间的性能行为可能不同。其次，部署也意味着应用程序需要以从 **未** 使用过的方式进行测试——用户提交没有意义的输入，浏览器禁止了cookies或者Javascript，攻击者试图把你的站点变成一个 **恶意软件** 的分发者（在12.9节将进一步讨论）——并且需要确保在不损害用户数据和响应性的情况下躲过这些问题。

用编译语言 例如Java，CI通常意味着编译整个应用程序和然后测试它。

对部署的代码提高保障的一个关键技术就是 **持续整合** （CI），其中的每一个改变都会推送到代码库中，引发一个集成测试集合来确保任何事不会垮掉。这个概念和第8章的`autotest`很类似。不同点在于，完整的集成测试可能会包含一个开发者在他自己的环境中通常不会运行的测试。例如：

- 浏览器兼容——不同浏览器中的正确行为在CSS或者Javascript的实现中会不同。
- 版本兼容——正确的行为在不同版本的Ruby解释器中有所不同，Rack应用程序服务器，或许多环境中托管的软件，有不同版本的Ruby gem。
- 面向服务的体系结构集成——当应用程序依赖外部访问时，正确的行为会以未预期的方式表现（非常缓慢的连接、返回大量的垃圾信息等等）。
- 压力——正如12.6节所描述的性能和压力测试。
- 硬化——防范恶意攻击的测试，正如12.9节所描述的攻击。

CI系统通常被整合到一个全面地开发过程中，而不是简单被动地运行测试。例如，Salesforce的CI系统跨许多机器并行地运行超过150000个测试。如果任何测试失败了，它对于各个提交点进行二分搜索，来探明究竟是哪次提交造成了问题，并且自动给那次代码提交的负责人发送一个错误报告（Hansma 2011）。Travis[8]，一个用于Ruby应用程序的开源托管CI系统，只要它收到GitHub的发送的 **post-receive URI** 代码推送通知，就自动运行集成测试；它然后使用OAuth（参见5.2节）来检查运行`rake test`的代码，另外一个使用`rake`任务来进行自动化的例证。SauceLabs[9]提供

了关注跨浏览器测试的托管CI：应用程序中的基于Webdriver Cucumber场景被运行在各种各样的浏览器和操作系统中，每一个测试运行都被截屏，这样能够可视化地观察那些测试在有问题的浏览器中看起来是怎样的。

虽然部署不是什么重大事件，但对于发行里程碑而言仍有一个作用：他们让用户相信新的工作正在被部署。例如，一个用户要求的特征可能需要多个提交来实现，每个提交可能包括一个部署，但是全部的特征仍旧“隐藏”在用户的接口后面，直到所有的变化完成了。“开启”这个特征将会是一个有用的发行里程碑。因为这个原因，许多持续部署流通常给特定的发行点指定不同和怪异的标签（例如Heroku的软件栈的“Bamboo”和“Cedar”），但是仅使用Git 的commit-id来识别部署将不会包括用户可见的改变。

持续整合（CI）的小结：

- CI包括在部署之前运行一组集成测试，这通常会比一个单独的开发者在它自己上面运行的更广泛。
- CI高度依赖自动化。工作流程通过当提交被推送到一个特定的代码库或者分支时，自动触发CI来进行测试。
- 持续部署（当所有的CI测试通过时，自动部署产品）可能会导致每天有多个部署，其中包括许多“面向构建”一个特征的改变不会被用户可见，它们将会在某个发行里程碑被公开。

■ **详细阐述：Staging**

许多公司除了开发和生产环境外，维护一个额外的称为 **测试站点（staging site）** 的环境。测试站点除了它通常比生产环境在规模上小点外，都是相同的，使用一个测试数据的单独数据库（也许提取自用户数据），且对外部用户封闭。其基本想法是，用最接近生产站点的环境来进行压力测试和集成测试。另外一个使用方式就是在生产中部署迁移前，测试一个和我们生产数据库非常类似的数据库上的迁移。Rails和其相关工具通过定义一个额外的`staging:`来支持分段运输：例如在`config/environments/staging.rb`和`config/database.yml`中的环境。

自我检查12.3.1. 持续部署在敏捷软件公司中非常流行，你怎样区别一个部署和一个发行之间的特征?

◇ 一个发行通常包含对用户可见的新特征，然而一个部署也许仅包含逐渐面向那些特征的新代码。

12.4 发行和特征标记

正如我们在第4章所看到的，应用程序有时候的改变需要迁移去改变数据库模式。当新的代码不能够和旧的方案兼容或者相反时，就会带来新的挑战。举个具体的例子，假设RottenPotatoes现在有一个`moviegoers`表格，表格有一个`name`列，但是我们希望改变模式有分离的`first_name`和`last_name`列。如果在改变代码之前改变模式，应用程序将会中断，因为方法查找期望的`name`列失败。如果在改变数据库模式之前改变代码，应用程序将会中断，因为新方法将会查找还不存在的`first_name`和`last_name`列。

代码文件：ch_operations/code/naive_migration.rb

```
class ChangeNameToFirstAndLast < ActiveRecord::Migration
  def up
    add_column 'moviegoers', 'first_name', :string
    add_column 'moviegoers', 'last_name', :string
    Moviegoer.all.each do |m|
      m.update_attributes(:first => $1, :last => $2) if
        m.name =~ /^(.*)\s+(.*)$/
    end
    remove_column 'moviegoers', 'name'
  end
end
```

图 12.2： 一个改变了数据库模式和修改了数据以适应这些改变的迁移，12.4节将会解释为什么没有下行迁移方法（本书配套光盘中包含了这些代码）

代码文件：ch_operations/code/schema_migration_a.rb

```
1 class SplitName1 < ActiveRecord::Migration
2   def up
3     add_column 'moviegoers', 'first_name', :string
4     add_column 'moviegoers', 'last_name', :string
5     add_column 'moviegoers', 'migrated', :boolean
6     add_index 'moviegoers', 'migrated'
7   end
8 end
```

图 12.3： 一个部分迁移仅添加列，但是没有改变或者移除任何列，12.8 节将解释了为什么索引（第6行）是一个好主意

代码文件：ch_operations/code/method_with_feature_flags.rb

```
1  class Moviegoer < ActiveRecord::Base
2    # here's version n+1, using Setler gem for feature flag:
3    scope :old_schema, where :migrated => false
4    scope :new_schema, where :migrated => true
5    def self.find_matching_names(string)
6      if Featureflags.new_name_schema
7        Moviegoer.new_schema.where('last_name LIKE :s OR first_name LIKE :s',
8          :s => "%#{string}%") +
9          Moviegoer.old_schema.where('name like ?', "%#{string}%")
10     else # use only old schema
11       Moviegoer.where('name like ?', "%#{string}%")
12     end
13   end
14   # automatically update records to new schema when they are saved
15   before_save :update_schema, :unless => lambda { |m| m.migrated? }
16   def update_schema
17     if name =~ /^(.*)\s+(.*)$/
18       self.first_name = $1
19       self.last_name = $2
20     end
21     self.migrated = true
22   end
23 end
```

图 12.4： 通过匹配它们第一个名字和最后一个名字的字符串，一个被特征标记包装的模型方法找到了moviegoers，15～22行设置了一个before-save回调函数，只要它被保存，就自动对新模式更新一个记录，因此应用程序的正常使用将会导致记录被零碎地迁移

我们可以尝试通过 **原子操作** 部署代码和迁移来解决这个问题：让服务下线，使用图12.2中的迁移来执行模式改变，并复制数据到新的列中，之后再让服务重新回到线上。这个方法是最简单的解决方案，但是有可能引起不可接受的不可访问性：一个在有成百上千行的数据库上的复杂迁移需要花费数十分钟，甚至几个小时。

第二个选择就是通过使用特征标记来把改变划分到多次部署中—— **特征标记** 是一个控制应用程序中代码执行路径的配置变量，其值在应用程序运行时可以改变。注意下面的每一步都是非破坏性的：正如我们在第9章进行的重构，如果应用程序在给定的某步出错了，那么它仍处于可运行的中间状态。

(1) 创造一个 **仅** 针对那些 **添加** 新的表格或者列的发生了改变的模式的迁移，包括一个指示当前的记录是否已经被迁移到了新模式的列，如图12.3所示。

(2) 创建$n+1$版本的应用程序，在其中每个受到模式改变影响的代码路径都被划分成两个代码路径，其中的一个是基于特征标记的值而执行。这个步骤的关键就是不管特征标记任何时候是怎样的值，正确的代码都会被执行，因此特征标记能够在不停止和重启应用程序的情况下被改变；典型的做法是把特征标记存在一个特殊的数据库表格中。如图12.4所示的例子。

(3) 部署$n+1$版本的应用程序，这可能需要把代码推送到多个服务器上，这样一个过程可能需要几分钟。

(4) 一旦部署完成了（所有的服务器代码都更新到了$n+1$版本），应用程序把特征标记的值设置为真。对于图12.4中的例子，每个记录下次因任何原因的修改都会迁移到新模式中。如果你想加速事情的进展，你可以运行一个低消耗的后台进程时而地迁移一些记录来减少应用程序额外的负载，或者当应用程序负载很低的时候，一次性迁移许多记录，如果这步某些地方出错了，关闭特征标记；代码将会回到版本n的行为中，因为新模式是旧模式的一个完全超集，并且**before_save**回调是没有破坏性的（也就是说，它会正确地在新旧模式中更新用户的姓名）。

(5) 如果所有都进展顺利，一旦所有的记录都迁移完了，部署$n+2$版本的代码，在其中特征标记被移除了，并且只有和新模式有关的代码路径被保留下来。

(6) 最后，执行一个移除旧的`name`列以及为了迁移临时创建的列的迁移（也同时移除这些列上的索引）。

如果一个数据库模式的改变是修改一个列的名字或者格式，而不是添加或者移除列，我们应该怎么办呢？其实策略是一样的：添加一个新列，移除旧的列，如果有必要重命名新的列，在每次转换中都使用特征标记，这样每个部署版本的代码对两个版本的模式都能工作。

当我们在第4章引入数据库迁移的时候，我们也提到了迁移有**up**和**down**方法，虽然在图12.3中没有**down**方法。为了防止升级出错，我们不应该包含一个吗？很奇怪，答案是不。下行迁移在开发的过程中很有用，但是在使用中是很危险的。因为它们很少使用，它们通常没有进行完整的测试，并且发现某些事出错之后，会不可避免地出现恐慌，很难有自信说下行方法能够在不引起更多破坏的情况下，能够进行你想要的工作。

即便你对下行迁移会正确地工作很自信，在你试着下行迁移后，其他开发者可能会推送不可逆的迁移。在某些情况下，你需要创建一个不可逆的迁移，在应用它的时候还需要一种从错误中恢复的方法。特征标记能够帮助你：如果某些事出错

了，可以恢复特征标记的值，这样代码就会恢复先前的行为，然后你可以慢慢的来调试问题。

小结：

- 为了执行一个应用程序代码和数据库模式都升级的复杂升级，可以使用特征标记，特征标记的值在应用程序在运行时可以随时改变。以一个包含新旧版本的迁移和代码推送开始，当中间版本在运行时，改变特征标记的值让新的代码路径使用新的数据库模式。
- 由于改变了特征标记的值，一旦所有的数据都逐步迁移了，你能够部署一个消除了旧的代码路径和旧模式的新迁移和代码推送。另一方面，在前进的过程中如果某些事出错了，你能够改变特征标记的值让程序以旧有模式和旧有代码继续工作，直到你确定是什么地方出错了。
- 总是向前部署应用程序：如果某些事出错了，用另一个解决损坏的上行迁移来修复它，而不是试着去使用一个没有在生产中进行测试的下行迁移，这可能会让事情变得更坏。

■ **详细阐述：特征标记的其他使用方式**

除了处理具有破坏性的迁移，特征标记也有其他作用：

- 预先检查：只针对一小部分用户推出新功能，为了确保新特征不会破坏任何事或者不会对整个网站性能有负面影响。
- A/B测试：对两个不同群体的用户推出同一个特征的不同版本，为了看哪个版本最大限度地提高用户的停留事件、购买率等等。
- 复杂特征：有时候一个特征的复杂功能可能需要多个增量的部署周期，例如上面描述的一个。在这种情况下，一个单独的特征标记能够被用来让特征对用户界面保持隐藏，直到新特征代码被100%部署了。

rollout[10] gem支持在所有情况中使用特征标记。

自我检查12.4.1. 下面哪个地方最合适存储一个简单的布尔特征，为什么？（a）应用程序的config目录下的一个YAML文件，（b）一个已经存在的数据库表中的一列，（c）一个单独的数据表？

◇ 一个特征标记的关键点是可以不修改程序的情况下，而允许其值在程序运行时被改变。因此（a）是一个很糟糕的选择，因为在程序运行时，不接触运行中的服务器是无法改变YAML文件的。

12.5 量化可访问性

最大的性能提高是从不可工作状态到可工作状态。

——John Ousterhout，**magic** 和Tcl/Tk的设计者

正如我们在第1章所学的，**可访问性** 是指你的网站可访问和正常工作的时间比例。例如，Google App保证[11] 其企业用户的可访问性至少有“三个九99.9%”，虽然Nygard讽刺没有纪律性的网站提供的可访问性接近“两个八88.0%”。

过度供应（Overprovisioning）不仅对减少延迟有帮助，也能够让你优雅地处理服务器崩溃：临时的一台服务器不可访问会降低$1/n$的性能，过度供应的一个简单解决方案是部署$n+1$台服务器。然而对于大规模应用，系统上的过度供应系统是不可行的：已经使用了1000台计算机的服务并不见得能负担得起添加额外的200台服务器，仅仅是为了提供冗余容量。

一个提高软件可读性的方法是让它更健壮。**防御式编程** 是这样一种哲学，它试着预测潜在的软件缺陷，并且编写代码来处理它们。下面是三个有关的例子：

- **检查输入值。** 一个常见的问题就是用户输入的值不是开发者所预期的。检查输入在一个合理的范围内，也就是对于单独的值，当有一系列数据时不会太大，以及输入集合是逻辑上连续，能够减少出问题的可能性。

- **检查输入数据类型。** 另外一个用户会犯的错误就是对一个查询输入了一个未预期类型的数据。确保用户输入一个有效类型的数据可以增加应用程序成功的可能性。

- **捕获异常。** 现代编程语言提供了在异常出现时执行代码的能力，例如算术溢出。提供能够捕获任何异常的代码，增加=当异常事件处理可以提升程序继续正常运行的能力。

另外一个可访问性的挑战就是，只有运行了很长一段时间或者极高的负载时才会导致中断的错误。一个经典的例子就是资源泄露：一个长期运行的进程最终用光了某个资源，例如内存泄露，一般是因为某个应用程序或者编程语言框架本身的设计，没有回收100%的未使用内存。**软件再生** 是长期以来人们使用的缓解资源泄露的方法。Apache网站服务器运行了许多同样的工作进程，当一个给定的工作进程已经工作了足够长的时间，这个进程停止接受请求并结束掉，被一个新的进程所取代。因为某个时间只有一个工作进程“被再生”（总能力的$1/n$），这个进程有时候被称为 **滚动重启（rolling reboot）**，大多数PaaS平台使用类似的方法。另外一个例子就是当用户会话被存储在一个数据库表格里的时候，会话存储空间可能会耗尽。这也是为什么Rails默认会把每个用户会话对象序列化到用户浏览器的cookie中，尽管使用cookie会把每个用户会话对象的大小限制在4KB。

小结：

- 可访问性测量了应用程序在某个时间段内对用户请求正常响应的比例。可访问性通常是通过“9”的数目来测量的，美国电话网络设立的金牌标准是99.999%（“5个9”相应的宕机时间是一年内5分钟），这个标准很少被SaaS应用达到。

- **防御性编程** 是在我们知道潜在的缺陷之前，通过添加代码来处理以提高可访问性。

- **软件再生** 是一种提高可访问性的方法，它通过重启一个通过轮询调度的同类型进程集合的成员，来缓和资源泄露。

自我检查12.5.1. 对于一个扩展到大量用户的SaaS应用，它必须在用户增加时维护____和____，而不增加____。

◇ 可访问性，响应性，每个用户的开销。

12.6 监控和查找瓶颈

如果你不在监控它，它极有可能已经坏掉了。

——多方面的总结

在给出了响应性和可访问性之后，我们怎样测量它们？以及如果它们没有被满足，我们该怎样鉴别是应用程序的哪个部分需要注意呢？**监控**包括收集应用程序的性能数据来进行分析和可视化。对于SaaS，应用程序的性能监控（Application Performance Monitoring, APM）是指监控**直接影响商业价值**的关键性能指标（Key Performance Indicator, KPI）。KPI本身是和具体的应用程序相关的——例如，一个网络零售商的KPI可能包括给购物车添加一个商品的响应性和用户搜索的商品在给出结果中排在前五个的比例。

SaaS应用可以通过内部和外部进行监控。内部监控，或称被动监控，通过在应用程序代码中添加数据收集代码来进行，这些代码收集应用程序本身和程序运行的环境，或者两者的监控数据。在云计算、SaaS及高效率的框架出现之前，这些监控需要在应用程序的源代码中安装周期性数据收集工具、修改程序代码，或者两者都需要。今天，托管的PaaS、Ruby的动态语言特征，及像Rails架构良好的框架的结合，允许在不修改应用程序源代码或者不安装软件的情况下就实现内部的监控。例如，New Relic[12]在不察觉的情况下收集应用程序的控制器动作、数据库查询等数据。因为数据被送回到New Relic的SaaS网站上进行查看和分析，这个架构有时候被称为远程性能监控（Remote Performance Monitoring，RPM）。免费的New Relic RPM工具可以作为Heroku的一个扩展项或者作为一个可以部署在你自己的non-PaaS生产环境的单独gem。

内部监控也能够在开发中出现，这个时候多半称为**性能分析(profiling)**。New Relic和其他监控解决方案也能够安装在开发模式下。你需要进行多少性能分析呢？如果在编写和测试应用程序时遵循最好的实践，你可能仅仅需要查看应用程序在负载下的表现。因为开发与生产环境具有不可避免的不同点，例如开发环境中缺乏真实的用户行为并使用仅针对开发的数据库，如使用SQLite3，而不是像PostgreSQL这样针对生产高度调优过的数据库。归根到底，对于敏捷开发，部署像实现基本缓存的增量修复（参见12.7节）和修复数据库的滥用（参见12.8节）是很容易的。

第二种监控就是外部监控（有时被称为探查（probing）或者主动监控），一个单独的网站向应用程序发出真实的请求来检查可访问性和响应时间。在可访问你代码的内部监控给出了具体的信息之后，你为什么还需要外部的监控呢？内部监控也许不能够发现应用程序是否迟缓，甚至会和应用一起都宕掉了。特别是如果问题不在于应用程序，而是其他因素——例如，展示层的性能问题，或者超出应用程序范围的软件栈的其他部分。外部监控像一个集成测试，是一个真实的端到端测试，如一个“从外部”只能够看到你有限代码路径的真实用户。图12.5区分了不同类型的监控和一些进行监控的工具，许多以SaaS方式处理。

一旦一个监控工具鉴别了最慢和最大开销的请求，一个测试站点服务器的**压力测试（stress testing）**或者**寿命测试（longevity testing）**能够量化那些成为了瓶颈的请求的需求水平，惠普实验室[21]维护的免费且广泛使用的命令行工具`httperf`，能够模拟指定数目的简化了的来自一个应用程序的URI序列的用户请求，并且记录有关响应时间的指标。虽然像Cucumber这样的工具能让你写出描述性场景和检查任意的复杂条件，但是`httperf`只能够遵循简单的URI序列和仅检查一个成功的HTTP响应是否来自服务器。在一个典型的压力测试中，测试工程师将会设置一些计算机来对测试站点运行httperf，并且逐步增加模拟的用户数量直到某些资源变成了瓶颈。

监控什么	关注点	实例工具
对于全世界范围的用户，我的站点的可访问性和平均响应时间是多少?	商业级别	Pingdom[13], SiteScope[14]
我的应用程序中哪些页面（视图）是最热门的? 以及用户在应用程序中遵循怎样的访问路径?	商业级别	Google Analytics[15]
哪个控制器行为或数据库查询是最慢的?	应用程序级别	New Relic[16], Scout[17]
用户出现了哪些未预期的异常或错误? 以及当错误出现时，他们在做什么呢?	应用程序级别	Exceptional[18], AirBrake[19]
在支持我的应用程序时，系统级别的进程使用的资源和健康情况如何？（Apache服务器，MySQL数据库服务器等等）	基础设施/进程级别	god[20]，**monit**

图 12.5: 不同种类的监控和给Rails SaaS应用提供的实例性工具。除了最后一行外（进程级别的健康监控）都是以SaaS的方式提供，并且对基本的监控提供了一个免费的服务类型

小结：

- 对于测试，没有一个单一类型的监控会提醒你所有的问题：你需要结合使用内部和外部（端到端）监控。
- 和早期的SaaS监控相比，像Pingdom的托管监控和像New Relic的PaaS 整合监控极大地简化了监控。
- 压力测试和寿命测试能够发现你的SaaS应用的瓶颈，以及时而地暴露潜在的错误。

自我检查12.6.1. 下面的哪些关键性能指标（key performance indicator，KPI）将会和应用程序的性能监控有关：某天计算机的CPU使用率、缓慢数据库查询的完成时间、5个最慢的视图渲染时间。
◇ 查询完成时间和视图渲染时间是相关的，因为它们直接影响响应性，而响应性通常是一个与交付到客户中商业价值有关的关键性能指标。知道CPU使用率是有用的，但是它不会直接告诉我们有关的用户体验。

12.7 用缓存来提高渲染和数据库性能

> 在计算机科学中只有两件难事：让缓存失效和命名事物。
>
> ——Phil Karlton, 被Martin Fowler引用，但他找不到准确的出处

缓存背后的思想很简单：自上一次请求之后没有改变的信息能够简单地被送回，而不是重新计算。在SaaS中，缓存能够帮助两种计算。首先，如果完成某个来自于数据库信息没有改变的行为，我们能够避免查询数据库。其次，如果处于某个特定视图或者视图片段的信息没有改变，我们能够避免再次渲染视图。（回忆一下，渲染是把Haml中包括的Ruby代码和变量转换成HTML的过程）。在许多缓存场景，我们必须强调以下两个问题：

(1) **命名：** 我们该怎样指定一些计算结果应该被缓存用于接下来的使用? 以及以确保它只会在完全一样的计算中被使用的方式命名。

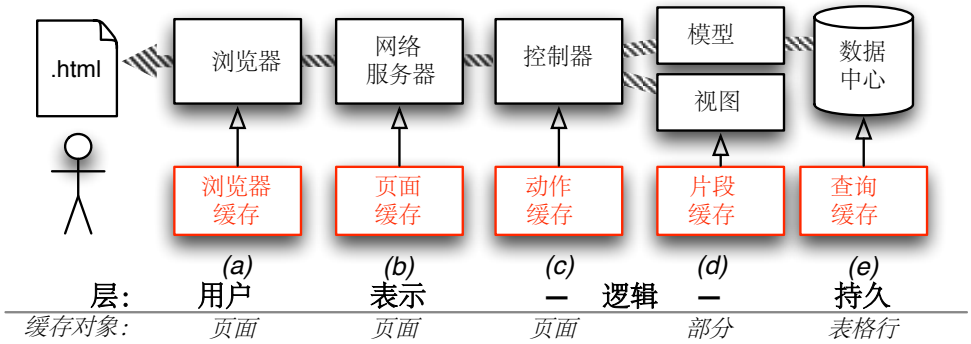


图 12.6: 多级缓存的目标就是尽可能地在接近用户的地方满足每个HTTP请求。（a）一个Web浏览器先前访问过的一个页面，在验证服务器上的这个页面没有改变之后，能够重用它本机的缓存。（b）如果不行，通过一起传递Rails，Web服务器也能利用页面缓存和Rails一起提供服务。（c）如果仍然不行，如果页面被一个由预先过滤所保护的控制器所生成，Rails也许能够在不查询数据库或者渲染模板下，从这个行为缓存来进行服务。（d）如果仍然不行，一些由视图模板所组成的片段也许在片段缓存中。（e）作为最后一种方法，数据库查询缓存可以提供最近没有改变的查询结果，如Movie.all

(2) **过期：** 怎样检测缓存版本过期了？如果信息依赖的东西改变了，又将它从缓存中移除呢？在微处理器设计中的类似问题经常被称为 **使缓存失效（cache invalidation）** 问题。

图12.6显示了在一个3级的SaaS架构中，在每一级怎样使用缓存，以及每一级中缓存了什么样的Rails的实体。我们能够做的最简单的事就是缓存整个某个特定控制器方法渲染得到的HTML页面。例如，**MoviesController#show**方法和其相应的视图只依赖某部特定电影被展示的属性（控制器方法中的**@movie**变量和Haml模板）。图12.7显示了怎样缓存一个电影的整个HTML页面，因此这个页面接下来的请求既不会访问数据库，也不会重新运行Haml渲染器，如图12.6(b)所示。

当然，这种缓存不适合被前置过滤器保护的控制器方法，例如需要用户登录的页面，这样需要执行控制器过滤。在这些情况中，将**caches_page**改为**caches_action**仍旧会执行任何过滤器，但是允许Rails在处理一个缓存了的页面时无需查询数据库或者再次渲染视图，如图12.6（c）所示。图12.9显示了这个进行简单页面和方法缓存的好处。注意在Rails的页面缓存中，被缓存对象的名字 **会忽略** URL中嵌入的参数，比如`/movies?ratings=PG+G`，因此影响页面显示效果的参数应该是RESTful路径中的一部分，比如`/movies/ratings/PG+G`。

一个介于两者之间的例子是主页面没有改变，但是布局改变了的方法缓存。例如，`app/views/layouts/application.html.haml` 也许包含了这样的消息“欢迎你，Alice”，包含了登录用户的名字。为了保证方法缓存在这种情况下正常工作，给**caches_action**传入**:layout=>false** 将会导致布局全部重新渲染，但是控制器方法（页面的部分内容）会利用方法缓存。记住既然控制器方法不会被运行，那么任何类似出现在布局中的动态内容必须在过滤前设置。

页面级别的缓存对于内容会动态改变的页面没有用处。例如，电影列表页面（**MoviesController#index**的控制器方法）会在新电影被添加或用户通过MPAA评分过滤列表时发生改变。但是我们通过观察包含了许多的表格行中的索引页面，其每一行都只依赖于某个特定电影的属性，这种情况我们仍能从缓存中获得好处。第5.1节的图5.2 显示了这种情况。图12.8显示了怎样给图5.2缓存的渲染过的HTML片段的部分添加一个单独的行，每一个片段都只和一部电影相关。

Rails提供的一个捷径是如果**缓存**增加的是一个ActiveRecord对象，这一对象对应

代码文件：ch_operations/code/action_caching.rb

```
1 class MoviesController < ApplicationController
2   caches_page :show
3   cache_sweeper :movie_sweeper
4   def show
5     @movie = Movie.find(params[:id])
6   end
7 end
```

代码文件：ch_operations/code/movie_sweeper.rb

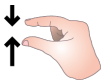
```
1  class MovieSweeper < ActionController::Caching::Sweeper
2    observe Movie
3    # if a movie is created or deleted, movie list becomes invalid
4    #   and rendered partials become invalid
5    def after_save(movie)    ; invalidate ; end
6    def after_destroy(movie) ; invalidate ; end
7    private
8    def invalidate
9      expire_action :action => ['index', 'show']
10     expire_fragment 'movie'
11   end
12 end
```

图 12.7: （上图）第2行告诉了**Rails**应该缓存**show**方法的结果。方法缓存是用一个`before-filter`过滤器来实现的，它检查是否应该用缓存中的版本，方法缓存还用到了一个`around-filter`过滤器，用于截取并缓存输出的结果。这是一个装饰器设计模式的例子（参见**11.4**节）。（下图）第3行的控制器所引用的**"sweeper"**，使用了观察者设计模式（参见**11.7**节）来添加**ActiveRecord**生命周期钩子（参见**5.1**节），使任何因为对某个特定电影更新会让过期的对象失效

代码文件：ch_operations/code/fragment_caching.html.haml

```
1 -# A single row of the All Movies table
2   - cache(movie) do
3     %tr
4       %td= movie.title
5       %td= movie.rating
6       %td= movie.release_date
7       %td= link_to "More about #{movie.title}", movie_path(movie)
```

图 12.8: 和**5.1**节的图**5.2**比较，只有第二行被添加。**Rails**将会给缓存对象生成一个基于复数的资源名字和主键名字的缓存对象名字，例如`movies/23`



的表包含一个名为updated_at或updated_on的列，如果一个片段的表格行在表格第一次缓存之后进行了更新，缓存将会自动过期这个片段。虽然如此，为了清晰化，图12.7 中的sweeper的第10行显示了如何显式地使一个其名字与增加的**缓存**相匹配的片段过期，无论底层的**movie**对象是被保存或者销毁了。

与避免运行控制器方法的方法缓存不同，检查片段缓存发生在控制器方法运行**之后**。你也许已经在好奇片段缓存是怎样帮助减少数据库的负载的。例如，假设给电影页面添加了一个区域来显示评分是**@top_5**的电影，并且给**index**控制器方法添加了一行来设置变量：

代码文件：ch_operations/code/topmovies.html.haml

```
1 -# a cacheable partial for top movies
2 - cache('top_moviegoers') do
3   %ul#topmovies
4     - @top_5.each do |movie|
5       %li= moviegoer.name
```

代码文件：ch_operations/code/topmovies.rb

```
1 class MoviegoersController < ApplicationController
2   def index
3     @movies = Movie.all
4     @top_5 = Movie.joins(:reviews).group('movie_id').
5       order("AVG(potatoes) DESC").limit(5)
6   end
7 end
```

方法缓存现在作用更小了，因为在添加一个新电影**或**评论时，**index**视图可能会发生改变（这也许会影响评分最高的5个电影）。如果控制器方法在片段缓存被检查之前运行，难道不是失掉了这一缓存的好处？因为设置控制器方法4和5行的**@top_5**会导致一个数据库查询。

很惊奇不是这样。事实上，4和5行 **不会** 导致产生一个查询：它们构建了一个如果曾被要求其结果，就能执行查询的对象！这被称为 **惰性求值**，一个来自于函数编程的 **lambda 算子** 的很强大的编程语言技巧。惰性求值被用在了Rails的ActiveRelation（ARel）子系统中，该子系统被ActieRecord所使用。真实的数据库查询直到Haml模板的第5行的**each**被调用才会执行，因为这是ActiveRelation对象首次被要求产生一个值。但是因为**cache**块那个行从第2行开始，如果片段缓存存在，这行将不会被执行，因此数据库也不会被查询。当然，你必须在你的缓存交换器中添加逻辑，使得当一个新评论被添加时，使top5的电影片段能够按正确的方式失效。

早期版本的Rails缺乏惰性查询求值，因此控制器方法不得不显式地检查片段缓存，以避免不必要的查询——非常不符合无重复的原则

总而言之，页面和片段级别的缓存是对我们从保持一样（可缓存单元）的东西中区分变换（不可缓存单元）的东西的能力的奖赏。在页面或者方法缓存中，我们把被前置过滤器所保护的控制器方法划分到使用页面缓存的一个“未保护”的方法，以及一个能够使用方法缓存的过滤器中。（在一个极端例子中，你甚至能支持类似于Amazon CloudFront的 **内容分发网络** （CDN）去复制遍布全球的数百台服务器中的页面。）在片段缓存中，使用partials隔离每个不可缓存实体到其自己能被片段缓存的partials中，例如一个单一模型实例。

没有缓存	行动缓存	与没有缓存的速度对比	页面缓存	与没有缓存的速度对比	与行动缓存的速度对比
449 ms	57 ms	**8x**	21ms	**21x**	**3x**

图 12.9: 对于一个Heroku框架中包含了1K个电影和每部电影超过100条评论的PostgreSQL共享式数据库，表格显示了通过创建时间检索前100条评论所需要的时间，时间用毫秒表示，检索包括是否使用了页面和方法缓存。数字来自于heroku logs日志文件

缓存的小结：
为了最大化地利用缓存的好处，从不可缓存单元中分离出可缓存的：根据是否需要运行一个前置过滤器，控制器方法能够被划分为可缓存和不可缓存版本，并且一部分还能够被用来将视图划分到可缓存的片段中。

■ **详细阐述：被缓存的对象存储在哪儿？**

在开发中，被缓存的对象一般是存储在本机的文件系统上。在Heroku中，Memcachier扩展把缓存的内容存储在一个基于内存的数据库`memcached`中（发音mem-cash-dee，后缀-d反映了Unix中命名持续在后台运行的 **守护(daemon)** 进程的惯例）。Rails的缓存存储必须实现一个不同存储能够被用到不同环境中的通用API——一个好的例子就是11.6节介绍过的依赖注入。

自我检查12.7.1. 我们提到过，即便页面布局包含了动态元素，例如登录的用户名，给**caches_action** 传递**:layout=>false**仍然提供了方法缓存的大部分好处。为什么不让**caches_action**方法也允许这个选项呢？

◇ 既然页面缓存是被展示层所处理，而不是逻辑层，一个页面缓存的击中意味着整个Rails都忽略了。展示层有整个页面的一个副本，但是只有逻辑层知道页面的哪部分来自于布局，以及哪部分来自于控制器的渲染方法。

12.8 避免滥用数据库查询

正如我们在2.4节所介绍的，数据库最终会限制横向扩展性——不是因为你用光了存储表格的空间，更可能是因为单独的计算机不能在保持响应性的同时维护每秒内必要的查询数量。当这发生时，你将会求助于诸如共享和复制的技巧，这些超出了本书的范围（请参看“深入学习”部分的建议）。

即便是对于一台计算机，数据库性能协调也是非常复杂的。广泛使用的开源数据库MySQL有许多的配置参数，并且大部分的数据库管理员（DBA）将会告诉你其中至少一半的参数对于得到好的性能是“关键的”。因此，我们关注于怎样让数据库的使用在一个PaaS提供者托管的限制范围内：Heroku、Amazon Web Services、Microsoft Azure，和其他由专业DBA负责基线调优的关系数据库提供者。许多有用的SaaS应用能够在这个尺度上创建——例如，所有的Pivotal Tracker[22]可以被一个单机数据库所满足。

一个减缓数据库压力的方法就是避免没必要的昂贵查询。没有经验的SaaS作者的两个常见错误是在有关联的时候：

(1) 如果遍历一个关联时使用了非必要的查询，会出现 **n+1查询** 问题。

(2) 当你的表格缺乏合适的 **索引** 来加速某些查询时，会出现 **全表扫描** 问题。

图12.10的1~17行展示了当遍历关联时所谓的n+1查询问题，也显示了为什么当代码涉及视图时，更易发生这个问题：没有方法让视图知道它正在造成的损伤。当然，贪婪加载你不需要的信息也同样不好，如图12.10的18~21行所示。`bullet`[23] gem可以帮助检测这两个问题。

另外一个需要避免的数据库滥用就是查询导致的 **全表扫描** 。考虑图12.10的第4行：在最坏情况下，数据库将不得不检查`moviegoers`表格的每一行来以找

代码文件：ch_operations/code/database_abuses.rb

```
1  # assumes class Moviegoer with has_many :movies, :through => :reviews
2
3  # in controller method:
4  @fans = Moviegoer.where("zip = ?", code) # table scan if no index!
5
6  # in view:
7  - @fans.each do |fan|
8    - fan.movies.each do |movie|
9      // BAD: each time thru this loop causes a new database query!
10     %p= movie.title
11
12 # better: eager loading of the association in controller.
13 # Rails automatically traverses the through-association between
14 # Moviegoers and Movies through Reviews
15 @fans = Moviegoer.where("zip = ?", code).includes(:movies)
16 # GOOD: preloading movies reviewed by fans avoids N queries in view.
17
18 # BAD: preload association but don't use it in view:
19 - @fans.each do |fan|
20   %p= @fan.name
21   // BAD: we never used the :movies that were preloaded!
```

图 12.10: 控制器方法中的查询（第4行）访问数据库层级来检索@fans行，但是行8∼10中的循环的每一个传递都会导致单独的数据库访问，导致对一个评论了n个电影的粉丝需要$n+1$条访问，行15，对比之下，执行了一个单一的 贪婪加载，也会检索所有的电影，它将会和行4一样快，因为小查询的大多数开销都是在执行数据库访问

代码文件：ch_operations/code/add_index.rb

```
1 class AddEmailIndexToMoviegoers < ActiveRecord::Migration
2   def up
3     add_index 'moviegoers', 'email', :unique => true
4     # :unique is optional - see text for important warning!
5     add_index 'moviegoers', 'zip'
6   end
7 end
```

图 12.11: 给一个列添加索引将会加速对这个列匹配的查询。如果你指定:unique，这个索引将会更快。如果你使用unique，你将会做出没有任何两个索引属性值将会相同的承诺。为了避免复制中的错误，可以联合使用unique和5.1节所描述的唯一性验证

到email列上的一个匹配，因此查询会在表格变大的时候变得越来越慢，一个n行的表格会需要$O(n)$的时间。解决方案是给moviegoers.email添加一个 **数据库索引**，如图12.11所示。索引是被数据库维护的分离的数据结构，当某个列被用作约束条件时，它通过对列值使用 **哈希** 等技术来允许对任何行的访问是常数时间。你能够对给定的一个表格有多个索引值，设置基于多个列的值的索引。除了在**where**或者**find_by_x** 中被显式命名的查询外，**外键** （关联对象) 也应该被索引。例如，在如图12.10所求的例子中，reviews表格中的**moviegoer_id**字段可能需要一个索引来加速被**fan.movies**暗指的查询。

当然，索引并不是免费的：每个索引都会占用和表格行数成比例的空间，并且由于当一个表格行被添加或者修改时，表格上的每一个索引都必须更新，对有大量索引的表格更新可能会很慢。然而，因为典型的SaaS应用是读取更多的行为，和其他诸如联机事务处理（OLTP）的数据库系统相比，SaaS应用相对简单，应用程序会在索引成为性能限制之前遇到许多其他的瓶颈。图12.12 显示了一个通过提供索引来大幅度提升性能的例子。

reviews表格的记录数：	2000	20000	200000	200000	
读100条，没有索引	0.94	1.33	5.28	创建1K条，没有索引	9.69
读100条，外键索引	0.57	0.63	0.65	创建1K条，全索引	11.30
性能	166%	212%	808%	性能	-17%

图 12.12: 对于一个Heroku框架中包含了1K个电影和每部电影超过100条评论的PostgreSQL共享式数据库，这个表格显示了索引的好处和代价。第一部分比较了没有索引情况下，和对reviews表格中的movie_id和moviegoer_id外键进行索引下读取100个评论的时间（单位是秒）。第二部分在没有索引，和对reviews列中的 每一个可能对 进行索引下创建1000个评论的时间。本图显示了即便是在病态的使用情况下，使用索引的代价都是轻微的

避免数据库滥用的小结：

- $n+1$查询问题是遍历一个1到 **n** 关联结果导致的$n+1$短查询，而不是一个单独的大型查询，能够通过正确地使用贪婪加载来解决。
- 查询中的全表扫描能够通过正确地使用数据库索引来避免，但是每个索引会占用空间且降低更新性能。一个好的开始点是给所有的外键列和**where**语句中经常涉及的全部查询列创建索引。

■ **详细阐述：SQL的EXPLAIN命令**

许多SQL数据库，包括MySQL和PostgreSQL（但不包括SQLite）提供了一个EXPLAIN命令用于描述查询计划：执行一个查询将会访问哪些表，以及这些表中的哪些索引可用来加速查询。不幸的是，EXPLAIN的输出格式在各个数据库中有所不同。从Rails3.2开始，比开发者在开发模式中指定的临界值更长时间的查询，EXPLAIN[24]会自动运行，查询计划将会被写到development.log。当前可以支持MySQL的query_reviewer[25] gem，对ActiveRecord产生的所有查询运行EXPLAIN，可以在开发模式下会把结果插入到每个页面最顶端的div中。

自我检查12.8.1. 一个数据库表格的索引通常加速____，以____和____为代价。

◇ 查询性能，空间，表的更新性能

12.9 安全：在应用程序中保护用户数据

Ronald Rivest（1947——）、Adi Shamir（1952——）和Leonard Adleman（1945——） 因为让公钥加密在实践中有用而获得2002年的图灵奖。在著名的RSA算法中，密钥对的安全性是基于分解大型整数和执行模幂运算的困难性，即寻找m使得$C = m^E \bmod N$。

> 我的反应是“祝贺你，Ron，它应该能用。”
>
> ——Len Adleman，对Ron Rivest的加密协议的反应，1977

因为安全是在计算自己的领域，因而从不缺少这方面的评论或学习资料。也许作为一个结果，安全专家已经把他们的建议总结为开发者可以遵循的三条原则，如下：

- **最少特权原则** 是说一个用户或者软件组件不应该被赋予更多的特权——也就是，除了执行指定任务所必需的，不能有更多的信息和资源访问权利。这类似于对于保密信息的“不该知道的秘密不问”原则。在Rails世界中有关这个原则的一个例子是你的Rails应用程序的Unix进程、数据库、和Web服务器（展示层）应该以最低特权运行，并且运行在一个不能够在文件系统中创建新文件的环境中。好的PaaS提供商，包括Heroku，都提供了以这种方式配置的部署环境。
- **防障缺省原则** 是说除非一个用户或者软件组织被明确地给予了某个对象的访问权，否则应该禁止访问这个对象。也就是说，默认下应该禁止访问。5.2节中介绍的用法遵循了这条原则。
- **心理可接受性原则** 是说防御机制不能够让应用程序比没有保护的时候更难使用。也就是说，用户接口需要使用起来简单，这样安全机制会自动地被用户遵循。

本节其余部分包括五个特定的与SaaS应用密切相关的安全隐患：加密保护数据、跨站点请求伪造、SQL注入和跨站脚本、禁止私有控制器方法调用以及自我拒绝服务。

加密保护数据。由于有实力的PaaS提供商在努力让他们的基础设施变得更安全，所有使用PaaS的开发者只需要将注意力放在通过好的编码规范可以解决的攻击上，SaaS上数据相关的攻击威胁着安全的三个基本元素中的一个或多个：隐私性、真实性和数据完整性。 **传输层安全** （TLS）和它的前辈 **安全套接层** （SSL）的目标就是使用被 **密文** （例如一个密码）所驱动的通信技术来对所有的HTTP数据流进行 **加密** ，使得通讯中的信息只会被通信的双方所理解。在这样一个安全连接上运行的HTTP被称为HTTPS。

和一个你之前从未访问过的网站建立一个共享的密码是一个有挑战的问题，其实际上的解决方案是归功于Ron Rivest、Adi Shamir和Len Adleman（ **RSA** 得名原因）的 **公钥加密（public key cryptography）** 。一个 **当事人** 或者交流的实体产生一个密匙对包含两个匹配的部分，其中一个保持公有（世界范围的任何人都可访问），另一个保持私有。

一个密钥对有两个重要的属性：

ssh-keygen 是本书配套软件中带有的标准密钥对生成工具。

(1) 一个使用私钥加密的信息只能通过公钥解密，反之亦然。

(2) 私钥不可以通过公钥推倒导出，反之亦然。

属性1提供了SSL的基础：如果你接受到了一个用Bob的公钥解密的信息，只有拥有Bob私钥的人可以创建它。一个变形就是数字签名：为了证明一个信息，Bob产生了一个单向的摘要信息（如果信息发生改变了，一个短的“指纹”将会生送改变），并且用他的私钥加密摘要来作为一种验证“我，Bob，担保被这个摘要所代表的信息。”

为了给他的rottenpotatoes.com网站提供SSL安全访问，Bob生成了一个包含了一个公有部分 **KU** 和一个私有部分 **KP** 的密钥对。他使用惯用的方法来证明其身份。比如政府颁发的身份证，到类似于VeriSign的 **认证机构(CA)** 去申请证书。CA使用它自己的私钥 **CP** 来签署一个它声明的 **SSL证书**，结果就是，“rottenpotatoes.com有公钥 **KU** 。”Bob在他自己的服务器上安装这一证书，且使他的SaaS栈接受SSL连接——在PaaS环境中通常不重要。最后，他通过在任何控制器中添加**force_ssl**[26]强制所有的方法使用SSL，来使得在他的Rails应用程序中允许SSL。或者使用**:only**或者**:except**过滤选项来限制哪个方法生效。

CA的公钥 **CU** 被装进了大多数的Web浏览器，因此当Alice的浏览器首次连接到https://rottenpotatoes.com和要求认证时，它能够验证CA的签名和从认证获得Bob的公钥 **KU** 。Alice的浏览器之后会选择一个随机的字符串作为密码，用 **KU** 加密，并且把它发送到rottenpotatoes.com，KP单独可以解密它。这个共享密码然后用更快的用于持续会话的 **对称密钥密码** 来加密HTTP流。在这个点上，任何通过HTTPS发送的内容对于防止偷听者都是相当有效的。同时，Alice的浏览器相信它正在说话的服务器是真正的RottenPotatoes服务器，因为只有一个拥有 **KP** 的服务器能完成密钥交换步骤。

认识到SSL的局限性是很重要的。特别地，服务器关于Alice的身份不知道任何信息，在传送到RottenPotatoes过程中，对Alice的数据除了私密性外不能够做任何担保。

跨站请求伪造(Cross-Site Requet Forgery, CSRF) 。一个CSRF攻击（有时候发音为“sea-surf”）包括欺骗用户的浏览器去浏览一个用户已经有一个有效的cookie的不同Web站点，并且伪装成用户在那个站点上执行一个违法的动作。例如，假设Alice最近登录了她的MyBank.com账户，因此她的浏览器有一个显示了她登录了MyBank.com的有效cookie，现在Alice访问一个聊天论坛，邪恶的Mallory在其中发布了一个如下嵌入“图片”的信息：

代码文件：ch_operations/code/csrf.html

```
<p>Here's a risque picture of me:
  <img src="http://mybank.com/transfer/mallory/5000">
</p>
```

当Alice查看了这个blog，或者她接受了一个嵌入了这个链接的邮件，她的浏览器将会试着通过RESTful URI“获得”这个图像，这样将会导致给Mallory的账号转入5000美元。Alice将会看到一个“坏掉的图像”图标而没有意识到已经收到损害。CSRF通常会结合跨站脚本来进行更高级的攻击。

Alice和Bob 是经常出现在安全场景中的 **最初原型的当事人**、还有 **force_ssl** 以一个前置过滤器来实现，导致从http:// **site/action** 到https://**site/action** 的立即重定向。

我们需要两个步骤来阻止这种攻击。第一步是确保RESTful动作的执行使用GET HTTP方法，而且GET方法没有产生副作用。一个诸如银行取款或完成一次购买动作应该通过一个POST来处理。这会让攻击者使用嵌入的像IMG的资源标签来传递“负载”更困难，因为浏览器总是使用GET处理这些资源标签。第二步就是插入一个基于当前会话的随机生成字符串到每个页面视图中，并且安排在每个表格中以隐藏的表格字段包括这个值。这个对于Alice的字符串在Bob看起来会不一样，因为他们的会话是不同的。当一个没带有正确随机字符串的表格被提交时，提交将会被拒绝。Rails自动化这种保护：你所要做的就是在每个这样的视图中渲染**csrf_meta_tags**，并且给任何可能进行这样一个提交的控制器添加**protect_from_forgery**。事实上，使用rails new来生成一个新的应用程序时，这些保护都相应地包括在了app/views/layouts/application.html.haml和app/controllers/application_controller.rb中。

SQL注入和跨站脚本。这两种攻击都利用了SaaS应用不安全地处理攻击者提

代码文件：ch_operations/code/sql_injection.rb

```
1 class MoviesController
2   def search
3     movies = Movie.where("name = '#{params[:title]}'") # UNSAFE!
4     # movies = Movie.where("name = ?", params[:title])   # safe
5   end
6 end
```

图 12.13: 对于一个SQL注入工具很脆弱的代码。去掉行4的注释，并且删除行3将会避免这个攻击。它使用了一个 prepared 语句 来允许ActiveRecord“净化”邪恶的输入

参数[:title]	SQL语句
Aladdin	SELECT "movies".* FROM "movies" WHERE (title='Aladdin')
'); DROP TABLE "movies"; --	SELECT "movies".* FROM "movies" WHERE (title=''); DROP TABLE "movies"; --

安全厂商赛门铁克(Symantec) 白皮书[27]中指出，XSS占据了2007年的安全漏洞中的80%多。

图 12.14: 如果Mallory在表格的第二行中输入文字作为一个电影标题，图12.13中行3变成一个删除整个表格的危险的SQL语句。（最后的--，SQL注释字符，避免任何DROP TABLE之后的SQL语句执行。）SQL注入攻击通常对早期的框架很成功，如PHP，因为其中的查询是通过开发者手动编写的

供的内容。在 **SQL注入** 中，Mallory输入了她希望的表单数据，该数据将被直接插入到一个应用程序执行的SQL查询语句中。图12.14显示了一个例子及其防御措施——使用一个预处理语句。在 **跨站脚本** 中，Mallory准备了一个会执行有害动作的Javascript代码片段，她的目标就是让RottenPotatoes渲染那个片段为HTML页面的一部分，引发脚本的执行。图12.15显示了一个良性的例子及其防御措施，真实的例子包括偷取Alice的有效cookie并传给Mallory的Javascript代码，她现在能够通过传递Alice的cookie作为她自己的来“劫持”Alice的会话。更糟糕的是，即便XSS攻击只是成功地从另一个站点读取页面内容，而非cookie，也是有风险的，因为这个页面内容可能包含为了阻止CSRF的**csrf_meta_tags**，因此XSS经常用来配合CSRF攻击。

JavaScript 是XSS问题最多的语言，但是任何在HTML页面中混入代码的技术都是脆弱的，包括ActiveX、VBScript和Flash。

禁止私人控制器方法调用。对于一个控制器去包括“敏感的”帮助方法是很常见的，这种方法它不是以实现最终用户的行为为目的，而只会来源于一个动作内部。应该对任何不是用户启动动作目标的控制器方法使用**保护**，并且检查rake路径以确保没有包括通配符的路径匹配一个非公有控制器动作。

自我拒绝服务 。一个恶意的拒绝服务攻击让一个服务器忙于无用的工作，阻

代码文件：ch_operations/code/xss_erb.html

```
1 <h2><%= movie.title %></h2>
2 <p>Released on <%= movie.release_date %>. Rated <%= movie.rating %>.</p>
```

代码文件：ch_operations/code/xss.html

```
1 <h2><script>alert("Danger!");</script></h2>
2 <p>Released on 1992-11-25 00:00:00 UTC. Rated G.</p>
```

代码文件：ch_operations/code/xss_defense.html

```
1 <h2>&lt;script&gt;alert("Danger!");&lt;/script&gt;</h2>
2 <p> Released on 1992-11-25 00:00:00 UTC. Rated G.</p>
```

图 12.15: 上图：一个使用Rails的内建eRB渲染而不是Haml的视图模板片段。中图：Mallory手动输入一个标题是<script>alert("Danger!");</script>字符串的新电影。当这个电影的Show方法被渲染时，“title”将会直接插入到HTML视图中，当Alice的浏览器渲染页面时，引起JavaScript代码被执行。下图：防御是净化任何将会篡改插入到HTML中的输入，高兴地，Haml的=操作符对这个自动进行，导致尖括号被正确转义为HTML的无效代码

止了正常用户的访问。如果你允许任意用户执行给服务器导致大量工作的动作，你能很疏忽地把自己开放给这种攻击了，例如允许上传大量的文件或者生成一个昂贵的报告。（上传文件也会产生其它危险，因此你应该把这种责任“外包”给其他服务，例如Progstr-Filer Heroku 添加项[28]。）一个防御是使用分离的后台任务，如用一个Heroku worker[29]进程去来把从主应用程序服务从长期运行的任务中解放出来。

当Amazon的1000个在售的Xbox 360仅仅只要100美元，而不是建议价格399美元时，由于数百万的用户点击“刷新”按钮来获得这个交易，在5分钟内站点的访问量就到达了一个很高高度。

攻击	Rails防御机制
窃听	安装SSL证书和在控制器中使用**force_ssl**而使用SSL来加密信息流（可选：**:only=>** 或者**:except=>** 来指定行为）
跨站请求伪造(CSRF)	在所有的视图中渲染**csrf_meta_tags**（例如，在主布局中包括它）和在**ApplicationController**指定**protect_from_forgery**
跨站脚本(XSS)	在渲染的同时使用Haml的=检查HTML
SQL注入	使用预先的查询，而不是直接把字符串插入到查询中
Mass assignment of sensitive attributes 敏感属性的大量赋值	使用**attr_protected**或者**attr_accessible**保护来自于用户赋值的敏感的属性（5.2节）
执行受保护的行为	使用**before_filter**来保护控制器中的敏感的公有方法；声明非公有的控制器方法为**private** or **protected**
自我拒绝服务，病理上缓慢的客户	使用独立的后台工作来执行长期运行的任务，而不是放在应用程序的服务器中

图 12.16: 一些常见的对SaaS应用的攻击及防范它们的Rails机制

对安全的最后一个警告是秩序井然。在SaaS开发者和做坏事的人之间的“军备竞赛”一直在进行，因此即便是一个仔细维护的站点也不是100%安全的。除了保护对用户数据的攻击，你也应该仔细处理敏感数据。不要以明文存储密码；加密存储它们，或更好地，使用第5.2节所描述的第三方认证来避免有关[30] 密码[31] 偷窃的[32] 令人尴尬的[33] 事件[34]，从不要 **考虑** 存储信用卡号码，即便是加密过的。支付卡行业协会每年给这样做的网站强制征收数千美元的审计开销（为了防止信用[35] 卡[36] 欺骗[37]）。如果你的代码曾经仅仅是操作了信用卡号码，而没有存储它，这个的危险性也只是稍微轻点。作为代替，你可以把这个责任交给专门从事满足这种高风险操作的网站，如PayPal[38]或者Stripe[39] 。

保护用户数据的小结：

- 遵循最小特权原则，防障缺省原则及心理可接受原则能够产生更安全的系统。
- SSL和TLS保持经HTTP连接传输数据的私密性，但是没有提供其他隐私保证。他们也保证浏览的服务器身份（除非最初验证服务器身份的认证中心失效了），但是反之没有。
- 部署在一个架构良好的PaaS开发者应该主要关注通过良好编程规范能够阻止的攻击。图12.16总结了一些关于SaaS的常见攻击和解决它们的Rails机制。
- 除部署应用程序级别的防御外，特别敏感的用户数据应该或加密存储或不存储，而通过外包给专门的服务来处理。

自我检查12.9.1. 对或错：如果一个站点有一个有效的SSL认证，跨站请求攻击（CSRF）和SQL注入攻击很难对它起作用。
◇ 错。HTTP通道的安全和这两种攻击都不相关。CSRF只和一个网站错误地接受一个有效但是来源于别处的cookie有关。SQL注入只取决于SaaS服务器代码将用户输入的字符不安全地插入到到一个SQL查询中。

自我检查12.9.2. 为什么CSRF攻击不能够通过检查一个HTTP请求的包头的`Referer`域来阻止？
◇ HTTP包头能很容易地伪造。

12.10 计划文档的视角

非功能性需求比增加新功能更加重要，因为违背会导致损失数百万美元，或者损失数百万用户，或者两者都损失。例如，Amazon.com在2012年第四季度的销售是233亿美元，因此Amazon因为一个小时的宕机的收入损失平均而言有1000万美元。同一年，一起Nebraska学生信息系统的破坏[40]泄露了自1985年以来申请Nebraska大学的人的社保号，估计有650000人。如果用户不信任一个SaaS应用，无论有什么样的好的功能，他们都会停止使用这一应用。

性能。性能在传统软件工程中不是一个受关注的话题。部分是因为它是坏的习惯的借口，部分是因为它被其他地方很好地关注了。性能可以作为非功能性需求的一部分，然后随后通过接收测试来确保性能需求能够被满足。

发行管理。计划文档过程通常生产的软件产品有主要发行版（升级性发行）和次要发行版（维护性发行）。以Rails为例，版本3.2.12中的最后一个数字是一个维护性发行，中间的数字是升级性发行，第一个数字是改变了API，以至于应用程序需要被再次导入到这个版本的大型变换。一个发行包括了任何事：代码、配置文件、任何数据以及文档。发行管理包括打包用于发行的数据，怎样分发信息，以及文档化任何东西，这样你能够知道发行中都有什么，以及怎样再次做到，这样不得不进行下一次发行的时候，非常容易去改变。发行管理在计划文档过程中被认为是一种配置管理，正如我们在10.7节所评论的。

可靠性。在我们的工具包中最大的工具是让一个系统的依赖是冗余的。通过有更多的硬件，而不是程序运行和存储数据所需要的最小值，即便是一个组件的失败，系统继续存在潜在的失败风险。因为所有的硬件都有一个非零的失败概率，一

个冗余指导方针是确保没有 **单一失败点**，因为它可能是系统的一个致命缺陷点。通常情况下，更多的冗余就有更小的失败机会。正因为高冗余的系统是非常昂贵的，和用户有次个正式的商洽，以正确了解应用程序对于可靠性的要求，是非常重要的。

可依赖性是整体的，包括软件、操作者以及硬件。无论硬件的可依赖程度如何，软件中的错误和操作者的失误能导致减少 **平均失效时间** （**MTTF**）的中断。由于依赖性是这条链中最薄弱环节的一个功能，较之于购买更高冗余的硬件来运行应用程序，训练操作者怎样使用应用程序和减少软件中的错误会更有效。由于“是人就会犯错误”，系统既应该既包括硬件失败的防护，也包括容忍和组织操作者失误的防护。

计划文档过程的一个基本假设就是，通过磨练其软件开发过程，一个组织能够使软件产品更可预测和更具重复性，这也会产生更可靠的软件。因此，组织者通常会记录下任何他们从项目中可得到的事情，以便学习能够做什么以提高其开发。例如，如果公司有一套流程，就会授予ISO9001标准，一个看其过程是否被遵循的方法，以及记录每个项目的结果，为了在过程中有所提高。很奇怪地，标准的验收不是关于结果代码的质量，而是关于开发过程。

最后，和性能一样，可靠性可以被测量。我们可以通过增长MTTF，或者让重启更快以缩短MTTR时间。如下面的方程所示：

$$\text{unavailability} \approx \frac{\text{MTTR}}{\text{MTTF}} \tag{12.1}$$

由于很难去测量MTTF中的提高，因为需要花费很长时间来记录失败，我们可以简单地使用MTTR。我们仅需让一台计算机崩溃，然后看让应用程序重启需要多长时间。我们可以提升能测量到的指标。因此，试着改善MTTR，而不是改善MTTF，会更有效，因为测量过程更容易。然而它们不是相互排斥的，开发者能够试着从这两个方面来增加可依赖性。

安全。虽然可靠性可以依靠概率来计算可访问性——如果存储系统在设计时没有省略可依靠性，很多的磁盘不太可能同时失效——这不是安全问题。因此有一个对手探查你设计中的薄弱角落，然后就可以利用它们闯入系统。常见的漏洞和风险数据库[41]列出了常见的攻击，来帮助开发者理解安全挑战的困难。

很幸运地，防御式编程不仅让系统更健壮，也让系统更加安全。例如，在 **缓冲区溢出攻击** 中，攻击者给一个缓冲区发送太多的数据，以让隐藏在数据中的代码改写附近的内存。检测输入确保用户没有发送太多的数据可以防止这样的攻击。同样地， **算术溢出攻击** 的原理可能是提供一个未预期的大数据，以至于当两个数据相加时，它会因为32位算术溢出显得很小。检查输入的值或者捕获异常也许能阻止这种攻击。因为现今的计算机都有多个处理器（“多核”），一个逐渐增加的常见攻击是 **数据竞赛攻击**，程序对于输入没有确定性的行为。这种并行编程缺陷更难检测和解决。

测试比安全更具挑战性，一个方法就是使用 **老虎团队（tiger team）** 来假扮攻击者进行w[Penetration test]触探检测，并且将攻击的结果报告给开发人员。

小结：
给出了让用户保持信任的重要性，非功能性特征会比功能性特征更重要，特别是对于SaaS应用。

- 计划文档过程很少关注性能，除了作为随后在综合测试计划被验证的系统需求书的一个潜在条目之外。
- 发行，被认为是配置管理的一部分，在计划文档过程中是一个很重要的事件。一个发行包括了在那个时间有关项目的一切，包括发行是怎样做的文档，以及代码、配置文件、数据和产品文档。
- 冗余是可靠系统的关键，高可访问性系统的目标就是没有单一缺陷点。**无故障运行时间** 是整个系统的一个功能，包括硬件、操作者和软件。另一个提供可访问性但比测量MTTF更容易的方法是关注于减少 **平均维修时间（MTTR）**。
- 与可靠性分析中的失败的概率基础不同，安全总是是基于一个智能的对手，它有目的地探查未预期的事件，比如缓冲区溢出。

自我检查12.10.1. 除了缓冲区溢出、算术溢出、和数据竞争外，列出一个违背以上列出的三个安全原则之一而导致安全问题的潜在错误。
◇ 一个例子就是不合适的初始化，它违背了预设安全模式。

12.11 谬论和陷阱

谬论：所有对持续集成测试中很少出现的情况的努力都是麻烦高于价值。

每天100万的访问量，百万分之一的“罕见”事件在统计上每天都会发生。每天100万的访问量是Slashdot在2010年的规模，每天80亿（8×10^9）的访问量是Facebook2010年的规模[42]，每天将会有8000个“百万分之一概率”的事件发生。这就是为什么像Google这样的公司的代码评审会关注角落事例：对于大规模应用，概率上不可能的事件每天都会发生（Brewer 2012）。处理错误的代码提供的额外抗性，能够让你晚上睡得更踏实。

谬论：应用程序仍在开发中，我们可以忽略性能。

Knuth的确说过过早的优化是万恶之源“……大约所有时间的97%”，但是这个引言还有：“然而我们不应该放弃在那关键的3%的机会。”盲目忽视像缺少索引或者不必要重复的查询设计问题，同在早期阶段盲目地关注性能一样糟糕。避免真正的让人震惊的性能错误，你将会在两个极端之间找到一条很幸福的路。

陷阱：认为你不需要担心性能，因为使用云计算的三级应用程序将会“自动地”扩展。

这不是一个真正的陷阱，如果你正在使用架构良好的PaaS，从某种意义上说这是真的。然而，如果应用程序比PaaS“成长得更快”，关于扩展性的基本问题和负载平衡问题将会交到你手上。换言之，使用PaaS，你并不能幸免于不理解和避免这些问

题，但是你自己可以临时地免于用自己的解决方案来处理它们。当从头开始建立系统时，用不了多久，你就会欣赏PaaS的价值。

谬论：因为计算机变得如此快速和便宜，处理器时间是免费的。

在第1章，我们主张将今天额外的计算机能力用于更有效的工具和语言。然而，很容易把这种观点诠释得过火。在2008年，性能工程师Nicole Sullivan报告了由不同大型SaaS提供者关于额外延迟如何影响其站点的实验。图12.17清晰地显示了当额外的处理器时间变成对终端用户额外的延迟（因此减少了响应性），处理器时间并不是免费的。

行为	增添的延迟	影响的结果
Amazon.com的页面视图	100ms	销售额下降1%
Yahoo.com的页面视图	400ms	全页面流量有5∼9% 的下降
Google.com的搜索结果	500ms	搜索行为减少了20%
Bing.com的搜索结果	2000ms	从每个用户获得的收入减少了4.3%

图 12.17: 各种大型SaaS应用在用户交互上添加的延时的测量结果，来自于Yahoo的性能工程师Nicole的"快速设计网站"上的演示[45]，和Velocity2009会议上[46]Google的Jake和Amazon的Eric的一个结合的演示

陷阱：没有测量就进行优化。

一些用户很惊奇当他们的某个客户应用程序变慢时，Heroku没有自动地添加额外的Web服务器能力（van Hardenberg 2012）。原因是没有指定和测量应用程序，你不知道 **为什么** 变慢了，风险就是添加Web服务器将会让问题变得更坏。例如，如果应用程序存在如缺少索引或者$n+1$查询的数据库问题，或者如果依赖于单独的像Google Map这样临时会变慢的服务，那么添加服务器来接受 **更多的** 用户请求只会让事情更坏。没有测量，你就不知道怎样修复。

陷阱：滥用持续部署，导致毫不相关的积累。

正如我们已经见过的，不断进化的应用程序会增长到这样一个点，此时改变设计或者改变架构将是添加新功能最干净的方式。由于持续部署关注于小的增量步骤，并且告诉我们避免担心并不立即需要的任何功能，在更多的代码固定到一个过时的设计时，应用程序有增加许多不相干的积累的潜在风险。逐渐出现的代码味道（第9章）通常是这个陷阱的早期症状，当气味开始增加时，可以通过周期性地设计评审和架构评审来避免，

陷阱：在命名或者失效逻辑中的错误，导致错误而未察觉地缓存动作。

正如我们指出的，对于任何类型的缓存，两个你必须处理的问题是命名和失效。如果你无意中对不同的物体重用相同的名称——例如，一个根据登录的用户不同而提供不同内容的非RESTful的动作，而总是使用相同的URI命名——那么一个缓存对象将会以不应该的方式被错误地提供。如果清理器不能捕获一组缓存的对象变得失效的所有条件，用户就会看到没有反映最近变换结果的过期数据，例如一个没有包含最近添加的电影的电影列表。单元测试应该包括这些问题（"缓存系统应该在新电影被添加的时候，立即在主页列表上显示新电影"）。遵循Rails的缓存指导[47]中的步骤在测试和开发环境中打开缓存，为了简化调试缓存默认是关闭的。

代码文件：ch_operations/code/soa_timeout.rb

```
require 'timeout'
# call external service, but abort if no answer in 3 seconds:
Timeout::timeout(3.0) do
  begin
    # potentially slow operation here
  rescue Timeout::Error
    # what to do if timeout occurs
  end
end
```

图 12.18: 如果外部服务是缓慢的，对一个外部服务使用带timeout的调用，能保护应用程序不会变得缓慢

陷阱：在一个SOA中缓慢的外部服务器会对你自己的应用程序产生有害后果。

如果应用程序和一个SOA中的外部服务器通信，你应该对这些外部服务器会很缓慢或是没有响应做好准备。最简单的情况是处理一个没有响应的服务器，因为一个被拒绝的HTTP连接会导致一个你能捕获的Ruby异常。困难的情况是一个外部服务在工作但是非常慢：默认情况下，对服务的调用会被阻止（等到操作完成或者TCP的“慢超时”过期，这能占用3分钟的时间），也会使应用程序慢下来。更坏的是，因为大多数Rails的前端（`thin`、`webrick`、`mongrel`）是单线程的，如果你运行N个这样的前端（Heroku的术语叫“dynos”），它只会采取N个同步请求来完全处理应用程序。解决方案是使用Ruby的超时**timeout**库来“保护”调用，如图12.18所示。

谬论：我的应用程序安全，因为它运行在一个使用防火墙和HTTPS的安全平台上。

没有所谓的“安全平台”。当然有不安全的平台，但是没有平台本身可以确保应用程序的安全。安全是需要一个系统和持续关注的问题：每个系统有一个薄弱的环节，随着新漏洞和软件缺陷被发现，最薄弱的环节可能从系统的一部分移动到其他部分。入侵者和合法开发者之间的“武器竞赛”使得去使用专业架构的PaaS基础设施变得越来越有吸引力，这样你就可以专注于应用程序代码的安全。

谬论：我的应用程序不是攻击者的一个目标，因为它服务特定用户，处于很小的规模，且没有存储有价值的信息。

恶意攻击者不一定在意应用程序；他们可能会把它作为一个载体来进行进一步的行动。例如，如果应用程序接受博客风格的评论，它会变成博客垃圾的目标，自动代理（机器人）张贴含有希望用户去访问的链接的垃圾评论，或者去购买某些东西或者导致一个恶意软件被安装。如果应用程序对SQL注入攻击是开放的，那么这样一个攻击的动机可能是影响你视图展示相关的代码，用来导入一个跨站脚本攻击，例如导致恶意软件被下载到一个毫无警觉的用户的机器上，即使没有恶意攻击者，如果应用程序的任何方面因为Slashdot或者Digg突然变得很流行，你将会突然被这些流量所淹没。教训就是：**如果应用程序是公开部署的，那么它就是一个被攻击的目标。**

12.12 结束语：性能、可靠性、安全和有漏洞的抽象层

性能、可靠性和安全性是必须不断评审的系统问题，而不是一次性就可以解决然后放在一边的问题，此外，在12.8节描述的数据库滥用，表

明Rails和ActiveRecord和大多数抽象一样是 **有漏洞的**：他们为了提升生产力试图隐藏实现细节，但是有时出于安全和性能的担忧，你必须得像一个开发者那样一样去理解这些抽象层是怎样工作的。例如，$n+1$选择问题从查看Rails代码不是很显然，例如提供的**:include**对于联合问题的解决方案，使用**attr_accessible**或**attr_protected**去保护被一个恶意用户大量分配的敏感属性都不是很显然的。

在第4章我们强调了保持开发和生产环境尽可能相同的重要性。这仍然是好的建议，但显然如果生产环境涉及多个服务器和一个巨大的数据库，在开发环境保存同样设置的要求不太实际。事实上，在这本书中我们开始开发应用程序时使用数据库SQLite3，但用PostgreSQL部署在Heroku。考虑到这些差异，那么开发中的性能提升（减少查询数量、添加索引、添加缓存）仍会在实际生产中起作用吗？绝对的。Heroku和其他PaaS站点在调整其数据库的基线性能和软件堆栈上做得很出色。但是没有什么调整可以解决如$n+1$查询问题，或没有部署缓存来减轻数据库负载这样的低效查询策略。

12.13 深入学习

鉴于篇幅有限，我们专注于每个SaaS开发人员应该知道的操作，即使给定了PaaS的可访问性。一个优秀、更详细、且关注SaaS的具体挑战，包含真实用户故事的书是Michael Nygard的 **Release It!**（Nygard 2007），该书更多关注“意想不到的成功”的问题（突然的流激增、稳定性问题等等），而不是抵制恶意攻击。

我们的监控的例子基于许多请求延迟的聚合指标，一个对比的方法是请求跟踪，它结合了使用详细的指标聚合和诊断缓慢请求。在开发模式下的Rails logs中，你可以看到一个简化版的请求跟踪，默认情况下将报告数据库查询时间、控制器动作时间和每个请求的渲染时间。真正的请求跟踪是多细粒度，跟踪一个穿过每层中的软件组件的请求并且在这一路上记下它的时间戳。运行大型站点的公司通常构建他们自己的请求跟踪，例如Twitter的“Big Brother Bird”和Google的Dapper，在最近的一篇文章中（Barroso and Dean 2012），Google的两位架构师讨论了怎样使用请求跟踪识别障碍，以保持Google大量的并行系统的高响应性。James Hamilton，一位工程师和Amazon网络服务架构师，他之前花费了许多年进行架构和调优Microsoft SQL Server，写了大量关于延迟开销[48]的优秀博客[49]，从许多一直在关注这个话题的行业参与者收集测量结果。

虽然我们关注对性能问题的监控，监控也能帮助你理解你的客户行为：

- 点击流——你的用户访问页面的最流行序列是什么？
- 考虑时间/停留时间——一个典型用户在给定的页面上会停留多长时间？
- 丢弃——如果你的站点包括一个明确定义了终止的流，例如买商品，用户“丢弃”这个流而不是完成它的比例有多少？以及它们进行了多远？

Google Analytics提供了免费的基本的分析作为服务：你在你站点的每一个页面里面嵌入一小段Javascript片段（例如，通过把它嵌入到默认的布局模板），它将会在页面每次加载时发送Google Analytics信息。为了帮助你使用这个信息，Google的“速度是一个特征”[50]站点链接了一个令人惊奇和全面的文章集合，关于所有不同的可加快SaaS应用的方法，包括许多减少页面总体规模的优化，哪个浏览器可以提高渲染速度。New Relic所维护的RailsLab博客[51]也包含了调优Rails应用程序的最好建议和技巧，包括在Rails调试上的截屏和怎样在开发模式下使用New Relic来进行

性能分析[52]。但是请注意，一些特定的代码示例不再有效，尤其是关于缓存，因为Rails2和Rails3之间的改变。

理解在部署和操作间发生了什么（尤其是自动化部署）是进行复制性能问题调试的先决条件。今天大多数的SaaS应用，包括那些托管在Windows服务器，运行在一个基于最初的Unix进程模型和输入/输出环境，所以理解这个环境对于调优任何重要的性能问题是至关重要的。**《Unix编程环境》**（Kernighan and Pike 1984)，由Unix的一个创建者合著，提供了一个有关Unix架构和哲学的丰富内容，通过做来学的旅程（使用C语言!）

分片 和 **复制** 是扩展一个数据库的强大技术，它预先需要大量的设计思考。虽然有Rails gem帮助这两个问题，这些技术通常需要数据库级别的配置变化，但许多PaaS提供商不支持它。随着“NoSQL”数据库的突然出现，分片和复制变得尤为重要，“NoSQL”数据库通过牺牲独立于SQL的良好表达数据格式来换取更好的可扩展性。**NoSQL生态系统**，Adam Marcus的 **《开源应用程序的体系结构》** Marcus 2012中的一章，对这个主题有很好的讨论。

安全是一个非常广泛的主题，我们的目标是通过使用内置的机制来帮助你避免基本的错误，这些机制阻止对应用程序和客户的数据的常见攻击。当然，攻击者不能危害应用程序的内部数据，但仍可能通过攻击应用程序所依赖的基础设施来造成损害。分布的 **拒绝服务**（DDoS）给一个站点带来巨大的流量，以至于站点不能对正常用户进行响应。一个恶意客户端可以让应用服务器或Web服务器“在线上挂起”，使得用户输出缓慢，除非Web服务器（展示层）有内置的超时。**DNS欺诈** 是当一个浏览器查询一个主机名时，尝试通过提供一个不正确的IP引导你到一个伪装站点，通常是结合了伪造证书证明服务器的身份的 **中间人攻击**。这个骗子网站看起来和实际行为像真正的网站，但是会收集来自用户的敏感信息。（在2011年9月，通过攻陷DigiNotar 黑客伪装成美国中央情报局，英国军情六处以及以色列摩萨德网站[53]，DigiNotar公司给这些真的签署了原始的证书，导致了DigiNotar的破产。）即使成熟的软件如安全的Shell和Apache都是脆弱的：美国国家漏洞数据库[54]列出仅在2012年3月和5月间的10个Apache中的新的安全相关错误，它们中的两个是“高危害”的，意味着它们允许攻击者控制整个服务器。虽然如此，尽管偶尔有漏洞[55]，架构良好的PaaS站点更可能雇佣有经验的专业系统管理员，他们熟悉最新的技术来避免这样的漏洞，让他们成为你的SaaS应用最好的第一道防线。在Ruby on Rails站点上的基本Rails安全手册[56]评论很多Rails特性旨在阻止常见的SaaS应用攻击。来自于CodeClimate的文章[57]列出了许多在Rails应用程序中的重要安全陷阱。

最后，在某些时候会发生不可思议的事情：你的生产系统将进入一个状态，此时部分或全部用户都收不到服务。无论应用程序崩溃或者是“挂起”（无法继续运行），这两个条件从商业角度看起来一样，因为这个应用程序没有产生收入。在这个场景中，首要任务是恢复服务，可能需要重新启动服务器或做其他事后查找状态操作，就是你首先想检查确定是什么导致这个问题。详细的日志可以起作用，因为日志提供了在服务恢复之后，你可以详细检查的半永久性的记录。

《有用事物的进化》（Petroski 1994）中，在论证了许多成功产品的设计主要由早期设计中的导致重新设计的失败所影响，工程师Henry Petroski建议改变格言“形式追随功能”（最初来源于建筑领域）为“形式服从失败”。有关好的设计的一个例子，请阅读Netflix的关于他们的设计是如何挽救2011年的Amazon Web访问中断事件的技术博客文章[58]，这个中断事件影响了许多依赖AWS的网站。

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

L. Barroso and J. Dean. The tail at scale: Tolerating variability in large-scale online services. **Communications of the ACM** , 2012.

E. Brewer. Personal communication, May 2012.

S. Hansma. Go fast and don't break things: Ensuring quality in the cloud. In **Workshop on High Performance Transaction Systems (HPTS 2011)** , Asilomar, CA, Oct 2011. Summarized in Conference Reports column of USENIX ;login 37(1), February 2012.

B. W. Kernighan and R. Pike. **Unix Programming Environment (Prentice-Hall Software Series)** . Prentice Hall Ptr, 1984. ISBN 013937681X. URL `http://www.amazon.com/Programming-Environment-Prentice-Hall-Software-Series/dp/013937681X`.

A. Marcus. The NoSQL ecosystem. In A. Brown, editor, **The Architecture of Open Source Applications** . lulu.com, 2012. ISBN 1257638017. URL `http://www.aosabook.org/en/nosql.html`.

M. T. Nygard. **Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)** . Pragmatic Bookshelf, 2007. ISBN 0978739213. URL `http://www.amazon.com/Release-Production-Ready-Software-Pragmatic-Programmers/dp/0978739213`.

H. Petroski. **The Evolution of Useful Things: How Everyday Artifacts-From Forks and Pins to Paper Clips and Zippers-Came to be as They are** . Vintage, 1994. ISBN 0679740392. URL `http://www.amazon.com/Evolution-Useful-Things-Artifacts-Zippers-Came/dp/0679740392`.

P. van Hardenberg. Personal communication, April 2012.

注释

[1] `http://www.opensourcerails.com/`
[2] `http://github.com/ucberkeley/researchmatch`
[3] `http://github.com/vinsonchuong/meetinglibs`
[4] `http://apdex.org`
[5] `http://code.google.com/speed`
[6] `http://rightscale.com`
[7] `http://github.com/capistrano`
[8] `http://travis-ci.org`
[9] `http://saucelabs.com`
[10] `https://github.com/jamesgolick/rollout`
[11] `http://www.google.com/apps/intl/en/business/details.html`
[12] `http://newrelic.com`
[13] `http://pingdom.com`
[14] `http://sitescope.com`
[15] `http://analytics.google.com`
[16] `http://newrelic.com`
[17] `http://scoutapp.com`
[18] `http://exceptional.io`
[19] `http://airbrake.io`
[20] `http://godrb.com`
[21] `http://www.hpl.hp.com/research/linux/httperf`
[22] `http://pivotaltracker.com`

[23]https://github.com/flyerhzm/bullet
[24]http://weblog.rubyonrails.org/2011/12/6/what-s-new-in-edge-rails-explain
[25]http://github.com/nesquena/query_reviewer
[26]http://apidock.com/rails/ActionController/ForceSSL/ClassMethods/force_ssl
[27]http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf
[28]https://devcenter.heroku.com/articles/progstr-filer
[29]https://devcenter.heroku.com/articles/background-jobs-queueing
[30]http://www.zdnet.com/blog/btl/26000-email-addresses-and-passwords-leaked-check-this-list-to-see-if-youre-included/50424
[31]http://www.neowin.net/news/main/09/10/05/thousands-of-hotmail-passwords-leaked-online
[32]http://hothardware.com/News/55000-Twitter-Accounts-Hacked-You-Should-Probably-Change-Your-Password/
[33]http://www.huffingtonpost.com/2012/06/07/eharmony-passwords-leaked-linkedin_n_1577175.html
[34]http://www.huffingtonpost.co.uk/2012/06/08/lastfm-hit-by-password-leak_n_1580012.html?ref=uk
[35]http://www.businessweek.com/technology/content/jul2009/tc2009076_891369.htm
[36]http://www.msnbc.msn.com/id/17853440/#.T9JsqxztEmY
[37]http://redtape.msnbc.msn.com/_news/2012/03/30/10940640-global-payments-under-15-million-account-numbers-hacked?lite
[38]http://paypal.com
[39]http://stripe.com
[40]http://nebraska.edu/security
[41]http://cvedetails.com/
[42]http://royal.pingdom.com/2010/01/05/facebook-twitter-myspace-page-views
[43]http://www.slideshare.net/stubbornella/designing-fast-websites-presentation
[44]http://velocityconf.com/velocity2009/public/schedule/detail/8523
[45]http://www.slideshare.net/stubbornella/designing-fast-websites-presentation
[46]http://velocityconf.com/velocity2009/public/schedule/detail/8523
[47]http://guides.rubyonrails.org/caching_with_rails.html
[48]http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx
[49]http://perspectives.mvdirona.com
[50]http://code.google.com/speed
[51]http://railslab.newrelic.com
[52]http://newrelic.com/demos/developer-mode.html
[53]http://catless.ncl.ac.uk/Risks/26.56.html#subj6
[54]http://nvd.nist.gov
[55]http://daverecycles.com/post/2858880862/heroku-hacked-dissecting-herokus-critical-security
[56]http://guides.rubyonrails.org/security.html
[57]http://blog.codeclimate.com/blog/2013/03/27/rails-insecure-defaults
[58]http://techblog.netflix.com/2011/04

12.14 建议的项目

对于这些练习中的许多，你将会发现对给定的模型类型，创建一个特别多的随机生成实例的rake任务将会非常有帮助。你也会希望让应用程序在Heroku上有一个“测试”副本，这样你能够在不修改生产数据库的情况下部署测试数据库进行实验。

SaaS性能和扩展性：

练习12.1. 用500部电影（随机生成就很好）作为RottenPotatoes的测试数据库的种子。使用New Relic来进行Heroku上的部署性能分析。对被**index**视图所使用**movie**部分添加缓存片段，并且在此对应用程序进行性能分析。一旦缓存开始起作用，你所观察的**index**视图的响应性提高了多少?

练习12.2. 继续先前的实验，在使用和没使用缓存的情况下，使用`httperf`来比较应用程序的一个副本上的**index** 动作的吞吐量。（在Heroku上，任何应用程序的一个免费账户默认接受一个“dyno”或者一个单元的并行任务，这样的请求是被连续执行的。）

练习12.3. 使用外部的监控来分析一个软件设计，例如从一个有意义的外部质量属性，如功能性，性能或者可访问性。注意：边缘图标表明一个项目来自于ACM/IEEE 2013软件工程标准（ACM 2013）。

练习12.4. 继续先前的练习，给索引视图添加缓存动作，这样如果没有排序或者过滤选项被指定，**index**动作仅仅返回所有电影。比较有或没有缓存动作的性能和吞吐量。以一个表格总结这三个练习的结果。

发行和特征标记：

练习12.5. 调查执行一个“原子”机制更新和在12.4节中描述的迁移的可访问性开销。为了做这个，取$N = 2^{10}, 2^{12}, 2^{14}$来重复下面的步骤序列：

(1) 用N个随机生成带有随机评分的电影来作为测试数据库的种子。

(2) 在测试数据库上运行一个改变`movies`表格中`rating` 列从一个string为integer的迁移。（1=G，2=PG，等等）。

(3) 注意时间是通过`rake db:migrate`报告。

假设对于一个30天的测试窗口，正常运行时间目标是99.9%。量化在不让服务下线的情况下，采取上述迁移的可访问性效果。

练习12.6. 列出回归测试过程的大纲和它在发行管理中的角色。

可靠性：

练习12.7. 列出能够应用到计划文档生命周期中来减少错误的方法。

练习12.8. 在5.2节中，通过添加身份验证提供者名称和一个特定的提供者UID到**Moviegoer**模型中，我们整合第三方认证到RottenPotatoes。

现在我们想更进一步，并且允许同样的moviegoer用任何一个服务验证提供者来登入到同样的账号。也就是，如果Alice有一个Twitter账号和一个Facebook账号，她应该能用任一个ID登录到同一个RottenPotatoes账号。

(1) 描述为了支持这个机制，对已存在的模型和表格的改变。

(2) 描述一系列部署和迁移能够使用特征标记在任何应用程序都不下线条件来实现这个新机制。

安全：

练习12.9. 2012年7月，Wired杂志的ThreatLevel专栏[1]报告有453000位雅虎语音用户的密码被黑客所偷窃。这个攻击者在一个在线的日志中说，密码是以明文存储在雅虎的服务器中，他们使用一个SQL注入攻击获得了它们。讨论这个事件。

练习12.10. 描述一般情况下的安全编码和防御编码实践。

对于接下来的项目，你将需要鉴别你将会检视的一个工作着的遗留软件系统。作为建议，你能够使用Open Source Rails上的开源Rails项目，或者你能选择使用本书的学生所创建的两个项目中的一个：ResearchMatch[2]，它帮助匹配学生和他们所在学校的研究机会和VisitDay[3]，它帮助组织学生和教员之间的见面。

练习12.11. 从使用一个关系数据库管理系统的角度，描述安全编码和防御编码实践。你正在检查的系统遵循了这些实践吗?

练习12.12. 从构建一个使用如Rails的框架的SaaS应用的角度，描述一些特殊的防御编码实践。查看Rails安全指导和一些如基本的Rails安全指导[4]的博客，来自于CodeClimate的文章[5]，和使用Google查找在Rails SaaS应用中最新的安全事故，哪些引起失败的安全问题遵循了这些实践?

练习12.13. 重写一个简单的程序来移除常见的漏洞，例如缓冲区溢出、整数溢出以及竞争条件。

练习12.14. 强调和应用最小特权原则和自动防御缺省原则，这些原则是怎样应用到我们的RottenPotatoes应用程序的?

13 后记

预测未来的最佳方式就是去创造它。

——Alan Kay

Alan Kay（1940——）因为开拓了当代的面向对象编程语言的根源上的很多思想而获得了2003年的图灵奖。他带领他的团队开发了Smalltalk语言，Ruby就继承了它的面向对象的方法。他还发明了"Dynabook"概念，也就是今天的笔记本电脑和平板电脑的前身，他认为这是教授编程的教学平台。

13.1 SaaS、SOA、Ruby和Rails的视角

在这本书里你主要是一个成功的分布式架构（Web、SOA）和框架（Rails）用户。作为一个成功的软件工程师，你可能需要创建这样的框架或者扩展现有的框架。注意这些框架成功的原则会对你有帮助。

在第2章里我们指出，通过选择构建SaaS，你需要做一些建筑性的选择。在构建不同类型的系统中，其他的选择可能是合适的，但是因为作用域的原因，我们一直都集中在这一组选择。值得指出的是，一些底层SaaS和SOA的重要建筑性原则也使用于其它架构，并且和Jim Gray在第3章前所说的那样，伟大的想法需要时间来变得成熟。

Rails本身随着软件行业向使用敏捷开发和云计算来部署的SaaS的转变而快速发展。今天几乎所有的传统的购买和安装程序都作为服务提供，包括PC领导者像Office（见Office 365）和TurboTax（见Turbotax在线）。Rails等工具使得敏捷比之前的软件开发方法更加容易使用和实践。值得注意的是，不仅软件的未来被彻底改变了，软件开发现在也更加容易学习！

13.2 回顾过往

图13.1，首先出现在第1章，显示了三个"皇冠上的宝石"，是这本书中的材料的基础。为了理解这种良性三角你需要学习很多新的概念；图13.2列出了将近120个概念和名词，仅仅是针对前三章！

每一对"宝石"形成了协同债券，互相支持，如图13.1所示。特别的是，工具和相关的Rails服务使它更容易遵循敏捷生命周期。图13.3显示了我们多次重复的敏捷迭代，但是这次它给出的是我们在这本书里使用的工具和服务。这些14种工具和服务支持遵循敏捷生命周期 和 开发SaaS应用。同样，图13.4总结了计划和文档生命周期的各个阶段和它们的敏捷等价物之间的关系，给出了这本书里详细描述的这些技术是如何扮演和早期软件过程模型中的类似角色。

Rails非常强大，但是从版本1.0之后就变化很大，其中版本1.0最初是从一个特定的应用程序中 **提取** 出来的。事实上，Web本身就是从具体的细节演变到更加通用的架构模式：

图 13.1: 工程SaaS的良性三角是由三个软件工程王冠形成的（1）云计算上的SaaS（2）高产的框架和工具，和（3）敏捷开发

第1章	第二章（续表）	第三章
验收测试	HTTP cookie	访问方法
敏捷开发过程	HTTP方法	匿名λ表式
自动理论证明	HTTP服务器	应用程序根目录
云计算	篡改	回溯
集群	IP地址	阴塞
DRY（不要自我重复）	负载均衡器	类变量
形式化分析	逻辑层	封闭
功能测试	标记语言	鸭子类型
集成测试	主从式	动态类型
遗留代码	中间件	封装
生活方式	模型	函数程序设计
模型检查	多宿主	图形环境管理器
面向对象编程	MVC（模型－视图－控制器）	生成器
公共云服务	网络接口	获取方法
回归测试	网络协议	幂等
SaaS（软件即服务）	点对点技术	实例变量
SOA（面向服务架构）	持久层	植入
系统测试	主关键字	迭代器
测试覆盖率	公钥加密	词法域
	推送式	查找方法
单元测试	RDBMS（关系型数据库管理系统）	元编程
效用计算	关系代数	方法链
确认	关系型数据库	迁移
验证	请示－应答协议	mix-ins
虚拟机	路由	更改器方法
仓库规模的计算机	选择器符号	诗歌模式
瀑布开发过程	会话	接收器
第2章	SGML（标准通用标识语言）	反射
动作	数据拆分	正则
应用服务器	无共享架构	正则表达式
主从式架构	无状态协议	根类
控制器	结构化存储	设定方法
CSS（层叠样式表）	TCP端口号	静态变量
CRUD(创建，读取，更新，删除)	TCP/IP（传输控制协议/互联网协议）	符号
数据一致性	URI（统一资源标识符）	语法糖
设计模式dn	视图	类型强制转换
DNS(域名系统)	web应用框架	良品率
HAML（HTML抽象标记语言）	网络服务器	
主机名称	XHTML（扩展的超文本标识语言）	
HTML（超文本标记语言）	XML（可扩展标示语言）	
HTTP（超文本传协议）		

图 13.2: 这本书的前三章引入的概念和名词

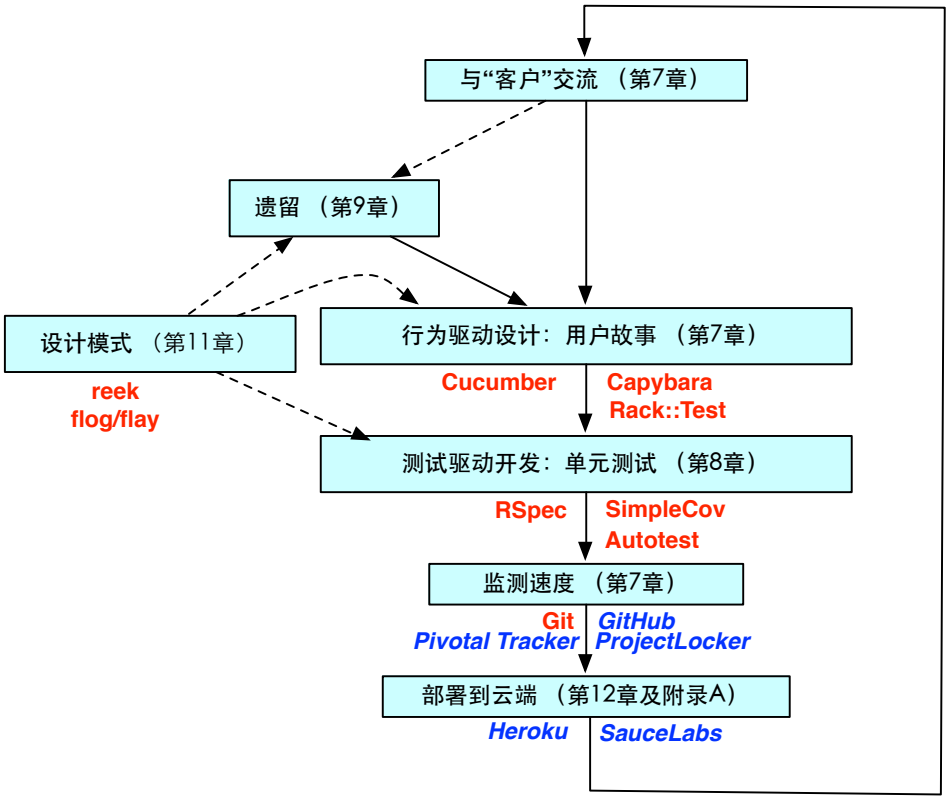


图 13.3: 敏捷软件生命周期的一个迭代以及它和这本书中的章节之间的关系，以及标出每一步的支持工具（红色粗体字）和支持的服务（蓝色斜体字）

- 从1990年的静态文件到1995年的动态内容；
- 从20世纪90年代早期的URI到21世纪早期的REST；
- 从20世纪90年代早期的会话“hack”（臃肿的URI，隐藏字段，等等）到20世纪90年代中期的cookie和实际会话；以及
- 从1990年的建立和管理你自己的自组织服务器到21世纪早期的部署在“管理”云平台。

编程语言Java和Ruby提供了另一个示范，好的渐进性想法很快就会受到欢迎，但是伟大的激进的想法需要时间来为大众接受。

Java和Ruby是相同的年龄，都出现在1995年。几年之内Java成为最流行的编程语言之一，然而Ruby仍然主要是编程语言文士们感兴趣的。Ruby在十年后发布Rails时才流行。Ruby和Rails证明了编程语言中的伟大想法真的可以通过广泛的软件重用带来高生产效率。同Ruby和Rails比较Java以及其框架，（Stella et al. 2008）和（Ji and Sedano 2011）发现代码的行数有3~5个点的减少，这是一个提高生产率的指示。

瀑布/螺旋	敏捷	章节
需求收集与分析	短迭代周期的BDD以使客户参与到设计中	7
周期性的代码审查	结对编程（两个人不断审查对方的代码）	10
定期设计评审	提取请示以促进关于改变设计的讨论	10
生成后测试整个设计	使用TDD按照你的设计持续地测试	8
后实现的集成测试	持续的集成测试	12
罕见的主要发行版	持续部署	12

图 13.4: 虽然敏捷方法并不适用于所有的项目，但是敏捷生命周期确实和传统的模型像瀑布模型和螺旋藻模型一样拥有同样的流程步骤，但是把作用域里的每一步减少为单一迭代，这样就可以频繁地重复做，不断细化产品的工作版本

13.3 展望未来

我一直对未来比对过去更加感兴趣。

——Grace Murray Hopper

鉴于快速发展的工具、模式和发展方法的历史，那么在未来的几年里软件工程师会期待什么呢?

我们希望在未来几年内变得受欢迎的一个软件工程技术就是 **三角洲测试** （delta debugging）（Zeller 2002）。它使用分割归并的方法来自动发现那些会引起错误的最小的输入变化。调试器通常使用程序分析来检测代码本身的缺陷。相比之下，三角洲测试标识出导致错误的程序 **状态** 的变化。它需要两个运行副本，一个有缺陷，另一个没有，并且着眼于状态集合之间的差异。通过不断地改变输入和使用二进制搜索策略重新运行程序以及自动化测试，三角洲测试有条不紊地缩小两个运行时之间的差异。三角洲测试发现依赖，形成一个因果链，它会一直扩展直到它标识出引发错误的输入表量的最小变化的集合。尽管它需要程序运行很多次，但是这个分析是全程序速度进行，并且不干预程序员，所以它节省开发时间。

程序合成 可能准备好了突破。今天的最先进的技术就是给出程序的不完整的片段，程序合成工具通常可以供给缺失的代码。这个技术的最有趣的一个用途是在微软Office Excel 2013中，称为 **闪光填充功能** （Flash Fill feature），它按照例子（Gulwani et al. 2012）来编程。你给出想对代码列或行做什么样的处理的例子，然后Excel就会试图重复和概括你做的内容。此外，你可以纠正它将它引导到你想要的内容上（Gantenbein 2012）。

随着正式方法实用性的发展，计划和文档与敏捷开发之间的分隔可能会变得更加明显。可以正式地验证程序的大小随着时间、工具的改善、更快的计算机以及更广泛的了解如何编写正式的规范而增长。如果在编码之前做了细致的规范要求就能不需要测试，就能得到完全验证过的程序，那么与代码变更有关的权衡就难以做到。为了正式的方法能够工作，显然变化需要比较小。当变化是司空见惯的时候，敏捷就是答案，因为变化就是敏捷的本质。

虽然对于今天的某些类型的应用程序来说敏捷比其他的软件方法工作得更好，但是它肯定不是软件开发的最后答案。如果一个新方法可以简化，包括一个好的软件架构和好的设计模式，同时保持敏捷的易于改变性，那么它可以变得更为流行。从历史上看，一个新的方法每隔10年或20年就会出现，所以可能很快就会有一个新的方法了。

这本书本身是在 **大规模开放网络课程** （MOOC）的黎明时期发展起来的，其中MOOC是另一种趋势，我们预测在未来几年里会变得更为重要。像这个现代世

界里的很多其他的进步一样，没有SaaS和云计算我们就不会有MOOC。授权组件包括：

- 可伸缩的视频分布通过YouTube这样的服务。
- 运行在评估作业的云计算上的复杂的自动评分系统可以立即扩展到成千上万的学生。
- 作为可扩展解决方案的讨论论坛，可以问问题和从其他学生和工作人员那里获取答案。

对世界各地的学生来说，这些组件组合起来形成一个美妙的、低成本的载体。例如，它肯定会提高我们快速变化的领域里的专业人士的继续教育，使得有天赋的预科学生可以达到他们的学校不能达到的领域，并且让世界各地没有机会上好大学的专注的学生仍然可以得到良好的教育。MOOC甚至可能会有提高传统课程的质量标准的副作用，通过提供切实可行的无效率讲师的替代者。如果MOOC仅仅可以兑现一半的这些机会，它们仍然会是高等教育的强大力量。

13.4 最后的话

> 最终，它归结为味道。它涉及到让你接触到人类已经做到的最好的事情，然后试着把这些东西应用到你正在做的事情中来。
>
> ——Steve Jobs

软件帮助人类登上月球，带来了用于救生的CAT扫描的发明，使得目击者公民新闻成为可能。作为一名软件开发者，你成为有改变世界的力量的社区的一份子。

但是能力越大，责任越大。软件错误引起了Ariane V火箭[6]和Mars Observer[7] 的损失，以及由Therac-25机器[8]的辐射过量引发的多个病人的死亡。

虽然早日的计算机和软件故事中占主导地位的是工作在车库或者创业公司的孤独天才的“前沿叙述”，但是今天的软件实在是太重要了，不是任何一个个体可以完成的，无论他多么有才华。就像我们在第10章中提到的一样，软件开发现在是一个团队运动。

我们相信这本书中的概念增加了你成为一个负责任的软件开发者 **和** 获胜的团队的一份子的可能性。没有教科书可以直接到达那种程度；继续写作、学习和重构来在你前进的时候应用你的课程。

和我们在第一章里说的一样，我们期待成为你和你的团队创造的美丽持久代码的热情粉丝！

13.5 深入学习

D. Gantenbein. Flash fill gives excel a smart charge, Feb 2012. URL `http://research.microsoft.com/en-us/news/features/flashfill-020613.aspx`.

S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. **Communications of the ACM** , 55(8):97–105, 2012.

F. Ji and T. Sedano. Comparing extreme programming and waterfall project results. **Conference on Software Engineering Education and Training** , pages 482–486, 2011.

L. Stella, S. Jarzabek, and B. Wadhwa. A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails. **10th International Symposium on Web Site Evolution** , pages 93–99, October 2008.

A. Zeller. Isolating cause-effect chains from computer programs. In **Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering** , pages 1–10, New York, NY, USA, Nov 2002. ACM. doi: 10.1145/587051.587053. URL `http://www.st.cs.uni-saarland.de/papers/fse2002/p201-zeller.pdf`.

注释

[1] `http://www.wired.com/threatlevel/2012/07/yahoo-breach`
[2] `http://github.com/ucberkeley/researchmatch`
[3] `http://github.com/vinsonchuong/meetinglibs`
[4] `http://guides.rubyonrails.org/security.html`
[5] `http://blog.codeclimate.com/blog/2013/03/27/rails-insecure-defaults`
[6] `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`
[7] `http://en.wikipedia.org/wiki/Mars_Observer`
[8] `http://en.wikipedia.org/wiki/Therac-25`

A 使用本书配套软件

Frances Allen
（1932——）因为有关优化编译器技术中的理论和实践中的开创性贡献而获得了2006年的图灵奖，这些技术奠定了现代优化编译器和自动并行执行的基础。

我做的所有事都只是一小块。我探究事物的边缘，找到做事的新方法。这让我能够保持非常非常投入。

——Fran Allen，计算机历史博物馆的研究奖上的留言，2000

A.1 Beta版本的指导 **412**
A.2 本书配套软件概览 **412**
A.3 在VirtualBox中使用Bookware的虚拟机 **412**
A.4 在Amazon的EC2上使用Bookware的虚拟机 **414**
A.5 编辑代码：编辑器和Unix生存技巧 **415**
A.6 开始使用Git进行版本控制 **415**
A.7 开始学习GitHub或ProjectLocker **417**
A.8 使用Heroku部署到云端 **420**
A.9 谬论和陷阱 **421**
A.10 深入学习 **422**

重要概念

这本书不仅仅是都关于创建SaaS：它也高度依赖SaaS、IaaS（**设施即服务，infrastructure as a service**）以及PaaS（**平台即服务**），所有这些都是 **云计算** 的模型。这个附录描述的云技术不仅仅让你作为学生时学习得更容易，而且在你部署真实的SaaS应用的时候，也是生态系统中必不可少的一部分。

正如本书所写的，所有这些云服务提供了一个零开销的使用层，这对于完成本书中的工作是足够的。

- 本书中使用的所有的 **开源软件** 都预装在了一个 **虚拟机镜像** 中——一台计算机的完整的硬盘内容，预装所有需要的软件。
- 为了使用一个虚拟机镜像，你需要把它部署到一个 **虚拟机监控程序** 上。我们给出在本机上使用VirtualBox[1]，或者部署在弹性计算云（Elastic Compute Cloud，EC2）上的说明，EC2是Amazon Web Services的一个 **设施即服务** 的云计算产品。
- GitHub[2] 和ProjectLocker[3]是SaaS站点，它们可以让你备份你的 **版本控制的** 项目以及和其他开发者进行合作。
- Heroku[4]是一个 **平台即服务** 的提供商，你可以在上面部署Rails应用程序。

A.1 Beta版本的指导

虽然我们在本书中采取了一系列措施来减少学生的痛苦，例如测试驱动开发（参见第8章）来快速地捕获问题，以及提供了一个带有一致性环境的虚拟机镜像，但错误仍然 **会** 出现。另外，这本书的Alpha测试是我们最开始预计的300倍大，并且被140多个国家使用，而不仅仅只是在一个教室里面，因此我们预计有未考虑到的问题，通过记住缩写词**RASP**：阅读，询问，搜索，发贴。大部分情况下，你都可以高效地解决问题。

阅读（Read）错误信息。错误信息看起来会是长得令人失望，但是一个长的错误信息通常是你的朋友，因为它对问题给出了一个强烈的提示，网上会有给出的错误信息的相关信息。

询问（Ask）同事。如果你在班上有朋友，或者有及时沟通的平台，请把信息放在那儿。

搜索（Search）错误信息。你将会很惊奇地看到有经验的开发者是如何通过诸如Google的搜索引擎通过查找错误信息中的关键字来解决问题的。

发布（Post）问题在StackOverflow[5] 或ServerFault[6] 这样的网站上（在你搜索一个类似的问题，确认这个问题还没有人问过之后!），这些站点专注于帮助开发者和允许你对某个问题最有价值的解答投票，这样它们最终会出现在解答列表的顶端。

A.2 本书配套软件概览

本书配套软件包含了三个部分。

第一部分是一个统一的开发环境，它预装了本书中提到的所有工具，为了方便性和统一性，这个环境通过一个 **虚拟机镜像** 提供。

第二部分包含针对开发者的一系列优秀的站点：GitHub[7]、ProjectLocker[8]、Heroku[9]和Pivotal Tracker[10]。**声明**：本书写作之时，这些站点提供的 **免费** 部分足够完成本书的任务。然而，这些服务的提供者或者工具可能在任何时候开始收费，这超出了我们的控制范围。

第三部分是关于本书的补充资料，无论你是否购买了本书，这些都是免费的：

- 本书的英文版的网站（`http://saasbook.info`[11]）包含了每个版本的最新的错误列表、补充的在线材料的链接、一个错误反馈机制以及高清晰的图片和表格，以防你在电子阅读器上阅读时有麻烦。
- GitHub （`http://github.com/saasbook`[12]）包含了本书中使用的不同的Rotten Potatoes例子的完整代码。
- Pastebin （`http://pastebin.com/u/saasbook`[13]）包含了本书中的每个例子的语法高亮，粘贴-复制的代码片段。这些代码也包含在了本书中文版的配套光盘中。
- Vimeo （`http://vimeo.com/saasbook`[14]）托管了本书中提到的所有视频。这些视频也包含在了本书中文版的配套光盘中。

A.3 在VirtualBox中使用Bookware的虚拟机

虚拟机技术允许在单一的物理机器上，“在你的机器内置的操作系统之上”运行一个或多个 **客户操作系统（OS）** 。这些虚拟机能够“打开电源”以及“关闭电源”，而不

需要和你计算机内置的操作系统进行交互（**宿主操作系统**）。一个 **虚拟机监控程序** 是一个用于运行虚拟机的应用程序。本书中的虚拟机运行在VirtualBox[15]之上，一个最初由Sun微系统公司（现在是Oracle公司的一部分）开发的免费且开源的虚拟机监控程序。VirtualBox可以运行在Linux、Windows或者Mac OS X上，只要宿主机有一个兼容Intel的处理器。

本书配套虚拟机的开发环境是基于GNU/Linux的。Linux是对Unix的一个开源实现，是最有影响力的操作系统之一，也是SaaS开发和部署的使用最广泛的环境。GNU（递归的定义为GNU's Not Unix）是几乎所有重要应用程序的开源实现的集合，特别是那些被开发者使用的应用程序。

■ **详细阐述：考虑过在计算机上安装自己的软件吗?**

本书每版的解释和例子都对 **特殊** 版本的Ruby，Rails，以及其他包括在虚拟机中的软件进行了交叉验证。跨版本的改变是显著的，用错误的软件版本来运行书中的例子也许会导致语法错误，不正确的行为，不同的信息，无声的错误，或者其他问题。为了避免迷惑，我们强烈推荐你使用虚拟机直到你对环境足够的熟悉，能够区分你自己的代码中由软件组件不兼容导致的错误。本书的网站包括了一个清单，包括我们给每个版本安装的任何东西，以及和Amazon的EC2兼容的虚拟机，以防你在使用Virtualbox的时候有麻烦。

遵循http://beta.saasbook.info/bookware-vm-instructions[16] 上面的指导去下载虚拟机以及在计算机上设置虚拟机，也可以解压后使用随书光盘中的saas-vm-1.0.0.vdi.zip文件。一旦设置完成之后，通常的工作流程如下：

(1) 开启VirtualBox，选择名称为`saasbook`的虚拟机，并且单击开始按钮。

(2) 进行你的工作。

(3) 当准备退出的时候，从VirtualBox的菜单中选择"关闭"命令。有三种选项来选择保存虚拟机的状态。

- **保存虚拟机的状态** 意味着下次恢复的时候，虚拟机将会和你离开的时候一样，就像你的宿主系统上的待机/恢复。
- **发送关机信号** 将会按序关闭客户操作系统；你的改变将会被保存，但是下次重启虚拟机的时候，虚拟机中现在正在运行的程序将会被关闭。这就像你在图A.1中使用`shutdown`命令一样。
- **关闭电源** 就像在虚拟机上拔掉电源一样，可能存在丢失一些数据的风险；除了其他选项不起作用，不推荐使用。这就像图A.1中使用`reboot -q`一样。

虚拟机网络 默认使用**网络地址转换**（NAT），给虚拟机一个能从宿主计算机访问的10.0.0.*地址，但是从外部世界却不可以访问虚拟机。如果这个在设置中不能用，VirtualBox提供了其他网络选项。

如果你的宿主计算机连接到了因特网，虚拟机也将会连上。然而，如果你暂停（睡眠）宿主计算机，然后在一个不同的物理网络上进行了重启（例如，在工作的时候休眠计算机，然后回家后唤醒计算机），你将需要重启虚拟机的网络子系统。为了这样做，在虚拟机的终端窗口中键入`sudo /etc/init.d/networking restart`。如果这不起作用，重启虚拟机。图A.1 给出了在虚拟机中工作的一些有用的命令。

重启虚拟机网络（当宿主计算机休眠，然后在一个不同的网络恢复）	`sudo /etc/init.d/networking restart`
正常地重启虚拟机	`sudo /sbin/reboot`
强制重启虚拟机（当正常不起作用）	`sudo /sbin/reboot -q`
正常地或强制地关闭虚拟机	在上面两行中用`shutdown`代替`reboot`

图 A.1: 一些有用的"内建"命令，你可以在虚拟机的终端中使用。一些可以通过虚拟机的GUI使用，但是在Unix的世界，最好能够理解命令行工具，以防GUI本身有问题

■ **详细阐述：自由和开源软件**

Linux最开始是由芬兰的程序员Linus Torvalds创建给自己使用的。GNU计划是由Richard Stallman发起的。Stallman是Emacs编辑器的创建者，以及自由软件基金的创立者（该基金管理着GNU），一个著名的开发者，对开源软件的角色有着十分强烈的观念。Linux和GNU被来自全世界的数千的贡献者持续地改善着；事实上，Linus后来发明了Git来管理这个大范围内的合作。尽管它们的开发表面上缺乏一个集中式的管理，和其他基于传统的中心模式下开发的软件相比，GNU和Linux的更健壮。这个现象在Eric Raymond的 **大教堂和市集** [17]中进行了解释，有人认为这是自由和开源软件运动（FOSS）的开创性宣言。

X 视窗系统 于1984年起源于MIT，是一种方法以使GUI可以跨操作系统和硬件。基于X11的程序通过TCP/IP来发送图形输出以及接收来自于X服务器的用户输入。令人疑惑的是，因为X服务器需要在屏幕上作图，并且接收来自于鼠标和键盘的用户输入，它通常运行在客户计算机上，而X客户端运行在云端的服务器上！

A.4 在Amazon的EC2上使用Bookware的虚拟机

在计算机上进行开发的另一个选择就是使用Amazon的EC2，我们准备了一个Amazon的机器镜像格式的虚拟机镜像，和在VirtualBox兼容的虚拟机上预装的软件一样。本书的站点[18]有一个指向和本书版本相应的AMI，也就是1.0，以及用于在EC2上的虚拟机进行部署和记录日志的指导。如果你使用这种方法，有两条警告你需要记住。首先，EC2实例默认的安全配置允许你通过`ssh`登录，但是不允许你通过HTTP连接到上面运行的SaaS应用。也就是说，你不能够在本机上运行一个浏览器，然后访问EC2镜像上的一个应用程序。有两种方法来处理。一个是改变默认的安全设置，允许HTTP和HTTPS访问，具体方法详见Amazon的EC2的指导[19]所述。另一种方法是在本机上运行一个 **X服务器** ，例如针对Windows的XMing[20]，或者Mac OS X的XQuartz[21]，并且允许浏览器运行在EC2实例上，但是把图形直接展示在计算机上。本书站点上的虚拟机信息页[22] 解释了如果你使用这个方案，怎样更改`ssh`来登录到实例上。

使用EC2的第二个警告涉及到当工作任务结束后，你可以选择 **停止** 或者 **关闭** 实例，（简单地退出实例不会关闭它；它将会继续运行并且可能会增加收费。）这两种选项都可以禁止实例，这样就停止了收费。然而，停止一个实例将会保留和它相关的虚拟硬盘，这样任何改变都会被保留下来，直到你下次重启实例。在这段时间内，虚拟硬盘将会在Amazon上面，并且有可能会引起使用收费，一旦超过了免费的份额（在2012年2月对于Amazon的EC2有30GB的免费硬盘空间）。如果你关闭了实例，虚拟的磁盘将会被 **永久清除** ，并且任何改变在重启实例的时候都 **不** 会出现。无论你使用哪种形式，确保使用Git和一个基于云端的Git备份，如GitHub或者ProjectLocker，正如A.6节所描述，请将重要工作备份到云端！

A.5 编辑代码：编辑器和Unix生存技巧

通过使用可以对所使用的语言进行语法高亮和自动缩进的编辑器，你会减少很多麻烦。你可以直接在虚拟机上编辑文件，或者使用VirtualBox的“共享文件夹”特色来使得虚拟机上的某些目录可以直接在Mac或者Windows PC上进行访问，这样就可以直接使用Mac或者PC上的本机编辑器。

许多支持Ruby的 **集成开发环境** （IDE），包括Aptana[23]、NetBeans以及RubyMine，都有语法高亮、缩进和其他有用的功能。这些IDE也提供一些其他和开发相关的功能，如运行测试，在本书中我们使用命令行主要出于三个原因：首先，和IDE不同，命令行在所有的平台上都是一样的；其次，我们在本书中高度强调自动来避免错误和提高生产率，GUI任务通常不是自动的，然而命令行工具可以组合到脚本中，这Unix哲学中的一个核心方法；第三，理解在开发的某个方面涉及到了哪些工具有助于了解IDE的“魔幻帷幕”，我们认为这有助于学习一个新系统，因为在使用GUI的时候某些事出错了，为了找到问题，你需要理解GUI实际上是怎样处理任务的。

记住这些，在虚拟机上有两种方法去编辑文件。第一个是使用虚拟机上面的编辑器。我们在虚拟机上预装了两个流行的编辑器。一个是 `vim`，一个轻量级的编辑器。我们进行了足够的自定义功能，包括语法高亮和自动缩进。这里有一系列关于这个流行的编辑器的教程和视频链接[24]。另外一个就是 **Emacs**，可定制编辑器的祖师爷，也是著名的 **Richard Stallman** 的作品之一。官方教程[25]是由自由软件基金提供，虽然有许多其他的教程。我们在虚拟机上的`vim`和`emacs`中包括了自动支持编辑Ruby和Rails应用程序。

vim 表示“**vi im** proved，”因为它是对早期的Unix编辑器 **vi** 的一个很大的改进，由Unix的传奇，Sun公司的联合创始人以及Berkeley的校友 **Bill Joy** 于1976年所编写。

第二种编辑文件的方式就是在Mac或者Windows计算机上进行，这需要设置VirtualBox的共享文件特色，如`http://beta.saasbook.info/bookware-vm-instructions`[26]所解释。支持Ruby的免费编辑器包括Mac OS X上的TextWrangler[27]，和Windows上的Notepad++[28]。

从不要把代码复制和粘贴到一个文字处理器中，如微软的Word或Pages。 许多文字处理器帮助把普通的引号（`"`）转化成“智能引号”，把两个破折号（`--`）转化成一个长破折号（——），以及执行其他会让你的代码不正确，引起语法错误，以及给你带来麻烦的转换。不要这样做。

A.6 开始使用Git进行版本控制

版本控制，也称为源代码控制或者软件配置管理（Software Configuration Management, SCM），是一个跟踪一系列文件的变更历史的过程。它能够告诉是谁何时进行了某次更改，重建一个或多个文件在过去某个点的状态，或者选择性地合并由不同人进行的更改。版本控制系统（Version Control System，VCS）是一个帮助管理这个过程的工具。对于单独的开发者，SCM给项目提供了一个有时间戳和标注的变更历史，和一个简单撤销那些引起错误的改变的方法。第10章讨论了SCM给小团队带来的许多额外的好处。

SCM还是VCS? 令人疑惑的是，缩写词SCM和VCS通常交互使用。

我们将使用Git进行版本控制。基于云的Git托管服务有GitHub和ProjectLocker，尽管不是Git所需要的，但使用他们是非常有好处的。因为它们使得小团队非常方便地进行协作（正如第10章所描述的），并且给个人开发者一个备份代码的地方。我们包括了几个用于GitHub和ProjectLocker的设置例子；一旦这些初始设置完成了，项目管理命令对于两者以同样的方式工作。这节包括了Git的基本知识，接下来的两节包括了一个GitHub和ProjectLocker的设置指导。

Linus Torvalds 发明了Git来辅助Linux项目的版本控制。即便你使用像Subversion等其他VCSs，你都应该阅读这部分。因为Git的概念模型完全不一样。

和所有的版本控制系统一样，Git的一个关键概念就是项目的 **repository** （仓

库），通常简写为 **repo**，它包括了组成项目的一系列文件的完整的变更历史。为了开始使用Git来跟踪一个项目，你应该首先`cd`到项目的顶层目录，和使用`git init`命令，它在目录中初始化一个空的仓库。 **跟踪的文件** 是那些是仓库永久部分的，因此它们的修改信息被维护、备份。`git add`用于添加一个文件到跟踪的文件集合中。并不是每个项目文件都需要被跟踪——例如，作为开发过程的一部分被自动创建的中间文件，如日志文件，通常是不跟踪的。

视频A.6.1演示了基本的Git工作流程。当你开始一个新的项目时，`git init`设置项目的根目录作为一个Git的仓库。当你在项目中创建了文件，对于每一个新文件，你使用`git add` **filename** 会导致新文件被Git跟踪。当到达对当前的项目状态满意的点，你 **提交** 改变：Git预先准备将是这次提交的一部分的所有变更的清单，并且在一个编辑器中打开清单，这样你能添加一个描述性的注释。编辑器通常通过 **配置设置** 来决定，正如下面所描述的。提交将会产生一个被跟踪的文件和注释被永久记录的快照。这个快照被分配一个 **提交ID**，一个40位的十六进制的数字，很奇怪的是，它将是全局范围内唯一的（不仅仅是在使用的Git仓库中，而是在所有的仓库中）；一个例子也许是`1623f899bda026eb9273cd93c359326b47201f62`。这个提交ID是指示项目在那个点上的状态的标准方式，但是正如我们将见到的，在冗余的提交ID之外，Git提供了更方便的方式来指定一个提交。通常的方式是给提交指定一个在仓库范围内唯一的前缀，例如对于上面这个例子使用`1623f8`。

SHA-1 算法 用于计算一个40位的单向哈希值来代表整个树，表示项目在那个时间点状态。

为了指定Git应该使用`vim`编辑器来进行更改，你应该设置`git config --global core.editor 'vim'`。这个操作不在乎你在哪个文件夹下进行，因为`--global`表示这个选项应该对 **所有** 仓库的 **所有** Git操作都起作用。（大部分的Git配置变量都能够在每一个仓库中进行设置。）对这个特殊的设置，其他有用的值是`'mate -w'`用于MacOS上的TextMate，`'edit -w'`用于MacOS上的TextWrangler，以及Windows的非常冗余的`"'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`，在所有的情况中，变量的引号都是必须的，用于防止空格导致编辑器的名字被划分成多个命令行参数。

和MacOS不同，Windows shell和Unix的标准不同，因此通常让Unix的工具在Windows上正确工作都很笨拙。

视频A.6.1: 单独开发者的基本Git流程.
光盘文件：ch_bookware/mov/git_basics.mp4
在这个简单的工作流程中，`git init`用于使用Git开始跟踪一个项目，`git add`和`git commit`用于添加和提交两个文件，然后一个文件被更改了，`git status`显示一个跟踪的文件有一些改变，`git diff`用于预览应该被提交的变更。最后`git commit`用于再次提交新的变更，`git diff`用于显示一个文件的不同提交版本之间的差别，表明`git diff`可以比较一个文件的两次提交，或者比较一个文件的当前状态和之前的一些提交。

记住尽管`git commit`永久地记录了当前仓库状态的一个快照，能够在以后的任何时候进行重建，但是它没有在其他地方创建仓库的其他备份复制，也没有让其他开发者可以访问。下一节将会描述怎样使用一个基于云端的Git托管服务达到这些目的。

代码文件：ch_bookware/code/ssh_keygen.sh

```
cd ~/.ssh
ssh-keygen -t rsa -C "yao@acm.org"
# when prompted for 'File in which to save key', press Return to accept
    default
# when prompted for passphrase, press Return to leave blank
chmod 0600 *
```

图 A.2：在Linux上创建一个 ssh （secure shell）密钥对，注意将这些命令中的yao@acm.org替换成你喜欢的电子邮箱地址。执行第二条命令后，一条"The key fingerprint is..."的消息表示执行成功。最后一条命令改变了密钥文件的权限，以使你可以读取它们——这是 ssh 能正常工作的一个条件。对于书中的练习来说，将密码留空也可以，这个项目GitHub指南[30]介绍了使用密码的步骤，以及为什么应该经常使用它们

■ **详细阐述：添加，提交及Git索引**

上面对Git的简化讨论忽视了 **索引** ，一个用于将要被提交的暂存区。`git add`不仅仅用于添加一个新文件到项目中，也缓存一个已经存在的文件状态用于提交。因此如果Alice修改了 **已经存在的** 文件`foo.rb`，她通常需要显式的使用`git add foo.rb`来让她的改变在下一次`git commit`的时候进行提交。分开这些步骤的原因是`git add`快照文件的中间状态，因此即便`commit`稍后才出现，被提交的版本和文件在`git add`进行时候的状态一样。通过给`git commit`使用`-a`选项，我们简化了讨论，这意味着"提交跟踪文件的所有当前的改变，而不管`git add`是否添加它们。"（`git add`在添加一个新文件的时候仍旧是必需的。）

A.7 开始学习GitHub或ProjectLocker

有许多种基于云的Git托管服务。我们推荐Github和ProjectLocker并给出其说明指导，这两者都提供免费账户。Github可以提供足够多的项目（仓库），但所有的项目都是开放阅读的；ProjectLocker将数据的规模限制在一定范围，但是允许将其限制在你允许的合作者之间，并不是所有人都能看到。这两种服务都允许你使用升级的付费服务，并且如果你的导师有优惠券或优惠码，就能临时升级到一个付费服务，但后续并不收费。

不管你选择哪个，基本的理念是相同的。你建立一个 **私钥** 来向云服务证明你是账户持有者，配置好Git使其知道这个密钥。不管有多少个不同的账户或仓库，你仅仅需要一个密钥就够了，所以你只用设置一次。接下来，你就可以 **提交** 你的仓库到云服务，为仓库创建一个名为 **remote** 的远程副本。在你允许的情况下，其他开发者就能 **推送（push）** 他们的改动，或者从云端 **拉取（pull）** 其他人的改动。

不管你选择哪个服务，以下两步都是必须做的。

(1) 告诉Git你的用户名和邮箱，以使一个多人合作的项目的每一次提交都与其提交者关联。

代码文件：ch_bookware/code/git_username.sh

```
git config ---global user.name 'Andy Yao'
git config ---global user.email 'yao@acm.org'
```

(2) 创建一个 **ssh** （secure shell）密钥对，在使用图A.2的命令时向云端托管服务证实你的身份。

开始学习Github：

(1) 如果你还没有账户，那么在GitHub[31]创建一个。你可以选择免费账户，如果是全日制学生的话也可以申请一个学生账户[32]，或者如果你的导师有特定优惠码，就可能会给你一个临时的微型计划（允许私有仓库）而不用付费。

(2) 在你的账户上按照Github添加公钥的说明[33]的**第4步到第5步**执行。（你已经完成了设置用户名和电子邮箱的其它步骤，而如后面的步骤所述，我们也不用设置一个令牌。）

(3) 创建一个GitHub仓库将会创建一个已有项目仓库的远程仓库，填写New Repository[34]表格并提交，同时注意你的仓库的名称。与你的项目最顶层目录名称对应是个不错的选择，例如`myrottenpotatoes`。

(4) 回到你的开发机器（虚拟机）在一个终端中进入（`cd`）项目的顶层目录（你之前输入的地方）并输入：

代码文件：ch_bookware/code/push.sh

```
1 git remote add origin git@github.com:myusername/myreponame.git
2 git push origin master
```

第一条命令是告诉Git你在GitHub上添加一个新的远程副本，并且从现在开始我们用名称`origin`来代表这个远程副本。（这个名字符合Git用户的习惯，第10章解释了原因。）第二条命令告诉Git将仓库在本地的任何改动 **推送（push）** 到`origin`这个还不存在的远程副本上。

开始学习ProjectLocker：

(1) 如果你没有账户的话，在ProjectLocker[35]建立一个。它的免费方案对本书中的工作是足够的。

(2) 当登录到ProjectLocker时，在左边导航工具栏上，单击管理公钥（Manage Public Keys），然后点击新密钥（New Key）。

(3) 在新公钥的对话框中，在Name一栏为这个密钥填写一个容易记的名称（例如，用“SaaS”来提醒你这是你在读这本书的时候建立的密钥）。在用户名（User Name）一栏，输入你在图A.2使用的电子邮箱地址。最后，使用命令`cat ~/.ssh/id_rsa.pub`查看公钥文件，并将文件的 **所有内容** 小心地复制粘贴到密钥（Key）栏中，点击保存公钥（Save Public Key）。

(4) 为了给你已有的项目仓库创建一个ProjectLocker仓库，点左边导航栏中的添加项目（Add Project），填写表格，注意你选择的项目名称，最好与你本地的项目顶层目录相对应，例如`myrottenpotatoes`。

(5) 点击左边导航栏中的用户主页（User Home)链接，你会看到你的ProjectLocker项目列表，包括刚刚创建的那一个。注意与项目相关的字段“（你的Git位置）Your Git Location”，对用户`andrewyao`以及项目`minimax`来说，这个位置是类似`git-andrewyao@pl5.projectlocker.com:minimax.git`的东西。

(6) 回到你的开发机器（虚拟机），在一个终端窗口中进入（`cd`）你项目目录的最顶层（即之前你输入`git init`的地方）并输入：

代码文件：ch_bookware/code/push_pl.sh

```
# replace 'andrewyao' with your ProjectLocker account name,
# 'pl5' with whatever appears after '@' in the My Git Location
#  of your project on ProjectLocker, and 'minimax' with the
#  name you chose for 'Add Project' on ProjectLocker
git remote add origin git-andrewyao@pl5.projectlocker.com:minimax.git
git push origin master
```

第一条命令是告诉Git你在ProjectLocker上添加一个新的远程副本，并且从现在开始我们用名称`origin`来代表这个远程副本。（这个名字符合GitHub用户的习惯，第10章解释了原因。）第二条命令告诉Git将仓库在本地的任何改动 **推送（push）** 到`origin`这个还不存在的远程副本上。

以上建立账户和管理密钥的步骤只需要进行一次。但对每一个项目一为说，创建一个新的仓库时都必须使用`git remote`。每次对一个特定仓库用`git push`时，你都将自上次推送（push）以来的所有改动传到了GitHub的云端，这样做的一个好的作用就是保持了项目的最新备份。

图A.3总结了本章中介绍的基本Git命令，这对一个初学的单独开发者来说已经足够了。当你在一个团队中工作时，你需要使用第10章介绍的更多的Git功能和命令。

命令	功能	使用情景
`git pull`	从其它开发者获取最新的发动，并将其合并到你的仓库	每次你坐下来准备编辑共享项目中的文件时
`git add` **file**	将文件 **file** 添加到提交范围	当你添加了一个沿未被追踪的文件时
`git status`	查看哪些发动在等待提交，以及哪些文件未被追踪。	在提交之前，确保没有重要的文件“没被追踪”（如果这样的话，使用`git add`来追踪它们）
`git diff` **filename**	查看某个文件在当前版本和上次提交的版本之间的差异	查看你改动了什么，以防造成破坏。这个命令有许多选项，在第10章中有详细描述。
`git commit -a`	将 **所有(-a)** 追踪的文件的发动提交。一个编辑窗口将会打开，允许你输入提交的信息来描述将要提交的发动	当你处在一个稳定的点，想要对项目的状态有个快照，以便之后需要时可以回滚到这个点
`git checkout` **filename**	将一个文件还愿到它上次提交后的状态。**警告：** 任何从上次提交以来你做的发动都将丢失。这个命令有许多选项，一些在第10中有描述	当你需要“回滚”一个或多个文件到一个已知版本时
`git push` **remote-name**	将你的仓库中的发动推送到名为 **remote-name** 的远程仓库中。如果你按照A.7节的指导创建了项目的话，这个名称默认为`origin`。	当你想让其它开发者可以获得你最新的改动时，或者将你的发动备份到云端

图 A.3: 常用的Git命令。一些命令看起来是任意指定的咒语一样，因为它们是很多功能强大的命令的特殊形式，随着学习到更多的Git功能，你会更理解它们的意义

A.8 使用Heroku部署到云端

第4章的重要概念中是这里讨论的重点，所以请先阅读该章节。

像Heroku这样的新的云计算技术使软件即服务（SaaS）部署变得比任何时候都要简单。如果你还没有一个Heroku[36]账户的话，创建一个免费的。对本书中的项目来说，免费账户提供了足够的功能。Heroku支持很多种语言和框架的应用。对Rails应用来说，Heroku提供了包括必要功能的一个工具包，这在Bookware的虚拟机中已经安装了。当你创建了一个Heroku账户，登录到虚拟机并在终端中输入`heroku login`和`ssh-keygen`，创建一对密钥在部署Heroku时会用到。这个步骤只需要进行一次。

另一个可能的选择，其它生产环境的选择可以是**mysql**工具包，它可以与流行的开源MySQL相连接。

第4章介绍了Rails定义的三种环境（开发、生产、测试）。当部署Heroku或其它平台时，你开发的应用将在生产环境中运行。开发环境与Heroku的生产环境主要有两个地方不同。其一，Heroku使用PostgreSQL数据库而不是SQLite3，所以需要更新`Gemfile`来指示你的应用在生产环境中依赖于PostgreSQL连接器工具包，但在开发时依赖于SQLite3。以下摘录说明了这一不同。

代码文件：ch_bookware/code/heroku_gemfile.rb

```
# for Heroku, replace "gem 'sqlite3'" in your Gemfile with this:
group :development, :test do
  # if you already have a 'group :development,:test' block in your
  # Gemfile, you can just move the line "gem 'sqlite3'" into it.
  gem 'sqlite3' # use SQLite only in development and testing
end
group :production do
  gem 'pg' # use PostgreSQL in production (Heroku)
end
```

其二，从Rails 3.1开始，Rails提供了能从用更高层次的格式生成CSS的工具**Sass**，以及用更高层次的格式生成JavaScript的**CoffeeScript**。为了告诉Heroku在没有Sass和CoffeeScript文件可用的情况下也可以使用CSS和JavaScript文件，你需要更新`config/environments/production.rb`中以下几行：

令你感到困惑的是，你不能从`project-locker.com`回访登录ProjectLocker，而只能从portal.projectlocker.-com[37]登录。

代码文件：ch_bookware/code/production_environment.rb

```
# in config/environments/production.rb:
# BEFORE - what 'rails new' generates for production.rb (l.18):
config.assets.compile = false
# AFTER - this is what you should change it to:
config.assets.compile = true
```

和通常一样，由于更新了`Gemfile`，你需要运行`bundle install --without production`来确保你的应用依赖能够满足。（`--without production`省略了安装**group :production**中的工具包。我们这样推荐是由于在没有完整安装PostgreSQL的情况下，在计算机上安装pg工具包会失败。）一旦做了这些改变，而且你对当前应用的状态比较满意，那么就可以开通部署了。

本质上来说，除了推送到远程的时候对部署应用有副作用外，Heroku与Git远程（参见A.7节）的行为比较类似。

(1) **确保应用运行正常，通过了所有本地测试。**远程调试一般会更难。在部署之前，想办法使你对本地应用更有信心。

Heroku部署的必然是`master`的Git分支。关于分支第10章有讨论。

(2) 登录到虚拟机，进入到你想要部署的应用的根目录，确保你本地副本Git仓库中`master`分支是最新的。因为这是将要部署的版本。

(3) `heroku create --stack cedar`
这条命令只在**第一次**部署给定应用时需要。它创建了一个新的Heroku应

用，并给了一个预先分配的古怪命名，这个命名也决定了URI。例如，如果预告给定的名称是`luminous-coconut-237`，那么应用将会被部署在`http://-luminous-coconut-237.herokuapp.com`。你可以登录到更Heroku账户，点击我的应用来更新应用的名称。

(4) `git push heroku master`
这个命令将当前Git仓库的`master`分支最新版本推送到Heroku，就像Heroku是一个远程Git一样。该命令还让Heroku在其后运行`bundle install`命令来确保所有必需的工具包都是可用的，然后部署应用。在这个版本的`git push`命令中，我们推送到了远程的`heroku`副本（而不是默认的`origin`）并指定我们推送的是`master`分支，因为所有的Heroku开发都必须发生在这个分支中。

(5) `heroku ps`
这个命令检查你部署的应用的进程状态（`ps`）。**状态**一列应该显示类似"Up for 10s"这样的消息，表示应用已经启动了10秒钟。还可以使用`heroku logs`来显示应用的日志文件。当在开发环境中运行正常却在生产环境中出现问题时，这是一个有用的方法。

(6) `heroku run rake db:migrate`
如果是你第一次部署这个应用，这个命令将会创建数据库。如果是一个新版本的发行部署，那么最好仅在等待迁移时使用（如果没有，那么这个命令不做任何事情）。Heroku也有关于如何从开发环境中导入数据[38]到生产环境的说明。

对于同一个应用后来的部署，路过3这一步骤即可。图A.4总结了一些你在开发模式中使用的命令是如何在Heroku中部署的应用上使用的。

本地（开发）	**Heroku（生产）**
`rails server`	`git push heroku master`
`rails console`	`heroku run console`
`rake db:migrate`	`heroku run rake db:migrate`
`more log/development.log`	`heroku logs`

图 A.4: 开发模式下有用的命令，以及如何在Heroku上部署的应用中获得同样的功能

■ **详细阐述：产品最佳实践**

在以上介绍中，我们省略了Heroku推荐的[39]在生产环境中"加固"应用的两种好的实践方法。首先，我们的Heroku部署仍然使用WEBrick作为表示层。Heroku推荐使用新型的`thin`网络服务器来获得更了的性能。第二，由于SQLite3和PostgreSQL的功能之间有微妙的差异，这些差异有可能引起迁移相关的问题，例如有可能使数据库模式变得更复杂。Heroku推荐在开发和产品环境中都使用PostgreSQL，这就要求在虚拟机中安装并配置PostgreSQL或其它开发计算机上。一般来说，我们建议保持开发环境和生产环境尽可以相似，以避免在开发环境中正常工作但在生产环境却崩溃的难以调试的问题。

A.9 谬论和陷阱

陷阱：进行过大的检入（提交）。

Git使提交变得更快更简单，所以你应该经常地提交并使每一次改动较小。这样，如果某一次提交引入了错误，你不需要撤销其它所有的改动。例如，如果你更新了关于功能A的两个文件以及关于功能B的三个文件，分两次提交以妨某一次部分改动在之后要撤销。事实上，高级的Git用户使用`git add`来从改动的文件中"选择最佳的"子集作为一次提交：添加你想要的特定文件，并且 **省略** `git commit`中的`-a`选项。

陷阱：忘记将文件添加到仓库中。

如果你创建了一个新文件但忘记了添加它到仓库中，那么代码的副本仍将正常工作但是文件不会被追踪或打包。在人提交或推送之前，使用`git status`来查看是否有没有被追踪的文件，并使用`git add`添加该列表不中任何你想要追踪的文件。你可以使用`.gitignore`[40]文件来避免收到关于你永远都不想追踪的文件的警告，例如二进制文件或临时文件。

陷阱：混淆了commit（提交）和push（推送）。

在 **你的** 一个仓库副本中，`git commit`会捕获一段时期内改动的一个快照，但除非你使用`git push`来将它们传送到其它仓库（例如origin）中，其他人不会看到你的改动。

陷阱：在宿主机移动后忘记重新设置虚拟机网络连接。

记住虚拟机是依赖于宿主机的网络设备的。如果你的宿主机移动到了一个新的网络，例如你在家里休眠了它但在上班唤醒，这类似于拔掉然后重新连接宿主机的网线。因此虚拟机必须重新连接它的虚拟网络。你可以通过VirtualBox的Devices菜单来完成。

陷阱：隐式的假设在开发环境和生产环境中有所不同。

第4章解释了Bundler和Gemfile如何自动管理应用对外部库的依赖性，以及migrations如何自动改变数据库。在本附录中，我们展示了Heroku如何依赖这些途径来成功部署应用。如果你手动安装了工具包而不是在Gemfile中列出它们，在Heroku中这些工具包会丢失或产生版本错误。如果你手工动改变数据库而不是使用migrations，Heroku就无法使生产环境的数据库与开发数据库匹配。应用其它的依赖性包括数据库类型（Heroku使用PostgreSQL），Ruby和Rails的版本，用作表示层的特定的网络服务器，等等。当类似Rails的框架和类似Heroku的部署平台竭尽全力保护应用在这些方面的变异时，使用migrations和Bunlder这样自动工具，而不是手动地改变开发环境，最大化依赖文档化的可能性，你才能保持开发和生产环境同步。如果一件事情能自动进行，并能用一个文件记录，那么你就应该这样做。

A.10 深入学习

- Git Community Book[41]是一个不错的在线索引，同时也能下载为PDF文档。

注释

[1]http://virtualbox.org
[2]http://github.com
[3]http://projectlocker.com
[4]http://heroku.com
[5]http://stackoverflow.com
[6]http://serverfault.com
[7]http://github.com
[8]http://projectlocker.com
[9]http://heroku.com
[10]http://pivotaltracker.com
[11]http://saasbook.info
[12]http://github.com/saasbook
[13]http://pastebin.com/u/saasbook
[14]http://vimeo.com/saasbook
[15]http://virtualbox.org
[16]http://beta.saasbook.info/bookware-vm-instructions
[17]http://catb.org/~esr/writings/homesteading/cathedral-bazaar/
[18]http://saasbook.info
[19]http://aws.amazon.com/documentation/ec2/
[20]http://www.straightrunning.com/XmingNotes/
[21]http://xquartz.macosforge.org/
[22]https://beta.saasbook.info/bookware-vm-instructions
[23]http://aptana.com/
[24]http://www.andreszenteno.com/index.php?option=com_content&view=article&id=21737:25-vim-tutorials-screencasts-and-resources&catid=55:nettuts&Itemid=166
[25]http://www.gnu.org/software/emacs/tour/
[26]http://beta.saasbook.info/bookware-vm-instructions
[27]http://www.barebones.com/products/textwrangler/
[28]http://notepad-plus-plus.org/
[29]http://help.github.com/ssh-key-passphrases
[30]http://help.github.com/ssh-key-passphrases
[31]http://github.com
[32]https://github.com/edu
[33]http://help.github.com/linux-set-up-git/
[34]https://github.com/repositories/new
[35]http://projectlocker.com
[36]http://heroku.com
[37]http://portal.projectlocker.com
[38]http://devcenter.heroku.com/articles/taps
[39]http://devcenter.heroku.com/articles/rails3
[40]http://book.git-scm.com/4_ignoring_files.html
[41]http://book.git-scm.com/