

The Addison Wesley Signature Series

用户故事 与敏捷方法

A KENT BECK
SIGNATURE
BOOK

- 敏捷大师Mike Cohn的软件需求方法圣经
- 小型团队(项目)不可或缺的敏捷开发宝典
- 亚马逊五星级长销图书，敏捷社区重点推荐
- 结合精髓和实例，充分演绎用户故事的智慧

(美) Mike Cohn 著
Kent Beck 作序
石永超 张博超 译
李国彪 滕振宇 审校



清华大学出版社



The Addison Wesley Signature Series

用户故事与敏捷方法

在敏捷社区的详尽评审和热切期盼下，《用户故事与敏捷方法》不负众望，为软件行业提供了一种节省时间和消除重复工作的需求管理方法，对开发更优秀的软件起着积极、高效的推动作用。

构建满足用户需求的软件，最好的方法是从“用户故事”开始，即简明扼要、清楚明确地描述对实际用户有价值的功能。在《用户故事与敏捷方法》中，敏捷大师 Mike Cohn 提供详尽的蓝图来指导我们如何编写用户故事，如何把它们应用于软件开发生命周期中。

《用户故事与敏捷方法》介绍了如何编写理想的用户故事，造成用户故事不理想的原因有哪些，如何在无法与用户交流的情况下有效地搜集用户故事，如何对已经写好的用户故事进行整理、排列优先级及在此基础上进行计划、管理和测试。

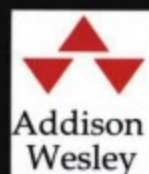
- 专注于“用户故事”这一灵活、敏捷和实用的需求方法
- 强调如何用更短的时间开发更符合用户需求的软件应用
- 揭示如何在不能直接与用户交流的情况下搜集用户故事
- 诠释用户故事的优势，用户故事与用例、使用场景和传统需求方法的不同
- 精辟阐述如何围绕着用户故事进行全面的规划、进度、估算和测试
- 极限编程 (Extreme Programming)，Scrum 或其他任何敏捷方法的最佳伴侣

采用XP和Scrum等敏捷开发方法或其他软件开发方法的开发人员、测试人员、分析师和管理人员，可以从《用户故事与敏捷方法》获得有价值的信息，以更少的人员在更少的时间内开发出更符合用户需求的软件应用。

更多信息，请访问以下网址：

www.awprofessional.com

www.tup.com.cn



上架建议：软件开发方法

ISBN 978-7-302-22340-5



9 787302 223405 >

定价：39.00元

用户故事与敏捷方法

(美)Mike Cohn 著

石永超 张博超 译

李国彪 滕振宇 审校

清华大学出版社

北 京



内 容 简 介

本书详细介绍了用户故事与敏捷开发方法的结合,诠释了用户故事的重要价值,用户故事的实践过程,良好用户故事编写准则,如何搜集和整理用户故事,如何排列用户故事的优先级,进而澄清真正适合用户需求的、有价值的功能需求。

本书对于软件开发人员、测试人员、需求分析师和管理者,具有实际的指导意义和重要的参考价值。

Simplified Chinese edition copyright © 2010 by **PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.**

Original English language title from Proprietor's edition of the Work.

Original English language title: **User Stories Applied: For Agile Software Development** © 2010

EISBN: 978-0-321-20568-1

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Pearson Education.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).

本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字: 01-2009-6813

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

用户故事与敏捷方法/(美)科恩(Cohn M.)著;石永超,张博超译;李国彪,滕振宇审校. —北京:清华大学出版社, 2010.4

书名原文: User Stories Applied: For Agile Software Development

ISBN 978-7-302-22340-5

I. 用… II. ①科… ②石… ③张… ④李… ⑤滕… III. 软件设计 IV. TP311.5

中国版本图书馆 CIP 数据核字(2010)第 056499 号

责任编辑:文开琪

责任印制:杨 艳

出版发行:清华大学出版社

地 址:北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:清华大学印刷厂

装 订 者:三河市李旗庄少明装订厂

经 销:全国新华书店

开 本:185×230 印 张:15.25 字 数:355 千字

版 次:2010 年 4 月第 1 版 印 次:2010 年 4 月第 1 次印刷

印 数:1~4000

定 价:39.00 元

产品编号:034966-01

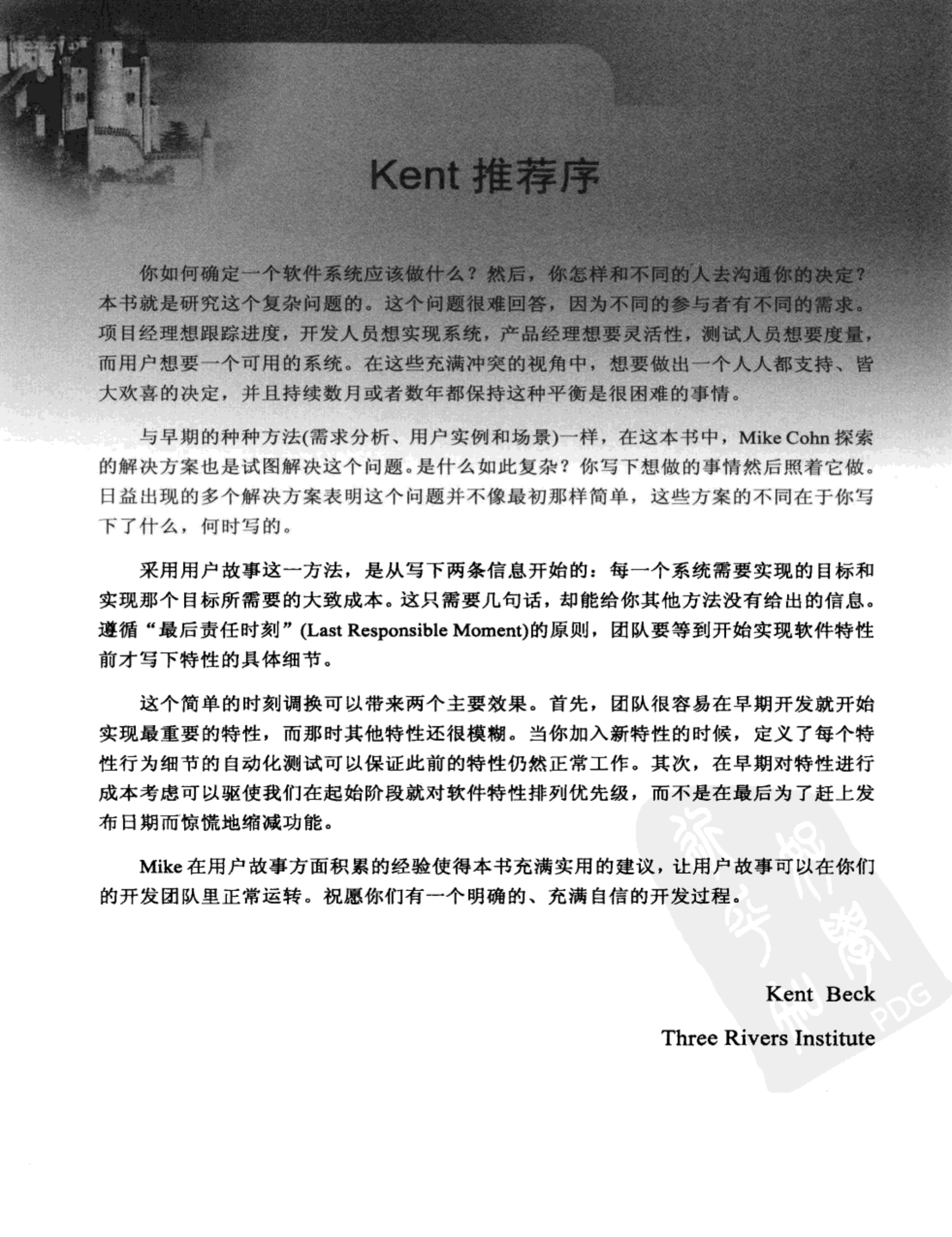
献给 Laura，因你用心阅读本书；

献给 Savannah，因你对阅读的喜爱；

献给 Delaney，因你总是设法让大家相信你已经会阅读。

有了你们这样的读者，写作变得如此轻松愉快。





Kent 推荐序

你如何确定一个软件系统应该做什么？然后，你怎样和不同的人去沟通你的决定？本书就是研究这个复杂问题的。这个问题很难回答，因为不同的参与者有不同的需求。项目经理想跟踪进度，开发人员想实现系统，产品经理想要灵活性，测试人员想要度量，而用户想要一个可用的系统。在这些充满冲突的视角中，想要做出一个人人都支持、皆大欢喜的决定，并且持续数月或者数年都保持这种平衡是很困难的事情。

与早期的种种方法(需求分析、用户实例和场景)一样，在这本书中，Mike Cohn 探索的解决方案也是试图解决这个问题。是什么如此复杂？你写下想做的事情然后照着它做。日益出现的多个解决方案表明这个问题并不像最初那样简单，这些方案的不同在于你写下了什么，何时写的。

采用用户故事这一方法，是从写下两条信息开始的：每一个系统需要实现的目标和实现那个目标所需要的大致成本。这只需要几句话，却能给你其他方法没有给出的信息。遵循“最后责任时刻”(Last Responsible Moment)的原则，团队要等到开始实现软件特性前才写下特性的具体细节。

这个简单的时刻调换可以带来两个主要效果。首先，团队很容易在早期开发就开始实现最重要的特性，而那时其他特性还很模糊。当你加入新特性的时候，定义了每个特性行为细节的自动化测试可以保证此前的特性仍然正常工作。其次，在早期对特性进行成本考虑可以驱使我们起始阶段就对软件特性排列优先级，而不是在最后为了赶上发布日期而惊慌地缩减功能。

Mike 在用户故事方面积累的经验使得本书充满实用的建议，让用户故事可以在你们的开发团队里正常运转。祝愿你们有一个明确的、充满自信的开发过程。

Kent Beck

Three Rivers Institute



乐在用户故事与敏捷方法

软件开发是充满挑战的探险。行业文献所记载的事实及实战经验告诉我们，这些挑战常常导致痛苦的结果：失败的项目。任何参与过软件开发的人毫无疑问都碰到过一次或多次的失败。我自己的经历也是确凿的见证。

虽然软件开发项目失败的原因是方方面面的，但是软件需求确实被识别为最常见的痛苦根源。不论是行业的研究，还是权威人士的讨论，或者是我自身的经历，都指向此环节。针对此挑战，多年以来行业中的解决办法是使用越来越冗长的文档，尝试用精确的语言来记录越来越细化、越来越具体的所谓全面的需求。制定标准的各个组织提供各种的文件模板和相应的语言书写规范让大家来清楚地定义需求。数不尽的书籍和文章论述怎样用详细的文档来描述需求。不知道多少森林因为各个组织产出堆积如山的需求文档而被砍伐。即便如此，失败的项目仍然持续不断地出现而且大家仍然发现需求依然是主要问题所在。为什么会这样？

或许使用固化呆板的文档及精确的语言的想法是错误的方向？问题的出路是否不在于复杂，而在于简单？解决方法是否不在于尽力在前期定义出细节，而在于及时响应，开发随需而动(just in time)？是否不在于文档，而在于密切的交流？

Mike Cohn 的这本 *User Stories Applied* 英文版是这一领域的奠基之作。现在终于有此书的中文版供中国广大的敏捷实践者参考。此书的译者们是国内敏捷界被认可的资深人士，确保了翻译的质量和准确性。

通过捕获一目了然的格式一致的用户故事，我们掌握了足够的信息可以继续前行。我们给用户故事排优先级，并且先开发最重要的。我们按需及时展开，通过交谈获取所需要的细节。我们产出具有真正价值的软件，成果可以被审核和验证甚至交付使用。然后我们继续开发后续最重要的用户故事。

这是否行得通？

是的！我最近在年终派对上有机会与我的团队反思了这个话题。在这个充满挑战、项目失败乃家常便饭的行业中，我们的组织有理由为我们的成果庆祝。因为在过去两年


半时间中，我们交付了 20 多个项目而没有一个失败。当然，需求只是项目成功的一个因素，但是我们确实深深体会到敏捷需求管理方式减轻了挑战无处不在的软件开发探险过程中的痛苦。

读好这本书。遵循它的原则。在此希望读者也能够享受到敏捷的好处。

Vernon Stinebaker(史文林)

Scrum 及 Agile Evangelist Perficient 技术总监

2010 年春于杭州



使用用户故事，不仅仅是为了快

如果去 Amazon.com 搜索 “user story” 这个关键字，你会惊奇地发现：*User Stories Applied*——这是唯一一本以用户故事为主题的书籍。随意翻开一页，你都可以从中发现有关用户故事乃至敏捷的真知灼见，无怪乎读者会称之为“用户故事圣经”。在“圣经”出现之前，我们又是如何应对需求的？

数年之前，每当我接手一个项目的时候，总会有几大本厚厚的文档扔在面前，不外乎需求规格说明、概要设计和详细设计。尤其是需求规格说明，拿起来翻开一看，那些格式化的语言就变成了世界上最好的催眠曲。读尚且如此，遑论写乎？人的大脑同时处理事物的能力是有限的，写这种正式的需求规格说明，既要思考内容是否表述客户的真实意图，还要想着符合公司对于格式、用词等等等方面的要求，怎能不心生厌惧？更何况大脑里还总回响着一个问题：“这东西写出来，能有人认真看么？”在 Wikipedia 的 User Story 页面上，这样讲述使用用户故事的目的：以更快的速度、更少的消耗应对现实世界需求的快速变化。高速互联网时代，更是如此。你还在吭哧吭哧地写需求规格说明，竞争对手的系统很可能都已经上线试运行了。

使用用户故事，不仅仅是为了快。

从大脑认知的角度来看，正式的规约说明，主要以格式化的文字为表现形式。而我们的人脑对此是很难提起兴趣的。O’ Reilly 有一个 Head First 系列广为人知，并在市场上热卖，这套书就是在这方面做出了突破。在每一本 Head First 书籍前，都有同样几页图文并茂的说明，介绍我们大脑的认知方式。面对同样一个主题，只有通过多种不同方式、不同活动的刺激，大脑才能深刻理解并记忆。显然，单一的需求规格说明无法做到这一点。回过头来看用户故事，著名的极限编程创始人之一 Ron Jeffries 提出了“3C”原则：Card、Conversation 和 Confirmation。使用卡片记录用户故事，一方面可以隐藏低层细节，另一方面也方便各方人员在白板上将其移来摆去，以整体图形的方式将与客户需求有关的内容深深印在团队的脑海中，更不用说这样给项目规划带来的好处。对话，是为了促使团队与客户之间的沟通，让大家谈论需求，大声说出来，这种活动也调动了大脑的不同区域，让人们能把相关内容学得更快，记得更牢，同时还可促进团队和客户之

间的沟通，加强人际联系，何乐而不为？用户故事的确认环节，则是以反复的方式，与用户确认某个具体使用场景中的关键细节，从而不会导致遗漏。


从软件开发的角度的入手，使用用户故事，从用户角度出发描述功能，这让我们可以站在最终用户的立场考虑问题，避免程序员的自行其是。同时还能促使团队按功能特性实现需求，而不是按架构层次，这样可以降低系统开发进入后期出现整体风险的可能，带来更大的灵活性。

不过，我想指出的是：用户故事、规划会议等类似系列非技术实践，实施起来可能并不复杂，但是必须要结合 TDD、持续集成、重构等技术实践，否则要想产生高质量的代码就是空谈，由此而完成的软件产品或是项目也必将成为沙滩上的城堡，连一个小小的浪头都无法抵挡，一瞬间即可坍塌。这两三年里，国内的敏捷声势一浪高过一浪，可大家更多讨论的也都是 Scrum 这个管理概念框架，如何减少技术债务、如何做好单元测试等方面的话题可谓寥寥。倘若写代码的基本功不扎实，长此以往，出现问题的时候，就会有人跳出来归罪于敏捷。我们作为软件开发从业人员，对此不可轻视。

回到本书，虽然它第一次正式出版是在 6 年之前，然而其中的内容却绝不过时。更值得指出的是，本书的译者团队本着快速迭代、频繁沟通的原则，以敏捷的方式完成了翻译，这个过程中，他们甚至还使用了持续集成！作为 InfoQ 中文站敏捷社区的首席编辑，我希望他们能尽快分享自己在“敏捷翻译”方面的经验，让更多的人将水平更高、质量更好的敏捷乃至技术内容介绍到国内，推动国内软件开发行业的进一步发展。

郑 柯

InfoQ 中文站敏捷社区首席编辑



译者简序：用敏捷的方式翻译敏捷经典

数年前，Mike Cohn 写了这本 *User Stories Applied: For Agile Software Development*，而我从 2007 年才真正开始接触敏捷，没想到在 2009 年我竟有机会能够参与翻译这本有关用户故事的经典著作，我感到十分的荣幸。

敏捷开发近些年在国内软件开发公司中十分流行，因为它为软件开发指引了一个方向。而用户故事是敏捷实践中一个十分重要的环节。它能帮助我们高效地收集客户真正的需求。软件开发都起始于需求收集与分析。如果一开始需求都弄错了，软件的成功也就无从谈起。同时，用户故事带来了一个十分重要的作用，即高效沟通，不论是开发团队与客户的沟通，还是团队内部成员之间的沟通。沟通使客户和团队成员都朝同一个方向前进，这意味着更少的错误，更少的浪费、风险和成本。用户故事还是敏捷计划与估算的重要基础。

我十分有幸能在 Irdeto BSS 进行敏捷开发。在这里，我们使用 Scrum，结合 XP 进行开发。用户故事自然是不可或缺的。在这里，每个团队都有一个白板，上面贴有一些卡片。它们是这个 Sprint 团队计划完成的用户故事以及这些故事划分的任务。这些用户故事卡片概要描述了需求，形如“作为(角色)，我想要(功能)，以此(商业价值)”，有时上面还附着另一张卡片写着验收条件。在做计划和开发的时候，团队可以拿着这个用户故事讨论故事细节，故事如何才算完成，等等，正如 Ron Jeffries 所描述的 3C：Card(卡片)，Conversation(交谈)和 Confirmation(确认)。

用户故事从始至终贯穿于整个开发流程。首先产品负责人根据收集来的需求编写用户故事，放入产品 Backlog 中。在 Sprint 计划会议中，团队成员讨论其中的一些用户故事，细化故事细节，确定验收标准，使用 Planning Poker(计划扑克)估算故事点。然后，将故事分成一些小的任务，并估算工作时间。最后，将故事放入 Sprint Backlog 中，并按优先级排序。Sprint 开始时，故事卡片和任务卡片都放在白板的 To Do 栏，团队成员按故事的优先级挑选任务，将任务卡片挪到 Doing 栏。任务完成后，将任务卡片挪到 Done 栏。团队尽可能地先完成优先级高的故事。在故事开发的初始阶段，测试人员和产品负



责人一起确认测试用例。故事的任务都完成后，产品负责人验收并确认故事已完成，将故事卡片挪到 Done 栏中。如此完成整个 Sprint 的所有故事。Sprint 结束时，团队还要将完成的故事演示给利益相关人、其他产品负责人和团队等。这样，每个 Sprint 团队都会通过完成一系列的用户故事来向客户输出商业价值。

这次翻译我们一共四个人参与，我和石永超主要负责翻译，滕振宇和李国彪主要负责审核。前期的第一个月，我们几乎没有什么太大的进展。我们讨论过后，发现这次翻译其实是一个具有固定交付时间、固定范围(21 章和两个附录)且依赖虚拟兼职开发团队的项目，包含开发(翻译)和测试(审核)等工作，何不用敏捷的方式来开展？于是，第二个月开始，我们使用敏捷的思想来指导翻译工作。我们首先把每一章当作一个用户故事，每个故事估算一个故事点(这个是我们做的不足的地方，一个故事点显然不能准确反映各章的不同篇幅)。我们以一个星期为一个迭代，用一份 Excel 文档作电子白板。每个周末固定时间用 Skype 开一次网上语音会议。如同 Scrum 的每日例会一样，大家回答三个问题：这个星期我做了什么，下个星期准备做什么，有什么困难。然后讨论大家遇到的一些翻译难点，统一一些术语的翻译。大家在完成每个星期的翻译工作的同时，必须及时简单地更新 Excel 中故事的状态。这样一来，每个人都能及时知道每一章的进度，哪一章可以审核，哪一章没有人翻译，可以任领。同时，Excel 中还有燃尽图，告诉我们离目标还有多远，是否需要调整。如此，这份 Excel 文档就成为我们的另一个沟通工具。另外，我们还有十分重要的工具，如同我们的软件项目一样，对于项目中的所有文件(包括电子白板和术语表)，我们使用版本管理工具 Subversion 和持续集成工具 CruiseControl。我们每次签入，持续集成工具都会立即发一封邮件通知大家有新的改动，邮件包含有签入的描述，这样可以提醒大家某一章翻译完了，应该审核，等等^①。这样，从第二个月开始，我们的翻译工作就始终保持稳定的进度，团队通过更多更频繁的回馈不断学习成长，持续改善译稿和翻译过程，最终按时交付了翻译稿。

这次翻译成为我们软件开发之外的一次敏捷实践，获益良多。同样，我们也希望能够让更多的人了解敏捷，让更多从事软件开发的人和其他行业的人从中获益。翻译这本书也希望能传播敏捷思想与方法尽一份力。让更多的人了解用户故事，使用用户故事，带来更多成功的项目。

① 其实，这远没有达到持续集成的终极效果，如果有时间，可以开发一个小程序，将现有各章的 Word 文档合并成一个最终可交付的 Word 文档，甚至做一些简单的关键字检查等单元测试^②。

译者简序：用敏捷的方式翻译敏捷经典

在此感谢一起翻译的伙伴们：李国彪、滕振宇和石永超。感谢你们让我能够获得参与翻译此书的机会，我从中学习了很多很多，不仅仅是对用户故事更深入的了解，还有在这次翻译过程中对敏捷更全面、深刻的理解。

张博超

译者团队代表

2010 年春于上海



进入水中。小孩子大概只有两三岁，她先用脚趾头试了下水温，告诉父亲说“水要暖点”(make it warmer)。父亲把手放入水中，惊奇地发现，水并不冷，水已经比他女儿习惯的水温更热了。

父亲思考了一下孩子的要求，发现他们的沟通出现了问题，相同的词代表不同的意思。孩子的要求“水要暖点”对任何大人的理解都和“提高水温”是一样的。然而对孩子而言，“水要暖点”意思却是“让水温更接近我认为的暖的温度”。

语句，尤其是写在书面上的时候，对于表达像软件这么复杂的需求是比较有限的。由于它们可能被误解，所以需要与开发人员、客户和用户频繁沟通。用户故事提供了一个方法，让我们可以写下我们不会遗忘且我们可以估算和计划的，同时还鼓励沟通。

读完本书第 I 部分后，你即可改变以前总是严谨地记录下每个需求细节的方式。读完本书，你将知道在实现故事驱动流程的所有必要信息。本书由 4 部分和两个附录组成。

第 I 部分“起步” 描述用户故事编写须知。用户故事的目的之一是让大家交谈而不是写。第 I 部分的目的是让你尽快开始交谈。第 1 章概要介绍什么是用户故事，如何使用故事。接下来 9 章详细介绍编写用户故事，通过用户角色建模收集故事，在不能直接访问真实的最终用户时编写故事，测试用户故事。第 I 部分结束时，用一章的篇幅介绍用户故事改进指南。

第 II 部分“估算和计划” 有了一系列用户故事后，首先要做的是回答“要花多长时间来开发？”第 II 部分的各章全面介绍如何用故事点估算故事，如何做一个 3~6 个月的发布计划，具体如何为即将到来的一轮迭代做计划，如何测量进度和评估项目是否如期进行。

第 III 部分“经常讨论的话题” 第 III 部分开始讲述故事与用例，软件需求说明和交互设计场景之间的区别。紧接着的各章介绍用户故事特有的优势，如何发现错误，如何在敏捷过程 Scrum 中使用故事。第 III 部分的最后一章研究了一些小问题，例如故事写在纸质卡片上还是记录在软件系统中，如何处理非功能性需求。

第 IV 部分“一个完整的实例” 用一个扩充的实例做一个综述。如果我们认为开发人员能通过故事充分理解用户的需求，那么用一个扩展的案例展示用户故事的所有方面来总结本书是非常重要的。

第 V 部分“附录” 用户故事源自极限编程。虽然读本书不需要熟悉极限编程，不过，附录 A 仍然大致介绍了极限编程。附录 B 则包含每章问题的答案。



致 谢

本书受益于众多审阅者的评论。我需要特别感谢 Marco Abis, Dave Astels, Steve Bannerman, Steve Berczuk, Lyn Bain, Dan Brown, Laura Cohn, Ron Crocker, Ward Cunningham, Rachel Davies, Robert Ellsworth, Doris Ford, John Gilman, Sven Gorts, Deb Hartmann, Chris Leslie, Chin Keong Ling, Philip, Keith Ray, Michele Sliger, Jeff Tatelman, Anko Tijman, Trond Wingard, Jason Yip 以及其他几位匿名的审阅者。

我向本书正式的审阅者致以诚挚的感谢: Ron Jeffries, Tom Poppendieck 及 Bill Wake。Ron 使我保持诚实和敏捷。Tom 让我看得到我原来忽视的一些想法。而 Bill 则使我不会偏离方向以及与我分享其 INVEST 缩写词模型。我很骄傲能与这三位在此项目中共事, 因为他们任何一位提出的很多建议都对改善本书做出了不可估量的贡献。

我也要感谢 Lisa Crispin, *Testing Extreme Programming* 一书的作者。她鼓励我写这本书, 并告诉我她与 Addison-Wesley 出版社的良好合作经历。没有她的鼓舞, 我可能现在都还没有动笔。

在过去 9 年中, 我经常就我所知的东西与 Tod Golding 进行争论。其实我们的共识远远超过我们之间的争执。不管怎样, 我常常从这些争论中学到东西。感谢 Tod 这些年来他所教我的。本书的很多内容因为我与他的那些交谈而增色不少。

感谢 Alex Viggio 和 Denver XP 小组的每一位, 让我有机会展示本书中的很多早期想法。也感谢 Mark Mosholder 和 J. B. Rainsberger, 告诉我他们怎样使用软件来替代记录卡片。也要感谢 Kenny Rubin, *Succeeding With Objects* 的作者之一(与 Adele Goldberg 合著)。他们书中显而易见的自豪感促使我在搁笔数年后重新开始写作。

衷心感谢 Fast401k 的创办人 Mark 和 Dan Gutrich。他们由衷地拥抱用户故事和 Scrum。还要感谢 Fast401k 我的每一位同事, 我们正在越来越接近实现我们的目标——成为科罗拉多州最好的团队之一。

千言万语都无法表达对我家人的感激, 因为那么多的时间我无法与他们相伴。感谢我那两个令人惊奇的女儿兼小公主, Savannah 和 Delaney。特别感谢我曼妙美丽的太太



Laura, 她因为我的繁忙而需要额外的操劳。

我还欠 Addison-Wesley 团队太多的谢意。Paul Petralia 使过程由始至终都非常愉快。Michele Vincenti 是事情的推进者。Lisa Iarkowski 无私地帮助我使用 FrameMaker。Gail Cocker 为我的图表润色不少。Nick Radhuber 在最后使所有一切努力成功地整合在一起。

最后一点, 同时也是非常重要的一点, 谢谢 Kent Beck 的真知灼见和时间, 感谢他把本书包含在他的签名系列中。



目 录

第I部分 起 步

第1章 概览.....3

什么是用户故事?4

细节在哪里?5

“必须多长时间完成?”6

客户团队7

使用故事的过程是怎么样的?7

规划发布和迭代9

什么是验收测试?11

为什么要变?12

小结13

问题14

第2章 编写故事.....15

独立的15

可讨论的16

对用户或客户有价值的18

可估计的19

小的20

 分割故事21

 合并故事23

可测试的23

小结24

开发人员职责25

客户团队职责25

问题25

第3章 用户角色建模.....27

用户角色27

角色建模的步骤28

 通过头脑风暴, 列出初始的用户

 角色集合29

 整理最初的角色集合30

 整合角色31

 提炼角色32

两个额外的技术33

 虚构人物33

 极端人物34

如果有现场用户该如何?35

小结35

开发人员职责35

客户职责35

问题36

第4章 搜集故事.....37

引出和捕捉是不合用的37

够用就行, 不是吗?38

方法38

 用户访谈39

 问卷调查41

 观察41

故事编写工作坊	42
小结	45
开发人员职责	45
客户职责	45
问题	46
第 5 章 与用户代理合作	47
用户的经理	47
开发经理	48
销售人员	49
领域专家	49
市场营销团队	50
以前的用户	50
客户	51
培训师和技术支持	52
业务分析师或系统分析师	52
与用户代理合作时，做些什么？	52
能接触到用户但访问受限时	52
实在不能接触到用户时	53
可以自己来吗？	54
设立客户团队	54
小结	55
开发人员职责	55
客户团队职责	56
问题	56
第 6 章 用户故事验收测试	57
在写代码之前写测试	58

客户定义测试	59
测试是过程的一部分	59
多少测试才算多？	59
集成测试框架	60
测试类型	61
小结	62
开发人员职责	62
客户职责	62
问题	62
第 7 章 优秀用户故事准则	63
从目标故事开始	63
切蛋糕	63
编写封闭的故事	64
卡片约束	65
根据实现时间来确定故事规模	65
不要过早涉及用户界面	66
有些需求并不是故事	67
在故事里包括用户角色	67
只为一个用户编写	68
以主动语态编写	68
由客户编写	68
向故事卡编号说“不”	68
不要忘记意图	69
小结	69
问题	70

第 II 部分 估算和计划

第 8 章 估算用户故事	73
故事点	73

以团队估算	74
估算	74

三角测量	75
使用故事点	76
如果用结对编程呢?	77
一些提醒	78
小结	79
开发人员职责	79
客户职责	79
问题	79
第 9 章 发布计划	81
我们想在什么时候发布	81
希望在发布中包含哪些功能?	82
排列故事优先级	82
混合优先级	84
高风险故事	84
根据架构需要安排优先级	85
选择迭代长度	86
从故事点到预计工期	86
初始速率	87
猜测速率	87
创建发布计划	88
小结	88
开发人员职责	89
客户职责	89

第 III 部分 经常讨论的话题

第 12 章 故事不是什么	109
用户故事不是 IEEE 830	109
用户故事不是用例	112
用户故事不是场景	115

问题	89
第 10 章 迭代计划	91
迭代计划概览	91
讨论故事	91
分解任务	92
准则	93
承担职责	94
估算并确认	94
小结	95
开发人员职责	96
客户职责	96
问题	96
第 11 章 测量并监控速率	97
测量速率	97
计划速率和实际速率	98
迭代燃尽图	100
迭代中的燃尽图	102
小结	104
开发人员职责	105
客户职责	105
问题	105

小结	117
问题	118
第 13 章 用户故事的优势	119
口头沟通	119


用户故事容易理解	121	Scrum 基础	136
用户故事的大小适合做计划	122	Scrum 团队	137
用户故事适合于迭代开发	123	产品 Backlog	137
用户故事鼓励延迟细节	124	Sprint 计划会议	138
用户故事支持随机应变的开发	124	Sprint 评审会议	140
用户故事鼓励参与性设计	125	每日 Scrum 简会	140
用户故事传播隐性知识	126	在 Scrum 中使用用户故事	142
用户故事的不足	126	Scrum 和产品 Backlog	142
小结	127	在 Sprint 计划会议中使用	
开发人员职责	127	用户故事	142
客户职责	128	在 Sprint 评审会议中使用	
问题	128	用户故事	143
第 14 章 用户故事不良征兆一览	129	在每日 Scrum 简会中使用	
故事太小	129	用户故事	143
故事互相依赖	129	一个案例	143
镀金	130	小结	144
细节太多	131	问题	145
过早考虑用户界面细节	131	第 16 章 其他话题	147
想得太远	132	处理非功能性需求	147
故事划分太过频繁	132	纸质还是软件?	148
客户很难为故事安排优先级	132	用户故事和用户界面	150
客户不愿意写用户故事, 也不愿意		保留故事	152
为故事安排优先级	133	缺陷的用户故事	154
小结	134	小结	154
开发人员职责	134	开发人员职责	155
客户职责	134	客户职责	155
问题	134	问题	155
第 15 章 Scrum 与用户故事	135		
Scrum 是迭代和递增的	135		



第 I 部分 起 步

在第 I 部分，我们首先从一个简介开始，说明什么是用户故事，如何使用它们。接着进一步呈现细节，介绍如何编写用户故事，如何利用系统的用户种类来确定故事，如何在难以接触到用户的情况下与充当用户角色的人一起工作，如何编写测试来验证故事已经成功完成。最后，给出一些有助于编写良好故事的指导原则。

完成这部分内容的学习后，你将能充分掌握如何着手识别、编写和测试自己的故事。与此同时，为学习第 II 部分的内容做好准备，学习如何利用用户故事进行估算和规划。



第 1 章 概 览

软件需求是一个沟通问题。需要新软件的人(使用或销售软件的人)必须与开发新软件的人进行交流。一个项目的成功,依赖于很多不同的信息,这些信息来自各有不同的人员:一方是客户和用户,有时还有分析人员、领域专家和其他从业务或组织视角来审视软件的人;另一方是技术团队。

一旦任何一方在沟通中把持绝对地位,项目就会遭受损失。如果业务方把持绝对地位,他们就会关注软件功能和交付日期,却很少关注开发人员是否能够同时满足这两个目标,或者开发人员是否确切地了解需求。如果开发人员把持绝对地位,技术术语就会代替业务语言,从而导致开发人员无法倾听业务方的实际需求。

我们需要一种协同工作的方法,让双方都不占绝对主导地位,共同面对感情用事和办公室政治化的资源分配问题。若资源分配问题完全落在一方,项目必定会失败。如果只让开发人员来承担这些问题(他们通常会被告知“我不关心你们怎么做,但请你们在6月份之前完成”),他们可能会牺牲质量来换取额外的特性,也可能只部分实现一个特性,或者自行做出一些本该在有客户和用户参与情况下才能做出的决定。如果只是客户和用户承担资源分配的责任,那么我们通常会在项目开始时看到一系列漫长的讨论,项目中的特性逐渐减少。之后,在最终发布软件时,只剩下很少的功能,甚至少于被减掉的功能。

至此我们已经了解到,我们不能完美地预测软件开发项目。当用户看到软件的早期版本时,他们会想出新的点子,从而改变他们的观点。由于软件的这种不可控性,大部分开发人员都会遇到众所周知的艰难时刻,估计需要多长时间才能完事儿。因为这些因素及其他一些因素,导致我们无法勾勒出一幅完美的 PERT 图^①来展示项目中所有必须完成的事情。

① 译者注: PERT 是 Program Evaluation and Review Technique 的缩写, PERT 图是计划评审图,是一种非肯定型网络计划技术。

那么，我们该怎么办？

一般情况下，我们根据手头的信息来做决策。我们经常这么干。不要在项目开始时就做一套包罗万象的决策，我们要把各个决策分散在项目过程中。为此，我们要确保有一个获取信息的过程，越早越好，越频繁越好。用户故事由此应运而生。

什么是用户故事？

用户故事描述了对用户、系统或软件购买者有价值的功能。用户故事由以下三方面组成。

- 一份书面的故事描述，用来做计划和作为提示。
- 有关故事的对话，用于具体化故事细节。
- 测试，用于表达和编档故事细节且可用于确定故事何时完成。

由于用户故事的描述信息以传统的手写方式写在纸质卡片上，所以 Ron Jeffries(2001)对这三个方面起了一个非常好的以相同英文字母开头的名字：卡片(Card)、对话(Conversation)和确认(Confirmation)。卡片可能是用户故事最明显的表现，但它并不是最重要的。Rachel Davies(2001)说过“卡片代表客户需求而不是记录需求”。这是对用户故事的最佳诠释：卡片包含故事的文字描述，然而需求细节要在“对话”中获得，并在“确认”部分得以记录。

故事卡 1.1 是一个用户故事的例子，这张故事卡来自假想的职位发布和搜索网站 BigMoneyJobs。

故事卡 1.1 写在卡片上的用户故事雏形

用户可以在网站上发布简历。

出于一致性考虑，本书其余部分的很多例子都将使用 BigMoneyJobs 网站作为例子。其他 BigMoneyJobs 示例故事可能包括下面几个。

- 用户可以搜索职位。
- 公司可以发布新职位。
- 用户可以限制浏览其简历的人。

因为用户故事代表对用户有价值的功能，所以下面的例子对于这个系统来说就不是

理想的用户故事。

- 这个软件将用 C++ 语言来编写。
- 程序将通过连接池连接到数据库。

第一个例子对 BigMoneyJobs 来说不是一个理想的用户故事,因为它的用户不需要关心系统是用什么语言编写的。然而,如果这是一个应用程序编程接口(Application Programming Interface, API),那么系统的用户(她自己也是程序员)写下“这个软件将用 C++ 编写”的用户故事则比较恰当。

第二个用户故事例子在此案例中也不是一个很好的用户故事,因为系统用户没有必要关心应用程序如何连接数据库之类的技术细节。

读完这些故事后,你可能会马上觉得“但是,等等——在我的系统中,使用连接池是需求之一!”如果这样,请停一下,关键在于故事应该以对客户有价值的方式写下来。还有很多其他方法来表示对用户有价值的故事。第2章将提供相关例子。

细节在哪里?

“用户可以搜索工作”,说起来很容易,但仅以这句话为指南着手开发和测试却是另外一回事。细节都在哪里?下面这些尚未解决的问题又该怎么办?

- 用户可以搜索哪些值?省份?城市?职位?关键字?
- 用户必须是网站的注册会员吗?
- 搜索参数可以保存吗?
- 要显示哪些与工作匹配的信息?

许多如此这般的细节可以用另外的用户故事来描述。事实上,多个小故事远远胜于一个庞大的故事。例如,整个 BigMoneyJobs 网站可描述成下面两个故事。

- 用户可以搜索工作。
- 公司可以发布工作信息。

很明显,这两个故事太庞大了,派不上太大用场。第2章将彻底解答故事大小的问题,但最开始,理想的情况是所写的故事能够让一两个程序员花半天到两周时间完成代码和测试。坦白地说,前两个故事覆盖了 BigMoneyJobs 网站的大部分,每个故事大概需要占用大多数程序员一周多时间。



如果一个故事很大，我们有时候就称之为史诗故事(Epic)。史诗故事可以分成两个或更多的小故事。例如，“用户可以搜索工作”可以分成下面几个小故事。

- 用户可以通过地区、薪水范围、职位、公司名和发布日期之类的属性来搜索工作。
- 用户可以查看搜索结果中每个工作的信息。
- 用户可以查看发布工作的公司的详细信息。

然而，我们并不需要不断地分解故事，直到有一个故事能够覆盖每一个细节。例如，“用户可以查看搜索结果中每个工作的信息”就是一个比较合理且实际的故事。我们不需要进一步将它分成下面的小故事。

- 用户可以查看工作描述。
- 用户可以查看薪水范围。
- 用户可以查看工作地点。

类似地，用户故事不需要像典型的需求文档样式那样扩充，如下所示。

4.6) 用户可以查看搜索结果中每个工作的信息。

4.6.1) 用户可以查看工作描述。

4.6.2) 用户可以查看薪水范围。

4.6.3) 用户可以查看工作地点。

与其写下这些故事细节，还不如让开发团队和客户讨论这些细节，即在这些细节变得重要时才讨论。当然，像故事卡 1.2 一样，在故事卡上加一些注解也没什么坏处。但是，讨论才是关键，而不是故事卡上的注释。开发人员和客户，谁也无法在 3 个月后指着卡片说：“但是，请看一看我之前说过的吧。”故事并不具有契约性质。达成的协议将由测试来记录，这些测试将演示故事是否被正确开发。

故事卡 1.2 带有注释的故事卡

用户可以查看搜索结果中每个工作的信息。

“必须多长时间完成？”

在我高中的文学课上，每当布置论文作业时，我总是问“必须多长时间完成？”老师素来不喜欢这个问题，但我仍然觉得这是一个不错的问题，因为它能让我知道老师的

期望。这就像理解项目用户的期望是什么一样重要。用户的期望最好以验收测试的形式被记录下来。

如果使用的是纸质片卡，可以把卡片翻过来，在背面记录下这些期望。如何测试故事的提示语句，就是这些期望，如故事卡 1.3 所示。如果使用的是电子文档，也有地方可以输入验收测试提示语句。

故事卡 1.3 故事卡背面记录着如何测试故事的一些提示语句

用空的工作描述来试试。

用很长的工作描述来试试。

用缺少薪资来试试。

用六位数的薪资来试试。

测试描述可以简短、不完整，可以在任何时候加入或者删除。写这些测试描述的目的在于传递故事的额外信息，以便于开发人员知道故事于什么时候结束。就像老师的期望对我何时写完《白鲸》(Moby Dick)的文章很有用一样，开发人员如果能了解客户的期望，他们便能知道什么时候算是完成了客户要求的功能。

客户团队

在一个理想的项目中，我们会拥有一个专职人员为开发人员的工作排列优先级，回答他们所有问题，在软件完成时使用软件，并且写下所有的故事。然而这几乎总是可欲而不可求，所以我们要建立一个客户团队。客户团队中包括确保软件满足用户需求的所有人。这意味着客户团队可以包括测试人员、产品经理、实际用户和交互设计师。

使用故事的过程是怎样的？

相较于过去的项目，使用故事的项目会有不同的感觉和节奏。使用传统的面向瀑布模型的过程会有一个书写所有需求、分析需求、设计方案、编码、最终测试的周期。在这样的过程中，客户和用户只在开始的时候参与进来写需求，在结束的时候验收软件，但用户和客户在搜集完需求后到验收之间的这段时间几乎都不参与。如今，我们已经知道这种方法是行不通的。



对于故事驱动的项目而言,最引人注目的是客户和用户在项目整个过程中全程参与。我们不希望(或者不允许)他们在项目进行时离开,不管团队是否在使用极限编程(XP,详情参考附录 A)、敏捷版本的统一过程(Agile Unified Process)、Scrum 之类的敏捷过程(参考第 15 章)或开发人员自己发展出来的故事驱动的敏捷过程。

软件的客户和最终用户应该在编写用户故事时承担着非常活跃的角色,尤其是在团队使用极限编程进行软件开发时。编写用户故事的过程最好从考虑系统的用户类别开始。例如,如果是在构建一个旅行预订网站,你可能会有诸如常旅客、假期计划者等的用户类别。客户团队应该尽量包括这些实际的用户类别。但是如果这个要求无法满足时,我们可以使用用户角色建模(关于这个主题,请参见第 3 章)。

客户为什么要编写故事?

由客户团队而不是开发人员来编写用户故事主要基于两个原因。首先,每个故事必须用商业语言来写,不是用技术术语,这样一来,客户团队可以排列故事的优先级,放入迭代和发布。其次,作为主要的产品构想者,客户团队所处的位置最适合描述产品行为。

一个项目的用户故事初稿通常是在故事编写工作坊(workshop)中写就的,但用户故事可以在项目生命周期的任何时候编写。在故事编写会上,大家集思广益,充分想象用户故事。有了可以开始工作的故事集合后,开发人员便可以估计每个故事的大小。

客户团队和开发人员一起选择迭代长度,可能一周到四周的时间。在项目进行期间将使用同样的迭代长度。在每轮迭代结束时,开发人员将负责发布完全可用的应用程序子集。客户团队在迭代期间高度参与,与开发人员谈论迭代期间正在开发的故事。在迭代期间,客户团队也会详细定义测试,并且跟开发人员一起编写运行自动化测试。此外,客户团队要确保项目能够达成交付所需产品的目标。

一旦确定迭代长度,开发人员就会估计每轮迭代中可以做多少事情。我们称之为速率(velocity)。团队的第一次速率估计可能是错误的,因为无法事先知道团队的速率。然而,我们可以用初步估计来勾勒出大致的蓝图或者发布计划,用以说明在每轮迭代中会完成哪些工作,需要多少轮迭代周期。

为了做发布计划,我们把故事排列成许多堆,每一堆代表一轮迭代。每一堆包含一定数量的故事,其中故事的估计总和不超过事先估计的速率。最高优先级的故事放在第一堆。当那一个堆放满以后,次高优先级的故事放入第二堆(代表第二轮迭代)。这样持续进行,直到已经有很多堆故事需要完成,估计会占用完这个项目的时间为止,或者直

到这些故事堆已经可以代表产品的一个新发布计划为止。(相关详情请参见第9章和第10章)

在每轮迭代开始前，客户团队可以在中途修正计划。当迭代结束后，我们可以得知开发团队的实际速率，然后用它代替估计速率进行估计。这意味着每一堆故事可能需要通过增加或者移除来进行调整。有些故事可能比预期的简单很多，所以有时开发团队想在迭代中完成另外的故事。但有些故事比预期的难，这时就要把有些工作移到下一轮迭代或者下一个发布计划中去完成。

规划发布和迭代

一个发布由一个或多轮迭代组成。发布规划指的是确定项目时间表和预期功能集合之间达到平衡。迭代规划涉及选择迭代包含的故事。客户团队和开发人员在发布和迭代规划中都要参与。

在进行发布规划时，客户团队首先从排列故事优先级开始。在排列优先级时，他们要考虑下面几点。

- 大部分用户和客户对特定特性的渴望程度。
- 小部分重要用户和客户对特定特性的渴望程度。
- 故事之间的关系。例如，“缩小”(zoom out)这个故事的优先级可能不高，但是它可能被看作是高优先级的，因为它与高优先级的另一个故事“放大”(zoom in)互补。

在许多故事的优先级上，开发人员可能与客户团队意见相左。他们可能基于技术风险方面的考虑，或者由于某个故事是其他故事的互补故事，而建议改变故事的优先级。客户团队应该倾听他们的观点，但是随后排列故事优先级时，应该坚持客户组织利益最大化的原则。

排列故事优先级时不能不考虑它们的成本。去年夏天，我的首选度假胜地是塔希提岛，然而考虑到成本，其他地方的优先级提升了。把每个故事的成本纳入优先级的考量中。故事的成本由开发者给出，每个故事用故事点来估计，故事点表明一个故事相对于其他故事的大小和复杂度。所以，一个4个点的故事成本是一个估计为2个点的故事成本的2倍。



为发布中的所有迭代分配故事后，发布计划便“浮出水面”。开发人员陈述他们预期的速率，在一轮迭代中他们认为可以完成多少个故事点。然后客户团队把故事分配到迭代中，他们要确保每轮迭代中分配的故事点数不超过开发团队预期的速率。

作为示例，假设表 1.1 列出了你们项目中所有的故事，它们按照优先级降序排列。开发团队估计每轮迭代的速率是 13 个故事点。故事在迭代中的分配如表 1.2 所示。

表 1.1 示例故事及其成本

故 事	故事点数
故事 A	3
故事 B	5
故事 C	5
故事 D	3
故事 E	1
故事 F	8
故事 G	5
故事 H	5
故事 I	5
故事 J	2

由于开发团队的预期速率是 13，没有迭代可以完成多于 13 个故事点的故事。这意味着第二轮迭代和第三轮迭代只能计划 12 个故事点。不用担心——估计很少会非常精确，这点差异不会造成什么影响，如果开发人员的速度超过计划，他们可以要求完成其他一两个小的故事。注意，在第三轮迭代中，客户团队实际上选择了故事 J 而不是优先级较高的故事 I。这是因为故事 I 需要 5 个故事点，在第三轮迭代中实现的话实际上太大了。

表 1.2 表 1.1 用户故事的发布计划

迭 代	故 事	故事点数
迭代 1	A、B、C	13
迭代 2	D、E、F	12
迭代 3	G、H、J	12
迭代 4	I	5

除了在迭代中临时跳过一个大的故事而放入一个较小的故事以外，可以把大故事分成两个小的故事。假设5个故事点的故事I可以分为故事Y(3个点)和故事Z(2个点)。故事Y包含故事I的最重要部分，现在在第三轮迭代中放入故事Y就比较合适了，如表1.3所示。关于如何以及何时分割故事，请参见第2章和第7章。

表 1.3 分割故事，做更好的发布计划

迭 代	故 事	故事点数
迭代 1	A、B、C	13
迭代 2	D、E、F	12
迭代 3	G、H、Y	13
迭代 4	J、Z	4

什么是验收测试？

验收测试用来验证实现的用户故事是否符合客户团队的期望。当一轮迭代开始时，开发人员开始编码，同时客户团队开始测试工作。取决于客户团队中成员的技术熟练程度，测试工作可以包括从故事卡背面写下测试描述开始，到将测试放入到自动化测试工具中的所有工作。客户团队中应该包含一个专业的、熟练的测试人员，由他完成这些任务中偏技术的工作。

测试应该尽早的在迭代中编写(如果你能大致猜到即将开始的迭代会产出什么，就可以在迭代开始前编写测试)。早期编写测试是非常有用的，因为这样一来，客户团队的假设和预期会更早与开发人员沟通。例如，假设写下故事“用户可以用信用卡为购物车中的物品付款”。然后在故事卡背面写下这些简单的测试描述。

- 用 Visa 信用卡、万事达信用卡(MasterCard)和美国运通卡(America Express)测试(通过)。
- 用大来卡(Diner's Club)测试(失败)。
- 用 Visa 借记卡测试(通过)。
- 用有效、无效和反面丢失卡 ID 号的信用卡测试。
- 用过期卡测试。
- 用不同购买金额测试(包括超出信用卡额度)。



这些测试捕获了这样的预期：系统将可以处理 Visa 卡、万事达卡和美国运通卡，不允许用其他卡购买。尽早把这些测试交给开发人员，客户团队不仅澄清了他们的预期，他们可能也同时提醒了程序员可能会忘记的情形。例如，程序员可能忘记了考虑过期卡的处理。注意，当程序员开始编程前，故事卡背面就写下测试描述时，可以节省程序员的时间。更多关于编写故事验收测试的内容，请参见第 6 章。

为什么要变？

现在你可能会问，为什么要变？为什么编写故事卡并进行所有这些对话？为何不继续编写需求文档或者用例(use case)？相较于其他方法，用户故事可谓优势多多。详情请参见第 13 章，但部分理由如下。

- 用户故事强调对话交流而不是书面沟通。
- 用户故事可以同时被你 and 开发人员理解。
- 用户故事的大小适合于做计划。
- 用户故事适用于迭代开发。
- 用户故事鼓励推迟考虑细节，直到你非常清楚地了解自己的真正需求。

由于用户故事的重点从文档转移到对话，所以重要决策不会写在文档里，因为很可能没有人阅读那些文档。取而代之的是，在自动化验收测试中捕获用户故事的重要方面，频繁执行验证。除此以外，我们避免在文档中出现下述含义不清的语句：

系统必须存储地址和办公电话或者移动电话。

这是什么意思？它可以被理解为系统必须存储：

(地址和办公电话)，或者移动电话

地址和(办公电话或者移动电话)

由于用户故事没有技术术语(记住，它们是由客户团队编写的)，所以开发人员和客户团队双方都能理解。

每个用户故事代表一个独立的功能；即用户在一个单一环境中可能做的事情。这便使用户故事成为一个非常合适的计划工具。你能够估计在不同发布中挪动故事顺序(优先级)的价值，这远远比估计去掉一个或多个“系统应该……”的陈述所产生的影响容易。

迭代过程是一个逐步求精的过程。开发团队首先开发系统的一小部分，知道它在某

些(或许很多)方面是不完整的或者不完善的。然后他们逐步加以相应的改进,直到产品让人满意。通过每轮迭代中增加的更多细节,软件被逐步改进。用户故事与迭代开发可以紧密结合,因为故事也是可以迭代的。对于最终需要但当前并不重要的特性,可以先写下一个大的故事(史诗故事[Epic])。准备好将大故事加入系统后,便可以提炼它,抛弃史诗故事继而用更小的、具体的故事代替它。

“故事集可以迭代”这一能力,恰恰可以佐证我们可以推迟考虑故事细节。因为假如你可以在今天写下用于占位的史诗故事,就没有必要再进一步写下系统这部分的用户故事,除非马上就要开发那些部分。推迟细节很重要,因为这样一来,我们在不确定是否真正需要某个新特性时,可以不花过多时间来考虑它。使用故事,我们不必假装可以事先知道并写下所有东西。以客户团队和开发人员的讨论为基础,不断地精炼我们的需求。

小结


- 故事卡包含对用户或者客户有价值的功能的简短描述。
- 故事卡是故事的可见部分,但客户团队和开发人员关于故事的对话更重要。
- 客户团队包括那些确保软件符合潜在用户需求的人,可以包括测试人员、产品经理、实际用户和交互设计师。
- 故事卡由客户团队编写,因为他们最了解如何表达需要实现的需求,也因为他们会后期与开发人员共同确定故事细节并安排故事的优先级顺序。
- 按照故事对客户价值来安排故事的优先级顺序。
- 将各个故事放入迭代,进行发布与迭代规划。
- 速率是开发人员可以在一轮迭代中完成的工作量。
- 放入一轮迭代的故事估计总和不能超过事先开发人员预计的速率。
- 如果故事太大以至于无法在一轮迭代中完成,可以考虑把它分成两个或更多的小故事。
- 验收测试用于验证实现的故事是否开发成符合客户团队的设想。
- 用户故事是很有意义的,因为它们强调口头交流,你和开发人员都可以理解,可以用于进行迭代计划,在迭代开发过程中能很好地工作,而且因为它们鼓励推迟细节。



问题

- 1.1 用户故事包含哪三大部分？
- 1.2 客户团队由哪些人组成？
- 1.3 以下哪些不是好的用户故事？为什么？
 - a. 用户可以在 Windows XP 和 Linux 上运行系统。
 - b. 所有绘图和图表将用第三方类库完成。
 - c. 用户可以最多撤销 50 步操作。
 - d. 软件将在 6 月 30 日发布。
 - e. 软件将用 Java 编写。
 - f. 用户可以从下拉列表框里选择她的国籍。
 - g. 系统将使用 Log4J 把所有错误信息记录到一个文件中。
 - h. 如果用户 15 分钟内没有保存文档，系统将提示用户进行保存。
 - i. 用户可以选择“导出到 XML”特性。
 - j. 用户可以导出数据到 XML 文件。
- 1.4 需求对话相对于需求文档，有哪些优势？
- 1.5 为何在故事卡背面写测试描述？





第2章 编写故事

在这一章中，我们将关注故事编写。为了构造好的故事，我们关注六个特征。一个优秀的故事应该具备以下特点：

- 独立的 (Independent)
- 可讨论的 (Negotiable)
- 对用户或客户有价值的 (Valuable to Purchasers or Users)
- 可估计的 (Estimatable)
- 小的 (Small)
- 可测试的 (Testable)

《探索极限编程》和《重构工作手册》的作者 Bill Wake，建议用英文缩写 INVEST 来代表这六个特征(Wake 2003a)。

独立的

我们要尽量避免故事间的相互依赖。在对故事排列优先级时，或者使用故事做计划时，故事间的相互依赖会导致一些问题。例如，假设客户团队已经选择了一个高优先级的故事，但它对一个低优先级的故事有依赖，这就会出现这个问题。故事间的依赖也会使得做估计变得更加困难。比如，假设我们正在开发 BigMoneyJobs 网站，现在需要编写客户公司如何对发布职位进行付费的故事。我们可以编写出这些故事。

1. 公司可以用 Visa 信用卡对发布职位进行付费。
2. 公司可以用万事达信用卡对发布职位进行付费。
3. 公司可以用美国运通卡对发布职位进行付费。

假设开发人员估计支持第一种信用卡(不考虑是哪种)需要 3 天时间，然后支持第二种和第三种信用卡各需要 1 天时间。对于这些有很高相互依赖性的故事，你不知道给每个故事估计多少时间——哪个故事应该给 3 天的估计？



出现这种依赖时，有两种方法可以绕过这种依赖。

- 将相互依赖的故事合并成一个大的、独立的故事。
- 用一个不同的方式去分割故事。

在这个案例中，把几个关于支持不同种类信用卡的故事合并成一个大的故事(“公司可以用信用卡对发布职位进行付费”)是非常可行的，因为这个合并后的故事仅需要 5 天时间。如果合并后的故事需要远远大于 5 天的时间，那么最好是找一个不同的维度来分割故事。如果这些分割后的故事需要更长的时间，那么可采用以下可行的分割方法。

1. 客户可以用一种信用卡支付。
2. 客户可以用另外两种信用卡支付。

假若你不想合并故事，但又找不到好的方法来分割它，还可以使用一个简单的方法，即在故事卡上加两个估计：当该故事将早于另一个故事实现时，一个较高的估计；当该故事将晚于另一个故事实现时，一个较低的估计。

可讨论的

故事是可以讨论的。它们不是签署好的合同或者软件必须实现的需求。故事卡是功能的简短描述，细节将在客户团队和开发团队的讨论中产生。因为故事卡的作用是提醒客户团队和开发团队在以后要进行关于需求的对话，它并不是具体的需求本身，因而它们不需要包含所有的相关细节。然而，如果我们在编写故事的时候已经知道了一些重要的细节，那么应该在故事卡上以注释的形式记录这些细节，如故事卡 2.1 所示的那样。难点在于掌握如何恰如其分地包括细节。

故事卡 2.1 是一张很好的故事卡，因为它提供了适量的信息给开发人员和客户团队进行交流。当一个开发人员开始编码和实现这个故事时，这张故事卡可以提醒她系统必须支持三种主要的信用卡(Visa、万事达信用卡和美国运通卡)，同时她也能根据卡片上的注释去询问客户团队是否已经做出了决定——关于系统是否支持 Discover 信用卡(Discover Card)。卡片上的注释可以帮助开发人员和客户继续先前没有进行(或者深入)的对话。理想状况下，不论对话的双方无论是开发人员还是客户人员是否与原来相同的，这种对话一般都很容易继续进行。把细节加入故事时，请把这一点作为指导原则。

故事卡 2.1 一张提供有额外细节的故事卡

公司可以用信用卡支付发布工作信息的费用。

备注：接受 Visa 信用卡、万事达信用卡和美国运通卡，考虑支持发现卡。

另一方面，让我们思考一下一个包含太多注释的故事，如故事卡 2.2 所示。这个故事有太多细节(“搜集信用卡的过期月份和日期”)，同时也合并了本应该成为单独故事的部分(“系统可以保存卡号以备将来使用”)。

故事卡 2.2 细节太多的故事卡

公司可以用信用卡支付发布工作信息的费用。

备注：接受 Visa 信用卡、万事达信用卡和美国运通卡，考虑支持发现卡。当支付金额超过 100 美元时，需要提供信用卡背面的 ID 号。系统可以根据卡号的前两位数字识别客户使用的是何种类型的信用卡。系统可以保持卡号以备将来使用。搜集信用卡的过期月份和日子。

处理类似故事卡 2.2 上的类似故事是很困难的。大部分人阅读这种类型的故事时，会过多地关注本不应该关注的细节。然而，在许多案例中，过早地制定细节只会带来更多的 workload。例如，有一个简单的故事如此描述“公司可以用信用卡支付发布职位的费用”，当两个开发人员一起讨论并估计这个故事时，他们知道自己的讨论多少是有些抽象的。他们不会误以为他们的讨论是明确的，或者估计是精确的，因为缺少很多细节。然而，如果在故事卡上加入故事卡 2.2 上的很多类似细节，那么关于这个故事的讨论会更容易让人觉得确定和真实。这会导致一种错觉：故事卡反映了所有细节，没有必要跟客户进行进一步的讨论。

若我们把故事卡用于提醒开发人员和客户进行关于需求的讨论，那么故事卡包含下面的信息就变得有意义。

- 一两句短语，用来提醒开发人员和客户进行对话。
- 一些注释，用以表明在对话中亟待解决的问题。

在讨论中确定的细节将变成测试。如果我们使用的是纸质卡片，那么测试可以被标



注在卡片背面；若是电子文档，也总可以标注在合适的地方。故事卡 2.3 和故事卡 2.4 展示了如何把故事卡 2.2 上的过多细节转变成测试，我们只把需要进行对话的注释留在故事卡正面。这样，故事卡正面包含故事本身和相关问题的注释；背面则包含以测试形式记录下来的细节，用于验证系统是否像预期的那样运作。

故事卡 2.3 修正后的故事卡正面，只有故事和将要讨论的问题

公司可以用信用卡支付发布工作信息的费用。

备注：我们将支持发现信用卡吗？

用户界面备注：不需要专门的字段来输入信用卡的类别卡片种类(可以从卡号的前两位数字获得该信息)。

对用户或客户有价值的

“每个故事必须对用户有价值”，这话说起来很诱人。但那是不对的。许多项目包含对用户没有意义的故事。要记住用户(软件的使用者)和客户(购买软件的人)之间的区别。假设一个开发团队正在构建一个支持大量用户的软件，可能需要在公司内 5000 台电脑上实施。像这样的客户比较关心 5000 台电脑是否在使用相同的软件配置。这就会产生例如这样的一个故事：“所有的配置信息都从一个中心读取”。用户不关心配置信息在哪里存储，但是购买者可能会关心。

故事卡 2.4 隐含测试用例的细节要和故事本身分开。这里把它们放在故事卡的背面来显示

用 Visa 信用卡、万事达信用卡和美国运通卡测试(通过)。

用大来卡(Diner's Club)测试(失败)。

用有效、无效和丢失卡 ID 号的信用卡测试。

用过期卡测试。

用高于 100 美元和低于 100 美元测试。

类似的，下面故事显示客户在购买时要考虑的价值，却不是用户所需要考虑的。

- 在整个开发过程中，开发团队要提供符合 ISO9001 标准评审的文档。
- 开发团队要按照 CMM 3 级的标准来构建软件。

应当避免那些只对开发人员有价值的故事。例如，应避免下述类似的故事。

- 所有的数据库连接要通过一个连接池。
- 所有的错误处理和记录应在一系列公共类中完成。

这些故事都在关注技术和实现细节。很可能这些故事背后的想法是好的，但是故事的编写方法应当体现对客户或用户的价值。这样使客户能方便地在开发过程中对这些故事排优先级。下面是这些故事更好的版本。

- 这个应用软件，最多 50 位用户能使用一个 5 用户的数据库许可。
- 所有的错误应以统一的方式呈现给用户并作记录。

同样，应当避免在故事中出现用户界面和技术方面的定义。例如，前面修改过的故事去掉了连接池的使用和一系列错误处理类。

保证每个故事对客户或用户有价值的最好方法是让客户来编写故事。开始时客户一般都会感觉不舒服，可能因为开发人员以前总是令他们觉得他们写下来的任何东西都有可能成为未来对他们不利的证据。（“好吧，需求文档并没有这样说……”）故事卡只是提醒他们需要之后进行需求讨论，而不是一个正式的承诺或某个功能的具体描述。大多数用户一旦接受这个概念，就会开始自己写故事了。

可估计的

对开发人员来说，能估算故事的大小(至少猜一下)，或者是把故事变为可用代码的时间量是很重要的。一般有以下 3 个原因会导致故事不可估计。

1. 开发人员缺少领域知识。
2. 开发人员缺少技术知识。
3. 故事太大了。

首先，开发人员可能缺少领域知识。如果开发人员不理解故事，他们应该和写故事的客户一起讨论。同样，没有必要理解故事的所有细节，但是开发人员需要对故事有个大概的了解。

其次，故事无法估计是因为开发人员不掌握所涉及的技术。例如，在一个 Java 项目中，我们需要提供一个 CORBA 接口给系统。团队里没有人有相关经验，所以当然无法估算这个任务。在这种情况下，可以让一个或多个开发人员去实施极限编程(XP)中所谓的探针试验(spike)。这是一个简短的试验，用于研究应用程序的某一方面。在做探针试



验的时候，开发人员不需要做十分深入的研究，只要能够大体了解足够信息来估计这个任务即可。探针试验本身总是会限定一个最大时间量(称为时间箱，timebox)，用这个时间量作为探针试验的估计。如此，一个不可估计的故事就变成了两个故事：一个快速的探针故事(用来获得足够的信息)和一个故事(真正实现功能)。

最后，如果故事太大，开发人员可能也无法进行估算。例如，对于 BigMoneyJobs 这个网站，“一个找工作的人可以找到一份工作”这个故事就太大了。我们要估计它，就要把它分解成多个更小的故事。

缺乏领域知识

举一个需要更多领域知识的例子，我们正在构建一个用于长期治疗慢性病的网站。用户(一个资深的护士)写了一个故事“新用户需要做一个糖尿病筛查(screening)”。开发人员不确定这是什么意思，这个涉及的范围可能从一个简单的网页问卷，到真正分发一套能在家检验的东西给新用户，就像公司为哮喘病患者提供的产品中所做的一样。最终，开发人员和用户一起讨论，结果发现她要的其实只是一个简单的带有一些问题的 Web 表单。

即使故事太大而不能进行可靠的估算，有时编写例如“一个找工作的人可以找到一份工作”这样的史诗故事也是很有用的，因为它们可以作为系统中有待讨论的一大块功能的占位符或者提示。如果希望暂时不细化系统的一部分功能，可以考虑写一两个史诗故事。也可以给史诗故事一个大的比较虚的估计值。

小的

像金发女郎“高莱亚”(Goldilocks)寻找一张舒适的床一样^①，有些故事可能太大，有些可能太小，有些则刚刚好。故事的大小很关键，故事太大或太小，都无助于制订计划。使用史诗故事来开展工作会很困难，因为它们通常包含多个故事。举个例子：在一个旅行预订网站，“一个用户可以计划一次度假”是一个史诗故事。对于任何旅行预订系统，计划一次度假都是非常重要的功能，包括一系列任务。史诗故事需要分成更小的故事。合适的故事大小最终取决于团队、它的容量及所使用的技术。

^① 译者注：Goldilocks 是少儿英语故事 *Goldilocks and the Three Bears* 中的女主人翁，这个单词后来用于表示“刚刚好”。

分割故事

史诗故事通常分为以下两种。

- 复合故事 (compound story)。
- 复杂故事 (complex story)。

复合故事是由多个小的故事组成的史诗故事。例如，BigMoneyJobs 系统可能包含这样的故事“用户可以发布她的简历”。在做系统初始计划时，这个故事可能是比较恰当的。但当开发人员跟客户讨论时，他们发现“发布简历”实际上包括以下几点。

- 简历可以包含教育情况、工作经历、薪资历史、出版物、演讲情况、社区服务和求职目标。
- 用户可以将简历标识为非激活状态。
- 用户也可以有多份简历。
- 用户可以修改简历。
- 用户可以删除简历。

取决于这些需要多久才能开发完成，前面每个需求可以自成一个故事。然而，也不能走向另一个极端。如果将并不那么大的故事再分解成一系列故事，那么它们可能就太小了。例如，取决于开发团队使用的是什么技术、团队的大小和技能情况，以下故事通常太小了。

- 求职者可以在简历上输入每个社区服务的日期。
- 求职者可以在简历上修改每个社区服务的日期。
- 求职者可以在简历上输入历任工作的日期范围。
- 求职者可以在简历上修改历任工作的日期范围。

通常，较好的做法是像下面这样对较小的故事进行整合。

- 用户可以创建简历，包含教育情况、工作经历、薪资历史、出版物、演讲情况、社区服务和求职目标。
- 用户可以修改简历。
- 用户可以删除简历。
- 用户可以有多份简历。
- 用户可以激活简历，也可以让简历失效。

一般有很多方法来分解一个复合故事。前面的分解方法，沿用了一种常见的分解方法，即按照“创建”、“编辑”和“删除”这些动作来分解故事。关于“创建”的故事



大小合适，因而可以被用作是一个故事时，这种方法还是不错的。另一种可行的方法是根据数据边界来分解。为此，我们将简历的各个部分当成单独的部分来增加和修改。这导致截然不同的分解结果。

- 用户可以增加、修改教育信息。
- 用户可以增加、修改工作经历信息。
- 用户可以增加、修改薪资历史信息。
- 用户可以增加、修改出版物。
- 用户可以增加、修改演讲信息。
- 用户可以增加、修改社区服务。
- 用户可以增加、修改求职目标。

.....

不同于复合故事，复杂故事是其本身就很大且不容易分解的故事。如果一个故事因为不确定性而复杂，可以将它分成两个故事：一个做调研的故事和一个开发故事。例如，假设给开发人员这样一个故事：“公司可以用信用卡支付发布职位的费用”，但没有一个开发人员曾经做过处理信用卡相关的工作。他们可以将故事这样分割：

- 调查研究网络上处理信用卡的相关技术。
- 用户可以用信用卡付费。

这个案例中，第一个故事会让一个或多个开发人员实施探针试验。这样分割复杂故事时，我们要针对做调研的故事或探针定义一个时间箱。即使不能对故事进行相对准确的估计，我们仍然非常可能定义最多需要花多少时间来进行调研。

在开发新的算法或者扩展已知算法时，复杂故事是很普遍的。一家生物技术公司中的工作团队有这样一个故事：将扩展算法加入 EM(Expectation Maximization)标准统计方法。这个复杂故事被重写成两个故事：第一个故事是调查研究扩展 EM 算法的可行性；第二个故事是将这个功能加入到产品中。在类似这样的情况下，很难估计需要多久才能完成那个做调研的故事。

考虑将探针试验放在不同的迭代里

如果可能，一种较好的做法是把做调研的故事放在一轮迭代中，另外的故事放在接下来的一轮或几轮迭代中。一般，我们只能对做调研的故事进行评估。将另一个无法估计的故事与调研故事放在同一轮迭代中，不确定性会高于平常，因为我们无法知道在那一轮迭代中能完成多少工作。

对无法估计的故事进行分解，主要好处是允许客户把调研工作从新功能中分离出来，以便对调研工作排列出优先级。如果客户只列出这个复杂故事(“将扩展算法加入到 EM 标准统计方法”)来排列优先级，且对它有一个估计，那么她可能会错误假设此新功能能实现的大致时间表，并以此来排列故事的优先级。相反，如果客户有一个调研故事(“调查研究扩展 EM 算法的可行性”)和一个功能故事(“扩展 EM 算法”)，她必须在两个选择中做出决定：在这一轮迭代中加入调研故事，不增加新的功能；或者加入一些其他故事来增加新的功能。

合并故事

有时候，故事太小了。对于太小的故事，开发人员会说她不想写下这个故事或者对它进行估计，因为那么做可能比实施该故事花的时间更长。缺陷报表和用户界面变更是典型的故事太小的例子。对于这样微不足道的小故事，在极限编程的团队中，一个比较好的方法通常是将其合并到需要半天或几天完成的故事中。给合并后的故事命名后，就可以同其他故事一样去计划实现它。

例如，假设一个项目有 5 个缺陷，还有一个更改搜索界面颜色的请求。开发人员可以估计所有涉及的工作，并把它们当成是一个单一的故事。如果选择纸质的故事卡，可以将它们用封面卡装订在一起。

可测试的

故事必须是可测试的。成功通过测试可以证明开发人员正确地实现了故事。如果故事不能被测试，开发人员怎么知道他们什么时候才算是完成了代码？

通常，不可测试的故事发生在一些非功能性的需求上，这些需求和软件有关，但不直接与功能有关。例如，下面这两个非功能性的故事。

- 用户必须觉得软件很好用。
- 用户绝不需要花很长时间等待窗口出现。

前面这两个故事都是不可测试的。无论什么时候，只要有可能，就要把测试自动化。这意味着我们要争取 99% 都自动化，而不是 10%。能自动化的测试基本上总是比你认为的要多。当产品是增量开发的，很多东西变化得很快，昨天能工作的代码，今天就会出问题。这时需要自动化测试来帮助你尽早发现这些问题。



实际情况中，总有极小部分的测试是不能进行自动化测试的。例如，“一个新用户不经过培训就能完成一般的工作流程”这个用户故事可以测试，但是不能进行自动化测试。测试这个故事需要一个人因素专家(human factor expert)设计一个测试来观察一批随机的典型新用户。这种类型的测试可能是耗时且昂贵的。但是这个故事是可测试的，而且可能适合某些产品。

故事“用户绝不需要花很长时间等待窗口出现”是不可测试的，因为它用了“绝不”，而且，“长时间等待”没有明确定义。要想演示某些东西永远不会出现是不可能的。一个更容易、更合理的目标是演示某些东西极少出现。这个故事可以改写为“在 95% 的情况下，新窗口会在 2 秒内打开。”甚至更好的做法是写一个自动化测试来验证它。

小结

- 理想情况下，故事之间是独立的。有时很难做到这一点，但我们要尽量实现这一目标。故事之间的交付顺序应该是无关的，可以任意拿一个故事来实现。
- 故事细节由用户和开发人员讨论得出。
- 故事应该很清晰地体现对用户或客户的价值。最好的做法是让客户编写故事。
- 故事可以注释一些细节，但是过多的细节会使故事难以理解，也可能给人一种开发人员和客户无需交谈的错觉。
- 给故事加上注释的最好方法是给它编写测试用例。
- 如果故事太大，复合故事和复杂故事可以分成几个小的故事。
- 如果故事太小，几个小故事可以合并成一个较大的故事。
- 故事应该是可以测试的。

开发人员职责

- 负责帮助客户编写故事，这些故事要能提醒你们同客户交谈，而不是记录详细的需求定义，故事应该对用户或者客户有价值，它们是独立的、可测的、大小合适的。
- 如果被问及实现故事所用的技术或者基础架构信息，应该使用对用户或客户有价值的术语来描述。

客户团队职责

- 负责编写故事，这些故事要能提醒你们同开发人员交谈，而不是记录详细的需求定义，它们对用户或你们自己是有价值的，它们是独立的、可测试的、大小合适的。

问题

- 2.1 以下故事中，哪些故事是好的故事，哪些是不好的故事。如果不是好故事，请说明理由？
- a. 用户能快速掌握系统。
 - b. 用户可以修改简历上的地址。
 - c. 用户可以增加、修改和删除多份简历。
 - d. 系统可以计算 n 元二次型方程分布的鞍点近似值。
 - e. 运行时错误都用同样的方法记录。
- 2.2 将这句史诗故事分解为大小合适的故事：“用户可以设置、更改职位自动搜索工具。”



寒 風 襲 擊

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒 風 襲 擊

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊，是冬季最常見的一種天氣現象。它是由於冷空氣大規模地向暖空氣地區移動，使暖空氣被迫抬升，冷空氣下沉，從而形成的。寒風襲擊時，常伴有大風、降溫、降雪等天氣現象。在農業生產中，寒風襲擊對農作物生長有不利影響，農民應採取相應的防寒措施。

寒風襲擊

PDG



第3章 用户角色建模

在很多项目中，需求分析人员只是从一个角度写用户故事，这样往往容易忽略一些需求(故事)，因为有些故事针对的并不是系统的一般用户。以用户为中心的设计(usage-centered design, Constantine and Lockwood, 1999)和交互设计(interaction design, Cooper, 1999)的规则使我们懂得，在编写故事前识别用户角色和虚构人物(persona)有很多好处。本章我们将学习用户角色、角色建模、角色映射和虚构人物，并学习利用这些初始步骤来编写更好的故事，开发更好的软件。

用户角色^①

假设我们正在开发 BigMoneyJobs 工作发布和搜索网站。这类网站会有许多不同类型的用户。当我们谈起“用户故事”时，我们说的用户是谁呢？我们是在谈论 Ashish 吗？他现在“骑驴找马”总在留意更好的工作。是 Laura 吗？她是大学应届毕业生，正在寻找第一份工作。还是 Allan？他将接受任何工作，只要那份工作可以让他搬到毛伊岛(Maui)，能够每天下午去冲浪。亦或是 Scott？他不讨厌现在的工作，但他觉得是时候换一份工作了。也许我们在谈论的是 Kindra，她六个月前被裁员了，正在找一份工作地点在美国东北部的工作。

或许我们应该考虑一下需要发布工作的公司内的用户？用户可能是 Mario，她来自人力资源部，由她发布新的工作信息。或是 Delaney，他也在人力资源部工作，但他的职责是审核简历。亦或者是 Savannah，她是独立的猎头，同时关注好工作和优秀人才。

显然，我们不能从单一的角度编写故事，让那些故事反映所有这些用户的经历、背景和目标是不现实的。Ashish，会计师，可能每月只上一次我们的网站，以保留他的选择余地。Allan，服务员，可能想创建一个过滤器。此过滤器可以在第一时间通知他毛伊

① 本章中许多关于用户角色的讨论，是基于 Larry Constantine 和 Lucy Lockwood 的成果。更多关于用户角色建模的信息，可以在他们的网站 www.foruse.com 上找到，或者参阅 *Software for User* (1999)一书。

岛上有新的工作发布。除非我们提供这个便捷功能，不然他实现不了这个想法。Kindra 可能每天花几个小时寻找工作，并不断扩大她的搜索范围。如果 Mario 和 Delaney 的公司比较大，有很多职位空缺需要填补，那么他们可能每天要在我们的网站上消耗 4 小时，甚至更多。

虽然使用软件的用户有着不同的背景、持有不同的目标，但我们仍可以把这些单独的客户进行分组，把每一类作为一种“用户角色”(User Role)。用户角色是一组属性的集合，这些属性刻画了一群人的特征以及这群人与系统可能的交互。我们可以看一下先前例子中的用户，将他们进行角色分组，如表 3.1 所示。

表 3.1 BigMoneyJobs 项目角色列表之一

类 别	姓 名
求职者	Scott
初次找工作者	Laura
裁员受害者	Kindra
工作地点搜索者	Allan
监视者	Ashish
工作发布者	Mario, Savannah
简历阅读者	Delaney, Savannah

显然，针对不同用户角色的故事之间会有些重复。求职者、初次求职者、裁员受害者、工作地点搜索者和关注者都会使用站点的工作搜索特性，但是他们使用搜索功能的方式和频率可能会不同，针对简历阅读者和工作发布者的故事也可能重复，因为这些角色的目标都是找到好的候选人。

表 3.1 并不是对 BigMoneyJobs 用户进行角色分组的唯一方式。例如，我们可以选择包含诸如兼职者、全职者和合同工等角色。在本章的余下部分，我们将学习如何制定一份角色列表，如何完善该列表，从而编写出好的故事。

角色建模的步骤

我们将使用以下步骤来识别、选择有用的用户角色集合。

- 通过头脑风暴，列出初始的用户角色集合

- 整理最初的角色集合
- 整合角色
- 提炼角色

每个步骤会在以下各部分讨论。

通过头脑风暴，列出初始的用户角色集合

为了识别用户角色，客户和开发人员(多多益善)聚集在一个房间里，房间里要有一张大桌子或一堵墙，这样他们就有地方粘贴或者固定卡片。理想情况是在项目启动时，把团队所有成员聚集在一起进行用户角色建模，但这并不是必须的。只要有一定数量的开发人员和客户一同参与，会议往往就能取得成功。

每个参与者从桌子中间堆放的记录卡中取出一叠。(即便打算用电子文档来记录用户角色，也应该从手写记录开始。)每个人先在卡片上写下角色名称，然后把它们放到桌子上，要么粘在或者钉在墙上。

放上新的角色卡片后，作者只说出新角色的名字，不做其他任何事情。在这个会议上只做头脑风暴，无需对卡片进行讨论，也不需要角色进行评估。每个人要做的只是尽量在卡片上写出自己想到的角色。不需要让大家轮流给出新的角色。想到一个新角色，就把它写到卡片上。

在头脑风暴过程中，房间里会充满书写卡片的声音，偶尔夹杂放置新卡片和朗读角色名称的声音。这样继续下去直到大家没有新的进展，并且很难再想出新的角色。尽管此时有可能还没有找到所有的角色，但其实已经很接近了。这样的头脑风暴很少会超过15分钟。

一个用户角色是一个用户

对项目的角色进行头脑风暴时，要坚持“已确认的角色代表的是单一用户”的原则。例如，对于 BigMoneyJobs 项目，可能会写下诸如“公司可以发布工作信息”的故事。然而，由于公司作为一个整体是没法使用软件的，因此如果这个故事提到代表个体的角色，这个故事就会更理想。



整理最初的角色集合

接下来需要整理这些角色。在桌子上或墙上移动卡片的位置，以表明角色之间的关系。对于有重叠的角色，把它们相应的卡片也重叠在一起。如果角色只有一点点重叠，那么卡片也只重叠一点点。如果角色完全重叠，那么卡片也完全重叠。如图 3.1 所示。

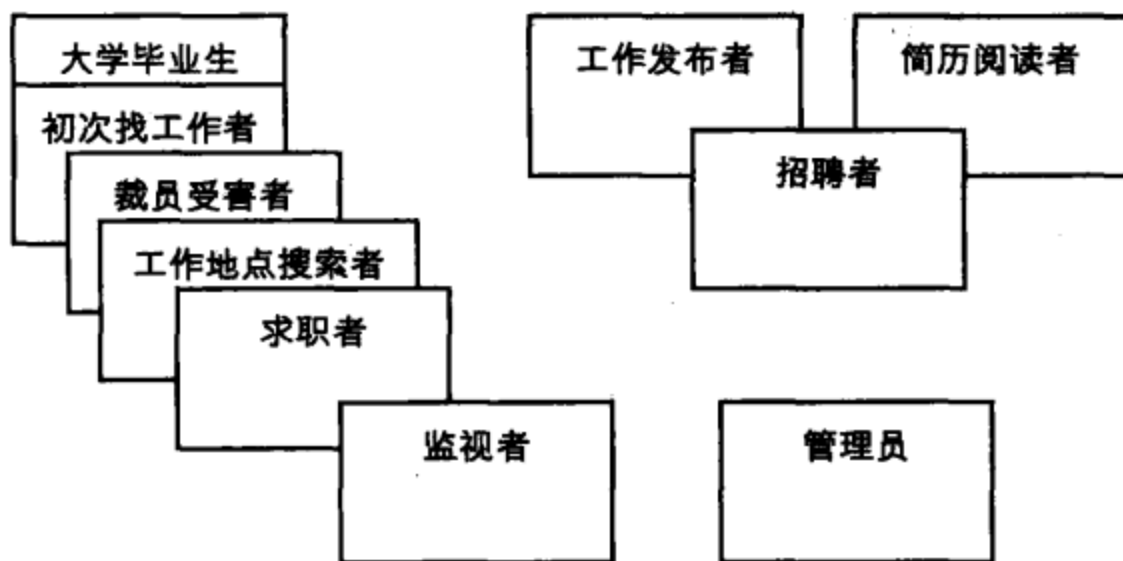


图 3.1 在桌子上整理用户角色卡片

图 3.1 显示了对于大学毕业生和初次找工作的人，他们的角色有显著的重叠。对于其他将使用工作搜索功能的人而言，它们的角色卡片之间也有较小但类似的重叠。监视者的角色卡片与其他卡片仅仅稍有重叠，因为这个角色代表的是那些对现有工作相对满意的人，但他们同时又喜欢留意好的机会。

图 3.1 右侧的角色是“工作发布者”、“招聘人员”和“简历阅读者”的角色卡片。“招聘者”这个角色，与“工作发布者”和“简历阅读者”这两个角色都有重叠，这是由于“招聘者”既会发布招聘广告，也会阅读简历。“管理员”角色也在图 3.1 中展示了，这个角色代表 BigMoneyJobs 的内部用户，他们将支持系统的运行。

系统角色

尽量坚持一个原则：用户角色定义的是人，而不是其他外部系统。如果觉得有帮助，可以偶尔确认一个非人物的系统角色(non-human user role)。然而，确认用户角色的目的是确保我们很周到地为用户考虑，我们要绝对地、积极地让用户对新系统感到满意。我们不需要为每一个可以想象得到的系统用户建立角色，但需要那些能影响项目成败的角色。由于其他外部系统很少会是我们系统的购买者，它们很少能决定我们系统的成败。

自然，事情总有例外，若觉得加入一个非人物的系统角色有助于思考系统，将它加入也未尝不可。

整合角色

在角色分组完成后，试着整合及浓缩角色。可以从完全重叠的卡片入手。首先，这些卡片的作者描述一下他们的角色名究竟代表什么，在简短的小组讨论之后，再判断这些角色是否等同。如果等同，那么这些角色要么合并成一个单一的角色(也许可以根据这两个初始的角色名取一个新的名字)，要么丢弃掉其中一张角色卡。

在图 3.1 中，“大学毕业生”和“初次找工作者”这两个角色有很大的重叠。由于任何关于“大学毕业生”这个角色的故事都与“初次找工作者”的相同，所以小组决定丢弃“大学毕业生”的角色卡片。尽管“初次找工作者”、“裁员受害者”、“工作地点搜索者”和“求职者”有显著的重叠，但小组还是决定留下它们，因为这些角色每个都代表了系统需要满足的重要方面，这些角色使用 BigMoneyJobs 网站的目标也有微妙的不同，这很重要。

看着图 3.1 的右侧时，小组认为区分“简历发布者”和“简历阅读者”没有什么价值。他们决定，“招聘者”这个角色会充分覆盖另外两个角色，所以那两个角色的卡片被丢弃。然后，小组觉得内部招聘者(为某个公司工作)和外部招聘者(为任何公司寻找合适的人选)有所不同。他们为内部招聘者和外部招聘者写了新的卡片，并将这两个角色当成是招聘者角色的特殊版本。

除了需要合并重叠的角色外，小组还应该丢弃那些对系统成功不太重要的角色卡。例如，“监视者”角色卡代表了那些只关注工作市场的人。监视者可能三年都不换工作。即便不关注那个用户角色，BigMoneyJobs 网站也可能做得很出色。他们认定最好能关注那些对公司成功更重要的角色，比如“求职者”角色和“招聘者”角色。

在团队合并完角色卡片后，在桌子上或墙上排列它们，以此展示角色之间的关系。图 3.2 展示了 BigMoneyJobs 角色卡片众多可能的排列方法中的一种。这里的通用角色，如“求职者”或“招聘者”，放在对应角色的具体版本之上。另外，卡片可以用小组期望的任何其他方式叠放或放置，以此展示出他们认为重要的关系。

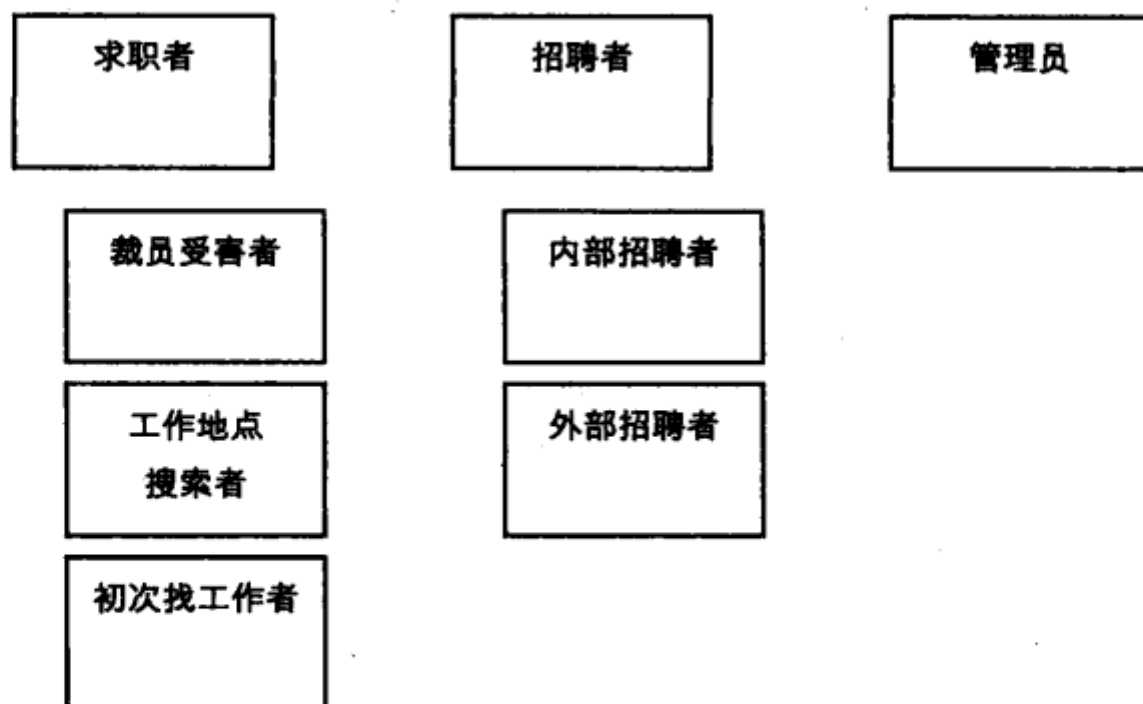


图 3.2 整合角色卡

提炼角色

一旦我们整合好角色，并且对角色之间的关系有了一个基本的了解，就有可能通过给每个角色定义一些特征来建立角色的模型。角色特征是关于同属于这一类的用户的事实或有用信息。任何可以区分这个角色的信息都可以用来做该角色的特征。这里有一些适用于任何角色建模的角色特征。

- 用户使用软件的频率。
- 用户在相关领域的知识水平。
- 用户使用计算机和软件的总体水平。
- 用户对当前正在开发的软件的熟悉程度。
- 用户使用该软件的总体目标。有些用户注重使用的便捷性，有些关注丰富的用户体验，等等。

除了这些标准的特征，应该考虑对于正在开发的软件，是否有一些对描述其用户有帮助的特征。例如，对于 BigMoneyJobs 网站，可以考虑某个用户角色是在寻找全职工作还是兼职工作。

在确定角色的有趣特征时，可以在角色卡片上写下注释。完成后，可以把角色卡挂在团队的公共区域，用来提示团队成员。图 3.3 展示了一张用户角色卡片的样例。

用户角色：内部招聘者

不是很擅长使用电脑，但是使用网络相当娴熟。不经常使用该软件，但每次使用强度大。她将阅读其他公司的招聘广告，以此选择最好的措辞来完成她们的招聘广告。使用简单很重要，但更重要的是，她学会的东西必须在几个月后能很容易地回想起来。

图 3.3 一张用户角色卡片样例

两个额外的技术

愿意的话，我们可以到此为止。到现阶段，这个团队可能已经花了一个小时(通常不需要这么长)。但是，在考虑他们系统的用户问题上，他们已经绝对比 99% 的其他软件团队多费了心思。大部分的团队确实可以到此为止。不管怎样，还有两个额外的技术值得探讨，因为它们有助于我们考虑某些系统的用户。不过，这两个技术仅适用于你觉得它们能为项目带来明显好处时。

虚构人物

识别用户角色是一个伟大的飞跃，但对于有些更为重要的用户角色，再进一步为角色创建一个虚构人物是很值得的。虚构人物是假想的用户角色代表。本章先前提到过 Mario，他为自己的公司发布工作信息。创建虚构人物需要的不只是在用户角色上加一个名字。对于虚构人物的描述应当是十分充分的，让团队中每个人都觉得他们知道这个人物。例如，Mario 可能如下描述：

Mario 在 SpeedyNetworks 的人事部门负责招聘工作，该公司是一个高端网络组件制造商。他已经在该公司工作了 6 年。Mario 有弹性的时间安排，每周周五他在家工作。Mario 对电脑相当在行，他觉得对于自己所使用的软件产品，他几乎都是超级用户。Mario 的妻子 Kim，正在斯坦福大学化学系攻读博士。由于 SpeedyNetworks 几乎一直在扩张，Mario 总是在特色优秀工程师。

假若选择为项目创建虚构人物，要注意，应该事先做好充分的市场和目标用户群调查，要确保虚构人物能够真正代表产品的目标用户。

前面的虚构人物描述，很好地给我们介绍了 Mario 的情况。如果有照片就更加生动了，应该找一幅 Mario 的照片，将它包括到虚构人物的定义中。可以从网络上获得照片，



或者你可以从杂志上剪下来。不错的虚构人物定义加上一幅照片，会让团队中每个人都对该人物有一个全面的了解。

大部分虚构人物的定义会比较长，很难在一张卡片上完整写下来，因此我建议写在一张纸上，并将它们挂在团队的公共空间里。不必为每个用户角色写下虚构人物定义。然而，你可以考虑给一两个主要的用户角色写下虚构人物定义。对于你们正在开发的系统，如果某一两个用户角色十分重要，那么那些用户角色就是需要扩展出虚构人物的候选角色。

从用户角色或虚构人物的角度描述会使故事变得更加生动。识别出用户角色，并且可能有一两个虚构人物后，就可以开始从角色和虚构人物的视角来说话，而不是宽泛的“用户”。你可以写一个“工作地点搜索者可以限定他搜索到的工作都在指定的地理区域内”的故事，而不是“用户可以限定他搜索到的工作都在指定的地理区域内”的故事。以这种方式编写的用户故事可以提醒团队想起 Allan，他正在寻找毛伊岛上的工作。使用用户角色或虚构人物的名字来编写故事并不意味着其他角色就不能执行那些故事，而是说明在讨论或实现故事时，用特定的用户角色或虚构人物来思考问题总是有一些好处的。

极端人物

Djajadiningrat 和其他合著者(2000)提出了第二种可以考虑的技术：考虑新系统的设计时，使用极端人物。他们用设计个人数字助理(PDA)掌上电脑作为例子。他们建议，不要只为一个典型的穿着考究、驾驶宝马的管理顾问做设计，系统设计师应该考虑那些有鲜明个性的用户。具体说来，作者建议为毒贩、教皇和穿梭于多个男友之间的 20 岁女子设计 PDA。

考虑极端人物很可能会让你编写出原本可能遗漏的故事。例如，很容易想象毒贩和有多个男友的女子都想要维护多份单独的时间表，以防被警察或男友看见。教皇可能没那么多保密需求，但可能想要有更大号的字。

使用极端人物可以导致新的故事产生，但很难事先确定是否应该把这些故事包含在产品中。当然，在极端人物上投入大量时间上可能是不值得的，但你可以尝试一下这个方法。至少，你们可以饶有兴致地花几分钟时间去考虑一下教皇如何使用你们的软件，这可能会带来一两个灵感。

如果有现场用户该如何？

即使有真实的用户在办公室现场，本章所述的用户角色建模技术仍然是有帮助的。与实际用户一起工作会大大提高成功交付所需要的软件的可能性。然而，即使与实际用户在一起，也无法保证有正确的用户或用户组合。

为了降低无法满足重要用户的可能性，即便有内部用户在的时候，你们还是应该对项目做一些简单的角色建模。

小结

- 大部分项目小组只考虑单一的用户类型。这会导致软件忽略原本需要的一些用户类型。
- 为了避免从单一用户的角度编写所有故事，要识别与软件交互的不同用户角色。
- 通过对每个用户角色定义相关特征，可以更清楚地看到不同角色间的不同点。
- 对于有些用户角色而言，用代表人物来描述会很有帮助。虚构人物是假想出来的用户角色代表。他们有名字，有照片，还有足够的相关细节，因为对项目成员来说，很真实。
- 对于有些应用程序，极端人物可能有助于搜集原本被遗漏的故事。

开发人员职责

- 负责参与确认用户角色和虚构人物的过程。
- 负责理解每个用户角色或虚构人物，以及它们之间的异同。
- 开发软件时，负责考虑不同的用户角色对于软件如何运行的不同偏好。
- 负责确保在识别和描述用户角色时，它们只是这个过程的工具，不应超越作为工具之外的任何用途。

客户职责

- 负责寻找用户(多多益善)，并识别恰当的用户角色。



- 负责参与识别用户角色和虚构人物的过程。
- 负责确保软件没有关注不恰当的用户。
- 在编写故事时，负责确保每个故事都能和至少一个用户角色或虚构人物联系起来。
- 开发软件时，负责考虑不同的用户角色对于软件如何运行的不同偏好。
- 负责确保在识别和描述用户角色时，它们只是这个过程中的工具，不应超越作为工具之外的任何用途。

问题

- 3.1 以易趣(eBay)为例，你能识别它有哪些用户角色吗？
- 3.2 整合前一个问题想出来的角色，展示一下如何排列这些角色卡。然后解释一下你的回答。
- 3.3 为其中最重要的用户角色编写虚构人物描述。





第4章 搜集故事

你怎么收集故事？本章将告诉你如何和用户一起工作，如何通过与他们沟通来发现故事。同时介绍各种方法的优点，怎样提出恰当的问题，从而获得用户真正的需求。

引出和捕捉是不合用的

一些需求相关的书中用到了像“引出”(Kovitz 1999; Lauesen 2002; Wiegers 1999)和“捕捉”(Jacobson, Booch 和 Rumbaugh 1999)这样的词来描述识别与确认需求的实践。这样的术语给我们一种错觉：“需求本来已经存在了，我们只要让客户给我们解释需求，然后把它们锁入一个笼子里就可以了。”很多需求并不容易想到。同样，用户并不知道所有的需求，所以不能单纯依靠引出(elicitation)。

Robertson 和 Robertson (1999)引入了拖网(trawling)这个词来描述收集需求的过程。怎样理解？要像“拖网渔船捕捞鱼”那样来收集需求。为什么用这样的比喻呢？理由如下。

首先，不同大小的网用来捕获不同大小的需求。第一遍，我们可以用大网眼的渔网捞一遍需求池，以此得到所有的大需求。通过这些大需求，形成对软件的整体感觉。接下来，用网眼稍微小一些的渔网得到中等大小的需求，暂时还不用顾及那些小需求。在这个比喻中，大小可以是对于此软件的商业价值高低或者必要性程度等。

其次，拖网表达了另一个含义：需求会像鱼一样，会成长，也可能会死亡。今天渔网可能会漏掉一个需求，因为这个需求对于系统来说不重要。但是，根据每轮迭代的反馈，系统会朝着事先不可预知的方向发展，有些需求会变得越来越重要。同样，有些曾经被认为是重要的需求，重要性可能会降低，有时甚至降低到我们认为这些需求已经无效。

第三，正如在某区域拖网捕鱼不可能捕获所有的鱼，我们也不可能捕捉到所有的需求。另外，在拖网捕捞需求的时候，也可能捞到一些废弃的货物或漂浮的残骸，他们会使需求膨胀。



最后，这个拖网捕捞需求的比喻还说明了一个重要的现实：技能也是发现需求的一个要素。一个熟练的需求分析人员(requirements trawler)知道要到哪里去找需求，而不熟练的需求分析人员只会用低效的方法或在错误的地方浪费时间。本章将学习那些让我们更有效“捞”到用户故事的方法。

够用就行，不是吗？

辨别传统规范过程和敏捷过程最简单的方法之一，是看它们搜集需求的方式：传统规范过程的特征是它过分强调在项目早期正确地获取并写出所有的需求。与此不同的是，敏捷项目则承认没有一种理想的方法可以在一个单一阶段获取到所有的用户故事。敏捷过程也承认用户故事有一个时间维度：随着时间的推移以及先前迭代中加入产品的故事，一个故事的相关性(relevance)会有所变化。

然而，即使我们承认无法为一个项目编写出所有的故事，我们还是应该在早期尝试编写我们可以编写的故事，即便许多故事还只能停留在十分笼统的阶段。使用故事的好处之一就是可以用不同的详尽程度来编写。我们可以写下“用户可以搜索工作”这样的故事，无论是作为一个占位符，亦或此时我们只了解到这个程度，这个故事都是恰当的(placeholder)。我们可以在今后将它演进为更小的、更有用的多个故事。因此，对应用程序的大部分功能编写故事是非常容易的，其工作量少于用其他方法搜集需求。

这并不是鼓励大家在开始一个新项目时花3个月时间编写用户故事。相反，它要求大家展望未来一个发布(大概是3至6个月)的时间，故事的发布时间越往后，我们越不需要编写得那么详细。例如，如果客户或用户说过他们“很可能在此次发布中想要报表功能”，那么简单地在一张卡片上写下“用户可以运行报表”。但请就此打住：先不用决定他们是否需要配置自己的报表，报表是否以HTML格式输出，或者是否可以保存报表。

类似地，在项目启动前，对应用程序的大小有一个大致的了解往往是很重要的。通常在对项目拨款以及批准启动它之前，有必要粗略地了解一下项目的成本和它能够带来的效益。为了获知这些问题的答案，需要对项目包含的故事进行初步的深谋远虑。

方法

因为故事会随着项目的进展而演进，所以我们需要一些可以反复使用的方法来搜集

故事。我们使用的技巧必须是足够轻量的，并且不咄咄逼人，可以持续地、或多或少地应用于故事搜集。以下是创建故事最有用的一些方法：

- 用户访谈
- 问卷调查
- 观察
- 故事编写工作坊

这些方法有很多是传统业务分析师所使用的。对于配备有业务分析师的项目，应该充分利用他们以“拖网捕鱼”的方式来搜集故事。

这些方法将在接下来的小节中逐个介绍。

用户访谈

用户访谈是许多团队用来获取故事的默认方法，很可能这也是你想使用的方法。访谈成功的关键之一是选择正确的受访者。如同第5章讨论的那样，“与用户代理一起工作”，有许多用户代理可以做访谈。但是显然，只要有可能，就应该访问真实用户。还应该访问担任不同角色的用户。

有一次，一位用户走进我的办公室对我说：“你们的确开发了我所要求的应用，但它并不是我真正想要的”。这件事让我明白一点：只询问用户“你们需要什么”是不够的，大部分用户不太善于理解，更难以表达他们的真实需求。

我曾经和一个团队一起工作，他们要开发一个调查速递软件。每个调查会通过电话、电子邮件和交互式语音应答系统来进行。不同类型的用户会使用不同种类的调查。这些调查非常复杂：对于一组问题的回答将决定下一个问题是什么。用户需要一种输入调查的方法，他们向开发团队演示了一些例子，建议用一种复杂的迷你型语言来确定问题。这种完全基于文本的方法，对于开发人员来说增加了不必要的复杂度。开发人员向用户展示了他们可以通过可视化的图标拖拽来创建调查，不同的图标代表了调查中不同类型的问题。之后用户放弃了他们的迷你型语言，并和开发人员一起开发可视化的调查设计工具。这说明一点：仅仅因为这些问题是由用户提出就认为只有用户才有资格提出解决方案，这种观点是不对的。

开放式问题和背景无关问题

要想获取用户的本质需求，最好的技巧是提问。我曾经与一个项目小组一起工作，他们在把应用程序开发成 Web 程序和开发成更传统的平台相关程序之间举棋不定。基于



浏览器的程序更容易部署，而且培训成本比较低；而与平台相关的客户端程序则更加健壮，两者如何选择，他们在进行着激烈的思想斗争。用户一定会喜欢浏览器的优势，但他们也重视特定平台客户端程序丰富的用户体验。

有人建议，询问一下目标用户的喜好。由于这个产品是对上一代产品的重新编写，所以市场部同意与目前的产品用户代表取得联系。询问每个接受调查的用户：“你们想在浏览器上运行新的应用程序吗？”

这个问题就像是去你最喜欢的餐馆，服务员问你是否想要免费餐。当然，你会回答是的。

同样，接受调查的用户会回答说他们喜欢在浏览器里运行新版本的软件。

市场部的人犯了一个错误，他们询问了一个封闭式问题，没有提供足够的细节以让对方更好地回答。这个问题假设了接受调查的用户知道浏览器方案和未明确说明的代替方案之间的优缺点。这个提问更好的版本是：

你想我们新的应用程序在浏览器里运行，而不是本地窗口程序吗？即使这意味着性能会有所减弱，总体上用户体验会差一些，交互也会少一些。

这个提问仍然有问题，因为它是封闭式的。调查对象只能回答简单的是或否，没有余地去回答其他的东西。询问开放式的问题要好得多，这可以让调查对象表达更深入的意见。比如，“为了让我们下一代产品运行在浏览器里，你愿意舍弃什么？”针对这个问题的用户回答可能有很多种。无论是哪种回答，对我们来说都会更有意义。

同样重要的是要提背景无关的问题，这种提问没有暗含答案或喜好。比如，你不应该问“你不会愿意为了让软件在浏览器里运行，而牺牲性能和丰富的用户体验，对吗？”很明显，我们知道大部分人会怎么回答。

类似的，不要问“搜索速度需要多快？”，而要问“需要怎样的性能？”或者“性能在应用程序中的某些部分更为重要吗？”第一个提问不是一个背景无关的问题，因为它暗含了有一个需求是关于搜索性能的。要么没有人这样问过用户，一旦问了，她的回答很可能是猜想出来的(但用户不太可能说自己的回答是猜想出来的)。

某些时候，需要使用非常具体的问题。然而，最好从背景无关的问题开始提问，这

样就有可能从用户那里获得更多样化的回答。如果从非常具体的问题开始，则很可能漏掉很多故事。

问卷调查

问卷调查会是一种有效的方法，有助于收集已有故事的相关信息。若你有一个庞大的用户群，那么问卷就是收集有关故事优先级信息的好方法。在需要得到大量用户关于某些具体问题的回答时，问卷是非常有用的。

然而，问卷不适合作为拖网捕获新故事的主要方法。使用静态的问卷不利于跟进后续问题。同样，不能像通过谈话方式那样，很方便地立即对用户的有趣想法进行深入探讨。

举一个使用问卷的例子，你可以调查用户今天使用软件功能的频率，以及为什么有些功能用得比较少。这样就可以将那些使用频率较高的故事优先级调高。另一个例子，“什么新功能你最想看到”这样的问卷问题价值就比较低。如果给用户一些选择，就可能错过几个自己从没想到过的关键功能；而如果让用户自己用自由格式提供回答，就很难归纳多个回答。

鉴于单向沟通的既有特点和时间滞后，我不建议在捕捞故事时使用问卷调查。假若想从已有的广泛用户群搜集信息，而且愿意等待一个或多轮迭代来分析收集到的信息，则可以使用问卷，但不要把它作为捕捞故事的主要方法。

观察

观察用户实际使用软件的情况，这个方法非常不错。每次我看到有人使用我的软件，我都会获得很多提高用户体验或生产力的想法。不幸的是，能观察用户使用情况的机会少之甚少，除非是为内部客户开发。太多商业产品开发采用的方法都是猜测用户需求。因此，如果有机会观察用户使用软件的情况，千万不要错过。这种机会可以让你快速直接地从用户那里获得反馈，从而可以更早、更频繁地发布软件。

曾经有一个公司的用户是在呼叫中心工作的护士。他们负责回答来电咨询的医学问题。护士指出他们需要一个文本框，在通话结束时，能够用它来记录通话结果。软件最初的版本中，在接电话时有一个覆盖屏幕的大文本框。然而，该版本发布后，每一个开发成员花了一天时间观察用户。他们发现用户在大文本框中输入的内容其实可以让系统来跟踪记录。通过观察，开发人员发现真正的需求是系统应记录用户在使用软件过程



中所作的决定。后来，开发人员用一个日志功能替换了文本框，该功能记录了护士所有的搜索和选择的建议。这个真正的需求，记录所有给来电者的解答。这个真正的需求由于最初护士对需求的描述而变得很不清晰，只有通过观察才能发现。

故事编写工作坊

故事编写工作坊是开发人员、用户、产品客户和其他对编写故事有帮助的人共同参加的会议。在工作坊期间，参与人员编写尽可能多的故事。此时不对故事排优先级，以后客户有机会排优先级。我认为，故事编写工作坊是快速捕捞故事最有效的方法。至少，我建议在开始每个计划发布前举办故事编写工作坊。在通往软件发布的路上，总是可以举办更多的故事编写工作坊，但这通常没有必要。

正确举办故事编写工作坊可以非常快速地编写大量故事。根据我的经验，良好的故事编写工作坊结合了头脑风暴的最佳要素和简单原型法。可以把一个简单原型画在纸上，笔记卡上，白板上，并画出软件内部高层之间的交互。在工作坊中，随着参与者对用户在使用软件过程中可能要做的事情进行头脑风暴，不断地构建原型。这并不是像传统原型法或联合应用设计中，要确定实际界面和字段，只是为了从概念上确定工作流。图 4.1 展示了 BigMoneyJob 网站的简单原型。

每一个方框代表着网站的一个新的组件。方框中带有下划线的文字是组件的标题。标题下面是组件要做的和包含的列表。方框间的箭头表示组件间的链接。对于一个网站，组件可能是一个新的页面或当前页面的一块区域。所以，一个链接意味着弹出一个新页面或者在同一个页面上显示信息。比如，搜索工作可能是一个页面或者是首页上的一块区域。这个区别不是那么重要，客户和开发人员在这之后有大量的时间来讨论类似的问题。

开始画原型前，首先要决定从哪种用户角色开始。需要用每个角色来重复这个过程，不论什么顺序。然后，画一个空的方框，告诉参与者这是软件的主界面，询问他们当前这个用户角色能在这个界面做什么。即使你现在还不知道主界面是什么，有什么用，这也没有关系。参与者会想出角色会做什么。对角色做的每一件事，画一条线指向一个新的方框，然后写一个故事。

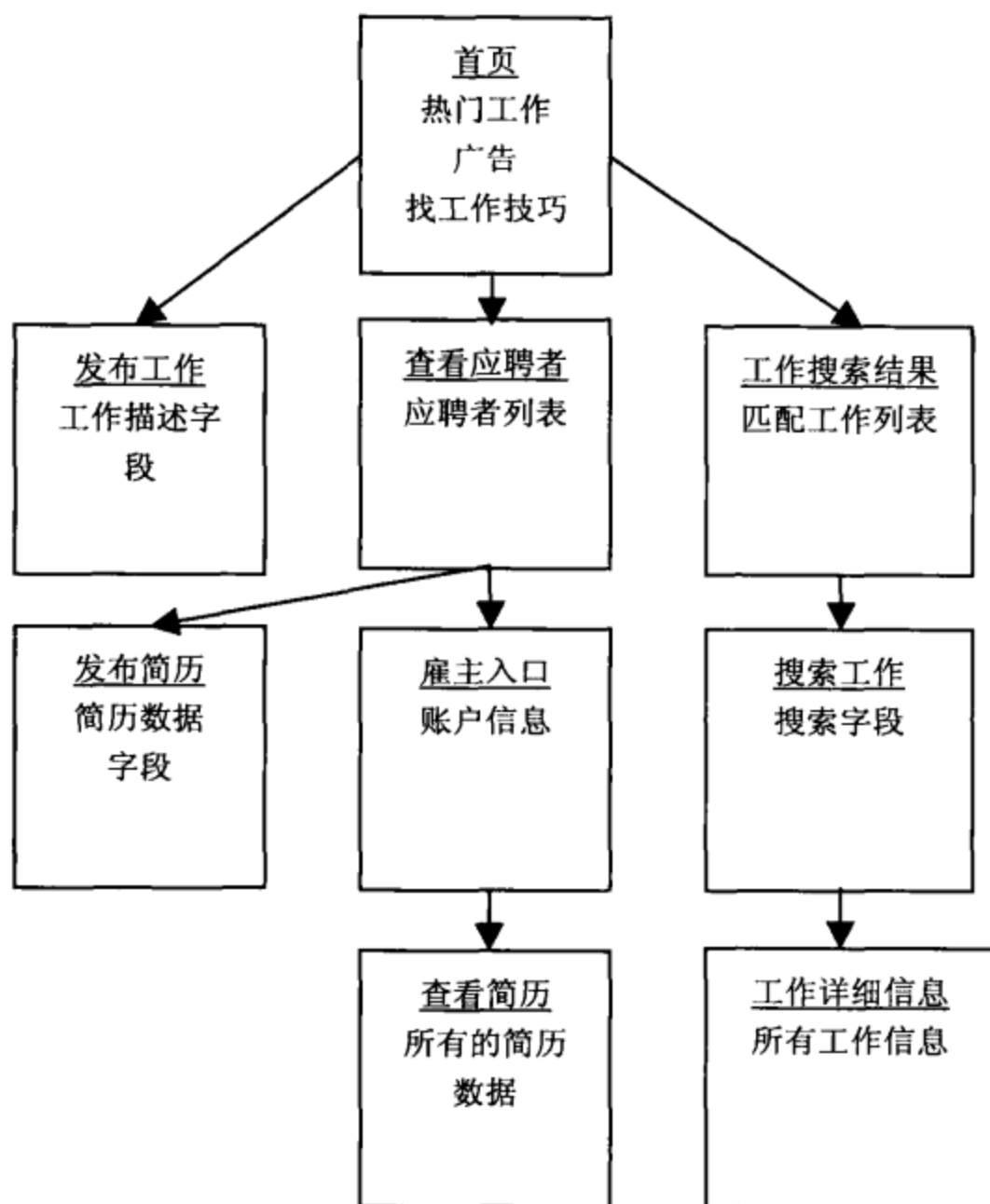


图 4.1 BigMoneyJob 的简单原型

在画图 4.1 的过程中，我们得到以下故事。

- 求职者可以发布他的简历。
- 雇主可以发布工作。
- 雇主可以查看提交的简历。
- 求职者可以搜索工作。
- 求职者可以看到符合搜索条件的工作。
- 求职者可以看到指定工作的详细信息。

这些故事都不需要知道界面该怎么设计。但是，走一遍流程可以帮助大家想出尽可能多的故事。我发现深度优先的方法最有效：对于第一个组件 A，写下主要的细节，接



着是与第一个组件相连的组件 B，一样写下其主要的细节。然后是与 B 相连的组件，而不是回到第一个组件 A，描述其他与 A 相连的组件。广度优先的办法非常不容易理解，因为很难记住自己刚才在哪条功能线上。

扔掉原型

在画好简单原型后几天内，一定要扔掉或擦掉它。原型并不是开发流程中的一个长期工件，因为长期留着可能会导致不必要的困惑。如果觉得在故事编写工作坊中还没有发现所有故事，可以把原型保留几天，再看看是否还能写出一些漏掉的故事，然后再考虑扔掉它。

在画原型的过程中，问一些有助于找到遗漏故事的问题，示例如下。

- 用户接下来最有可能做什么？
- 用户会在这里犯什么错误？
- 在这里，用户会有什么困惑？
- 用户需要什么额外的信息？

在问这些问题时，考虑一下当时用户的角色。许多答案都和用户当时的角色有关。

维护一个待办问题列表，留着以后再来解决。例如，在讨论 BigMoneyJobs 时，有人可能会问系统是否同时支持合同工和全职员工。如果在召开工作坊之前没有人想过这个问题，找一个可以看到的地方把它写下来，一会儿再来考虑它，可以在工作坊最后或者在工作坊后续的跟进工作之后。

在故事编写工作坊期间，我们应该把重点放在数量上而不是质量上。即使最后会用电子方式保存故事，但在工作坊里最好还是用卡片。只要把想法记下来就行。最初大家觉得不好的故事经过几小时后可能会变得很棒，或者这会激发我们想出其他故事。另外，不要为每个故事都陷入长时间的讨论中。如果一个故事是多余的或者能被更好的故事替换，就扔掉这个故事。同样，当客户为发布而确定故事优先级时，他可以给不好的故事安排低优先级。

有时，一些参与者在工作坊中很难开头，或者卡在某个点上过不去。这时不妨看看竞争对手的产品或类似的产品。

留意在故事编写工作坊中谁的贡献更多。有时，有些参与者在大部分或者整个工作坊期间都保持沉默。如果有这样的情况出现，在中间休息时去和这个参与者谈谈，确定

他并不是不适应这个过程。有些参与者不愿意在同事或上司面前说出自己的想法，因此不要在整个过程中评价某个故事好或坏。一旦参与者觉得大家只是在记录而不是评价故事，会更乐意参与。

最后，我再次重申故事编写工作坊中的讨论应在较高层面上。我们的目的是在短时间内写出更多故事。这不是设计界面或者解决问题的时候。

小结

- 能够引出及捕捉需求这一想法是错误的。它有两个有问题的假设：用户知道所有的需求；需求一旦被捕捉，就锁定，不再改变。
- 拖网捕鱼的比喻是非常有用的：它说明了需求有不同的大小，需求会随着时间的推移变化，需要一些技巧来发现需求。
- 即使敏捷流程支持需求的后期涌现，依然需要对预期的发布进行展望并开始写下容易发现的故事。
- 我们可以通过用户访谈、观察用户、问卷调查和举办故事编写工作坊来发现用户故事。
- 使用多种方法比过度使用一种方法更能获得好的效果。
- 通过开放式、与背景无关的提问更容易获得有用的答案，例如，“告诉我你怎么搜索工作？”就胜于“你要通过职位名称来搜索工作吗？”

开发人员职责

- 负责理解并使用多种技巧来捕捞用户故事。
- 负责知道怎么使用开放式和背景无关的提问。

客户职责

- 负责理解并使用多种技巧来捕捞用户故事。
- 负责尽早写更多的用户故事。
- 作为软件用户的主要代表，负责和他们多沟通。
- 了解怎么使用开放式和背景无关的提问。



- 如果需要关于编写故事的帮助，负责安排并举办一次或多次故事编写工作坊。
- 负责确保在捕捞故事过程中考虑所有用户角色。

问题

- 4.1 如果团队只通过问卷调查来搜集需求，会碰到哪些问题？
- 4.2 将下面的问题重新整理成开放式背景无关的问题：你认为用户必须输入密码吗？系统应该每 15 分钟自动保存用户的操作吗？用户能看到另一个用户的数据库录入信息吗？
- 4.3 为什么最好提供开放式的、与上下文无关的问题？



第 5 章 与用户代理合作

对于一个项目来说，客户团队里包括一个或多个真实用户是极其重要的。虽然其他人可以猜想用户如何使用软件，但事实上，关键往往还在于实际用户。遗憾的是，我们很难有机会与实际用户一起工作。例如，我们在开发一个广泛通用的产品，用户遍及全国各地，但我们没法也不适宜把一个(或多个)用户带到现场，与我们一起编写故事。或者我们正在开发一个给公司内部使用的软件，但被告知我们不能与用户一起讨论。我们期望与尽可能多的用户接触，这些用户代表了产品的不同角度，当我们无法接触到他们时，我们就需要求助于用户代理(user proxy)，他们自己可能不是用户，但他们在项目里代表着用户。

选择合适的用户代理对于项目的成功至关重要。我们要考虑潜在用户代理的背景和动机。有营销背景的用户代理识别故事的方法，不同于由领域专家担当的用户代理。重要的是要认识到这些差异。在本章中，我们会探讨有时代替实际用户、不同类型的用户代理。

用户的经理

在开发一个供内部使用的项目时，组织可能不愿意让你完全不受限制地接触一个或多个用户，却可能让你接触用户的经理。如果用户的经理不是软件的实际用户，这其实就是偷梁换柱。即使用户的经理的确是软件的用户，但他使用软件的模式肯定也与典型的用户不同。例如，在一个呼叫中心的应用程序项目中，开发团队获准接触呼叫中心的轮班主管。虽然轮班主管确实在使用这个软件，但他们在新版本中想要的功能主要集中于管理呼叫队列和在坐席之间转移呼叫上。这些功能对于轮班主管手下的人来说，重要性却很低，但这些人才是该软件的主要用户。若开发人员不能接触到更多典型的用户，他们就会过分关注轮班主管需要的功能，但这些功能很少使用。

有时候，用户的经理会从中干预，并且出于自负，想在项目中充当用户的角色。她可能承认自己不是典型的用户，但她固执己见，认为自己比她的用户更知道他们需要什么。当然，在这种情况下，务必小心，不要得罪用户的经理。但是为了项目的成功，在部分围绕她的同时，也要想办法接触终端用户。针对这种情况，在本章稍后的小节“与



用户代理合作时，做些什么？”将给出一些建议。

5 分钟不等于 1 分钟

这个内部项目的“用户”是副总裁，她从来没用过这个软件。在她和终端用户之间，还隔着经理层。在为下一轮迭代的故事安排优先级时，她希望开发人员专注于提高数据库查询的速度。开发团队也注意到了这个高优先级的故事，但他们有些困惑。他们知道应用程序的性能十分重要，所以已经在软件里建立了一个监测机制：每次执行数据库查询时，它的运行参数、执行查询花的时间以及用户的名字都会保存在数据库里。每天至少会监测一次这种信息，没有迹象表明有性能问题。可是，他们的“用户”仍旧告诉他们，有些查询需要花“多达 5 分钟”的时间。

在与副总裁会晤之后，开发小组研究了一下执行查询的历史记录。他们发现：有两个用户执行的查询操作实际只花一分钟就完成了。这肯定比所期望的时间要长，但考虑到他们查询的数据、数据库的大小以及执行这种类型的查询操作并不频繁，这种性能是符合预期的。但是用户已经将这件事情汇报给他们的经理了。之后经理又汇报给了副总裁；可是为了让副总裁注意到这个问题，经理却故意说查询花了 2 分钟时间。然后副总裁又将这个问题反馈给了开发人员，为了让开发人员足够重视这个问题，她将这个问题说成花了“多达 5 分钟”的时间。

用户的经理有时候是错误信息的来源。只要有可能，就要通过与实际用户交流来求证这些信息。

开发经理

让开发经理担任用户代理，是最坏的选择之一，除非你们开发的软件就是给开发经理用的。尽管开发经理可能没什么不好，但他们最想获取的是荣耀，他们的目标很可能是互相冲突的。例如，开发经理排列的故事优先级会不同于实际用户排列的，因为这样做可以让她提早给别人介绍令人兴奋的新技术。此外，开发经理的目标可能与企业目标不一致：或许她的年终奖跟项目的结束日期有关，这可能导致她想提前完成项目。

最后，大部分开发经理对正在开发的软件没有像用户那样的亲身经验，而且他们也不是领域专家。如果以后的用户代理是开发经理，而她恰恰又拥有领域知识，最好把她视为领域专家。在判断是否有合格的用户代理前，请阅读本章的“领域专家”小节。

销售人员

让销售人员充当用户代理是危险的，这会让大家对正在开发的产品没有一个全面的蓝图。对销售人员来说，最重要的故事是那些如果没有实现就会导致她“丢单”的故事。如果因为产品没有“撤销”(undo)功能而让她失去一位客户，完全可以打赌，她会马上把故事“撤销”的卡片调整到顶部。取决于丢单具体原因的重要程度，编写一两个新故事是值得的。但是，如果一家产品研发公司过分关注每一笔丢失的订单，他们可能会失去战略方向，产品的长期远景会停滞不前。

然而，销售人员是用户非常好的中转站，应该充分利用。具体做法是让他们通过电话或销售拜访把你介绍给客户。如果可以让他们参加行业展览，并且在你们公司的展位上工作，那就更好了。

同你的用户交谈

1995年，我带过这样一个团队，他们想挑战开发一个第一代全面的健康信息网站。由于那时没有竞争对手，所以在他们编写用户故事的时候，没有现成的网站可以参考。这个项目的用户代理是一位有市场营销背景的总监。由于他的市场背景，所以他知道与未来的用户交谈很重要，在交谈中，可以获知用户想在健康信息网站上获取哪些信息。然而，出于网站要尽早交付的压力，他急功近利，在开发这个站点的时候，单纯地跟着感觉走。

可以想像，这个项目无法满足用户的需求。大约一个月后，这个网站上线了，我走进那个营销总监的办公室，他指着显示器说道：“看那儿，就看那儿。”在他的屏幕上，是一个色情网站。过了一会儿，我问他为什么在看那个网站。我觉得他甚至没有注意到那个网站上的少儿不宜图片。他的眼睛只盯着点击计数器，说道：“你看，他们从昨天开始，已经有10万次点击率，我们却只有200次。”

如果希望软件是可用的，就要同未来的用户交流。

领域专家

领域专家，有时也称主题专家，是非常重要的资源，他们对软件应用领域的了解程度对软件的成败有直接影响。当然，有些领域相对于其他领域更难理解。我以前给律师



和律师助理写过很多软件，虽然有时候软件会很复杂，但我通常明白客户的真正需求。后来，我参与过为统计遗传学家开发软件。在那个领域里，充满了诸如表型(phenotype)、重组率(centimorgan)和单体型(haplotype)这些我以前闻所未闻的词汇，因此该领域变得更加难以把握。这使得每个开发人员需要更多地依赖于领域专家，让他们帮助我们了解我们正在开发的软件。

尽管领域专家是很好的资源，但他们是否对你有帮助，取决于他们是否目前或曾经使用过你们正在开发的这种软件。例如，开发一个工资系统时，毫无疑问你想要一位注册会计师(CPA, Certified Public Accountant)来作为领域专家。但是，由于未来的用户可能是薪资办理员，而不是注册会计师，很可能你会在薪资办理员那里了解到更好的故事。在建立领域模型、确定业务规则时，领域专家是理想的资源，但是最好从实际用户那里了解 workflow 以及使用方面的问题。

让领域专家来担任用户代理的另一个潜在问题是，最终开发出的软件可能仅仅针对那些与领域专家有类似水平的用户。领域专家会倾向于将项目指向适合他们自己的解决方案上进行，但这往往过于复杂，对目标用户群体而言明显是错误的。

市场营销团队

Larry Constantine 和 Lucy Lockwood 在 1999 年指出，了解市场的是营销团队，而不是用户。这会导致营销团队或者有营销背景的人更关注产品特性的数量，而轻视特性的质量。在很多情况下，营销团队可以为相关故事的优先级提供高层次的指导意见，但他们往往不具备很好的洞察力，无法提供故事的具体细节。

以前的用户

如果以前的用户在不久前还使用过你们的软件，那么由她来担任用户代理是非常好的。但和其他用户代理一样，应该谨慎考虑她的目标和动机是否与实际用户的完全一致。

客户

客户是那些做出购买决定的人，他们不一定是软件的用户。考虑客户的期望是很重要的，因为开支票买软件的人是他们，而不是用户(当然，除非你的用户和客户是同一

批人)。

企业的桌面办公软件是一个可以充分说明客户与用户区别的例子。企业的 IT 人员可以决定公司所有的员工使用哪一款字处理程序。这个例子中, IT 人员是客户, 公司所有的员工是用户(包括 IT 人员, 他们既是客户, 又是用户)。对于像这样的产品, 其功能一定要够用, 用户才不会大声抱怨; 但这些功能也一定要能够吸引客户, 使其决定购买。

再论与用户沟通

在一个公司里, 由市场营销团队担当用户代理, 他们将用一个新的产品来替换公司现有的纸质产品。这个公司的销售记录非常成功, 他们销售的纸质书包含了医院和保险公司之间商定的规则: 如果医院遵循了那些规则, 他们就可以向保险公司报销。例如: “若(以及其他条件)病人的白细胞数量高于一定的界限时, 医院才能做阑尾炎切除手术”。

市场营销团队没有兴趣通过与书籍的用户交谈来了解他们想让软件做什么。相反, 他们辩解说他们确切知道用户的需求, 开发团队可以在他们的指导下进行开发工作。营销团队选择了将软件做成纸书的电子版。他们并没有利用软件先天的灵活性, 而只是满足于将软件当成一本“自动化的书籍”。很明显, 用户对软件很失望。很不幸, 如果他们让实际用户而不是营销团队来担任用户代理, 那么公司很早就可能已经发现问题。

例如, 对于大部分桌面办公软件的用户而言, 安全特性通常不是很重要。然而, 对于那些做出购买决定的 IT 人员(客户)来说, 安全性却极为重要。

我曾经和一个项目团队一起工作, 他们设计过一个数据密集型应用程序。该程序的数据是从客户现有的其他系统载入的。开发人员需要定义一个文件格式, 用于交换数据。在这个案例中, 客户是公司的首席信息官(CIO); 这个功能的用户是他公司里的 IT 人员, 他们会编写数据抽取程序, 将现有系统的数据转换成新系统指定的格式。在问他对文件格式有什么样的偏好时, 客户(这位首席信息官)决定使用 XML 技术, 因为这个技术相对较新, 并且显然比非标准的逗号分隔文件格式(CSV)更有吸引力。交付软件时, 用户(IT 人员)完全不赞成——他们喜欢更简单、易于生成的 CSV 文件。如果开发团队从用户那里直接获取到故事, 那么他们早就可以知道这一点, 也不会浪费时间在 XML 格式上。



培训师和技术支持

由培训师和技术支持人员来充当用户代理看似是合乎逻辑的选择。他们整天与实际用户交谈，所以他们一定知道用户的需求。不幸的是，如果用培训师充当用户代理，你的系统最终只能成为一个容易培训的系统。类似的，如果你让技术支持来充当用户代理，那么你的系统最终只是使得支持工作变得较为容易。例如，某个做技术支持的人可能会把高级功能的优先级排得较低，因为她预计这会增加支持工作的工作量。把易于培训和良好的可支持性作为目标时，他们做优先级排列时很可能与真实用户不同。

业务分析师或系统分析师

许多业务和系统分析师是很好的用户代理，因为他们既懂技术，又熟悉软件相关的领域知识。能平衡好这些背景且努力跟实际用户沟通的分析师，通常会是非常出色的用户代理。

有些分析师暴露出来的问题是，他们遇到问题喜欢空想，而不是去做调查。我曾经与太多的分析师一起工作，他们相信自己可以坐在办公室里凭直觉就知道用户的实际需求，而不需要与用户交谈。必须注意，项目的分析师应该同用户讨论，她不能只根据自己的观点来做决定。

让分析师担任用户代理时，第二个问题是偶尔会有分析师喜欢在项目前期花太多时间。在两小时的角色建模和故事编写工作坊足以填满未来四个月的发布计划时，有些分析师却喜欢在这些活动上花3周时间。

与用户代理合作时，做些什么？

虽然不太理想，但不和实际用户一起而和用户代理一起，还是可能开发出优秀软件的。在这种情况下，有很多方法可以用来促进成功。

能接触到用户但访问受限时

访问实际用户受阻且团队被告知要和用户代理一起工作，由用户代理来做出项目相关的所有决定时，团队就要和他们合作，但同时也要与用户建立便捷的联系。最好的方

法之一是请求准许启动一个用户顾问团队(user task force)。用户顾问团队可以由数量不限的实际用户组成,从几个到几十个人都可以。顾问团队能够提出意见和建议,而用户代理依然是项目最终的决策者。大多数情况下,用户代理会同意那么做,特别是因为这让她有一个防护网从而避免做出错误决定的时候。

一旦建立起用户顾问团队,并且配备实际用户,它就可以用来指导每天越来越多的关于项目的决策。可以开一系列的会议来讨论软件的一小部分,然后让顾问团队来识别、编写并且排列用户故事。

曾经有一个项目小组,他们正在开发一个供内部用户使用的系统,并且已经在用户代理的大力指引下取得了巨大的成功,他们先给用户顾问团队展示原型,然后根据在用户顾问团队会议上获得的反馈来做改进。这个特殊的项目以一个月作为一轮迭代。迭代中每周的最初几天,用来开发原型系统,开几次用户顾问团队会议。用这种方式,用户代理(这个案例中是用户经理)很好地控制着项目的战略方向,而实现细节却转移到了用户顾问团队。

实在不能接触到用户时

实在不能接触到用户时,必须求助于用户代理,一种有价值的方法是使用多个用户代理。这有助于减少一种可能性,即开发的系统仅仅准确地满足了一个人的需求。使用多个用户代理时,要确保利用不同类型的用户代理。例如,将一个领域专家和有市场背景的人组合,而不是使用两个领域专家。为此,要么指定两个用户代理,或者只指定一位用户代理,但鼓励她依靠其他非正式的用户代理。

如果正在开发和其他商业产品竞争的软件,可以使用竞争者的产品作为一些故事的来源。在软件测评里面提到了同类产品中的哪些功能?在线新闻组里讨论过哪些功能?这些功能成为讨论的焦点是否由于其使用起来过于复杂?

记得几年前,我和一个用例(use case)拥护者争论过哪种类型的文档最能表达系统需求。他支持深思熟虑的用例模型。我支持用户的指引。我从未见过一个项目以完全准确和通用的用例模型来完成,即使有些人尝试过。但我却见过许多通过准确、通用的用户指引而完成的项目。假若正在开发新的软件与现有产品竞争,便可以从竞争产品中学到很多东西。

在与用户代理而不是实际用户合作时,另一个可用的方法是尽早发布产品。即使发布被称之为雏形版本或早期测试版,及早将软件交付到用户手里,有助于辨别出用户代



理与实际用户之间想法的不一致性。更妙的是，一旦软件交付到一个或多个早期用户手里，你就打开一条与用户沟通的途径，并且可以利用它与用户讨论后续的功能。

可以自己来吗？

当你无法找到或者访问到实际用户时，要避免思维陷入误区：你知道用户的想法，所以不需要或者可以忽略你的用户代理。虽然每种类型的用户代理较之实际用户都有一些缺点，但大部分开发人员冒充实际用户时，有更多缺点。总的来说，开发人员没有市场背景，他们不懂软件功能的相对价值，他们不像销售人员那样频繁联系客户，他们不是领域专家，等等。

设立客户团队

首先，请记住，在任何时候，实际用户总是优于用户代理。只要可能，就要邀请实际用户加入客户团队。然而，当实际用户不能加入客户团队时，就需要有一个或多个用户代理。应该将客户团队建立成一个优势互补的团队，一位成员的优势能平衡另一位成员的弱势。建立客户团队有三步。

第一，邀请真实用户加入。如果有不同类型的用户使用软件，试着邀请每种类型的用户。例如，在一个健康保健应用程序中，我们的用户是护士。在我们的客户团队中，有常规的护士、肿瘤专家、糖尿病专家等。

第二，在客户团队中确定一位“项目负责人”(project champion)或“一把手”(first among equals)。在商业软件公司里，这个人通常是产品经理，但也可以是其他人。这位项目负责人负责协调客户团队的协作。客户团队的所有成员，应尽力做到传递一致的信息。尽管可能有多个客户，但对于开发项目来说，必须只能有一个客户声音。

第三，确定项目成功必须的关键因素。这点随着项目的不同而不同。例如，如果项目是现有产品的下一代版本，那么成功的关键之一是如何让现有用户轻松地转移到新系统上。将具有相关知识、技能和经验的用户代理补充到客户团队中，是项目成功的关键因素。在将现有用户转移到新系统的例子里，这可能意味着要在客户团队中加入一位培训师。

小结

- 在本章中，我们学习了不同类型的用户代理，讨论了编写用户故事时，为什么用户代理不如实际用户理想。
- 除非用户的经理也是用户，否则她就不是合适的用户代理。
- 开发经理会试图担任用户代理，因为他们已经参与到项目每天的细节中。然而，开发经理大多不是正在开发的软件的用户，所以他们不是理想的用户代理。
- 在产品公司里，客户经常来自市场团队。来自市场团队的人经常是不错的用户代理，但他们通常关注于软件的功能数目，而不是其质量，这点必须要克服。
- 与很多不同的客户(而这些客户同时也是用户)联系的销售人员可以是很好的开发项目客户。销售人员必须避免把重点放在那些可以重新赢得已失去订单的故事上。在所有情况下，销售人员是与用户沟通的有效渠道。
- 领域专家可以成为优秀的用户代理，但必须避免一点：在为产品编写故事时，将产品开发成只适合那些与他们有相同水平的人使用。
- 客户，那些做出购买决定的人，如果他们能与用户密切地交流，那么他们能成为非常好的用户代理。显然，如果客户自己也是用户，那就是完美的组合。
- 为了成为好的用户代理，培训师和技术支持人员必须避免仅仅关注产品中那些他们每天关心的方面。
- 本章也简短地给出了一些与用户代理一起工作的方法，包括用户顾问团队的使用，使用多个用户代理，分析竞争产品，尽早发布软件来获取用户反馈。

开发人员职责

- 负责帮助组织机构为项目物色合适的客户。
- 负责了解不同类型的用户代理怎么考虑你们正在开发的系统，他们的背景如何影响交互。

客户团队职责

- 如果你不是软件的用户，则要负责了解自己属于哪类用户代理。
- 负责理解自己会将哪些偏见带入到项目中，如何克服这个问题，无论是依靠别



人还是其他方法。

问题

- 5.1 让用户的经理充当用户代理，会导致什么问题？
- 5.2 让领域专家充当用户代理，会导致什么问题？



第 6 章 用户故事验收测试

写验收测试的好处有很多，其中之一就是很多客户和开发人员讨论的很多细节可以通过验收测试记录下来。比起写冗长的需求列表，像“系统应该……”可以用测试来充实很多用户故事的细节。

一种比较好的观点提出，测试是一个两步流程：第一，将测试要点记录在故事卡的背面，任何时候发现新的测试，都可以记录到故事卡的背面；第二，将测试要点变成全面的测试，这些测试可以用来演示故事已正确、完整地实现。

以下是一个记录在故事卡背面的测试要点的例子，“公司可以用信用卡支付发布工作的费用”这个故事卡的背面可能有以下这些测试要点：

- 用 Visa 信用卡、万事达信用卡(MasterCard)和美国运通卡，(America Express) 测试。(通过)
- 用大来卡(Diner's Club)测试。(失败)
- 用正确的、错误的和空的卡号测试。
- 用过期的信用卡测试。
- 测试不同的交易金额(包括超过信用卡额度限制)

这些测试要点记录了客户提出的一些假设。假定 BigMoneyJobs 例子中的客户写了一个故事“求职者可以查看指定工作的详细信息”。客户和开发人员讨论这个故事，确定一些需要显示的一些工作信息——职位名称，描述，工作地点，薪水范围，如何申请，等等。然而，客户了解并不是所有公司都会提供所有这些信息，所以她希望网站能够自动处理未填的数据。例如，如果没有提供薪水信息，客户甚至不希望“薪水范围”标签出现在屏幕上。这个应该在一个测试里反映，因为程序员可能会假设系统发布工作模块要求所有工作都提供薪水信息。

验收测试也提供了确认故事是否被完整实现的基本标准。有了这样的标准，我们就知道什么时候某件事算是做完了，这是避免花太多或太少的时间和精力的最好方法。举个生活中的例子，我妻子烤蛋糕时，她的验收测试就是在蛋糕里插一个牙签。如果牙签拿出来时是干净的，那么蛋糕就算是做好了。而我则是将手指插入蛋糕，然后尝尝，以



此来验收测试她做的蛋糕。

在写代码之前写测试

在开始编写故事代码之前，验收测试可以为程序员提供了大量有用的信息。例如，想想“测试不同的交易金额(包括超过信用卡额度限制)”。如果在程序员开始写代码前写了这个测试，它会提醒程序员处理因信用额度不够导致交易失败的情况。如果没有看到这个测试，有些程序员就会忘记支持这种情况。

显然，为了让程序员尽早了解这些信息，应当在为这个故事编写代码前就开始制定验收测试。一般在下面这些时候写测试。

- 开发人员和客户讨论故事且需要记录明确的细节时。
- 在迭代开始时，在写代码前作为一项专门的任务。
- 在开发中或之后的任何时候发现新的测试时。

理想状况下，在客户和开发人员讨论故事的时候，他们把细节都写成测试。但是，在迭代的开始，客户就要过一遍所有故事，写一些他能想到的测试。比较好的做法是，考虑每个故事，然后问类似下面的一些问题。

- 关于这个故事，程序员还需要知道什么？
- 对怎么实现这个故事，我的想法是什么？
- 有没有一些特殊情况会使这个故事有不一样的行为？
- 这个故事在什么情况会出错？

故事卡 6.1 展示了一个真实项目的例子，一个扫描软件的故事。这个故事的作者清楚地知道他的期望(例如：在一个新的文档中打开新扫描的页面，即使软件已经打开了一个文档)。在这个例子中，这个期望被作为故事的一部分写在了卡片的正面。我们也可以轻松地将这个期望作为卡片背面的第一个测试。重要的是，在程序员开始实现这个故事前，通过故事卡片可以了解这个期望。否则，程序员很有可能写出不一样的软件行为，如将新扫描的页面插入当前文档。

故事卡 6.1 告诉程序员你的期望

用户可以扫描页面并将其插入新的文档。如果已经打开了一个文档，那么程序应提示用户并关闭当前文档。

客户定义测试

既然软件是用来实现用户的愿景，验收测试当然就应当由客户来定义。客户可以和程序员与测试人员合作创建测试，但是客户至少应该给我们详细指出一些测试，用以验证故事的实现是正确的。另外，一个开发团队(特别是有资深测试人员的)经常还会定义其他的测试。

测试是过程的一部分

最近我和一家公司一起工作，这里的测试人员对软件的理解都来自于程序员。程序员为新功能编写了代码，他们向测试人员解释这个功能，然后测试人员验证程序是否表现出所描述的行为。一般情况下，程序都能通过测试，但轮到用户开始使用时，却总出现这样那样的问题。问题当然是出自测试人员总是按照程序员的描述去测试。如果没有客户或用户的参与，我们不会真正从他们的角度来测试软件。

测试是开发过程中的一部分，而不是在编码完成后要做的事，这点对使用用户故事非常重要。一般情况下，产品经理和测试人员共同负责列出详细的测试。产品经理带来驱动项目的公司目标的知识；测试人员则带来怀疑的心态。在一轮迭代开始阶段，他们应该一起列出尽可能多的测试。但这还不够，也不是他们每周碰一次就足够了。随着故事细节逐渐展现，往往又能找出更多的测试。

多少测试才算多？

只要这些测试还在继续为故事增加价值和使它更加清晰，客户就应该继续写测试。如果针对“不能用过期万事达卡付费”这种情况已经写了一个测试，那就没有必要再为Visa卡写同样的测试。

同时记住，一个优秀的开发团队会为很多详细的用例写单元测试。例如，开发团队应该制定能识别2月30日和6月31日是不合法日期的单元测试。客户不负责定义所有可能的测试。客户应该更专注于那些能向开发团队说明故事意图的测试。



集成测试框架

客户负责引领系统的开发，而验收测试则向客户演示软件是可接受的。这意味着客户应该执行验收测试。至少，在每轮迭代结束时应该执行验收测试。因为每轮迭代产生的可工作的代码在接下来的迭代开发中可能遭到破坏，所以每轮迭代都要执行以往迭代的所有验收测试是非常重要的。这样，每轮迭代都要花更多的时间来执行验收测试。如果可能，开发团队应该自动化部分或全部验收测试。

一个非常好的自动化验收测试工具是 Ward Cunningham 最早开发的集成测试框架^①，简称 FIT。测试可以使用 FIT 写成熟悉的电子表格或其他表格格式。Bob 和 Micah Martin 主持 FitNesse^②的开发，它是一个简化测试编写的 FIT 扩展。

FitNesse 迅速成为敏捷项目中编写验收测试的非常流行的方法。因为测试在网页中以电子表格的样式呈现，这极大降低了客户定义编写测试的工作量。表 6.1 是该工具可以处理的一种表格类型。每一行表示一组数据。这样，第一行数据定义一个 Visa 卡在 2005 年 5 月过期，卡号为 4123456789011。最后一列展示卡是否应该在程序中通过有效性检查。^③因此，在程序中这张卡应该被认为是有效的。

表 6.1 可在 FIT 和 FitNesse 使用的有效信用卡测试表格

CardType	Expiration	Number	valid()
Visa	05/05	4123456789011	true
Visa	05/23	4123456789012349	false
MasterCard	12/04	5123456789012343	true
MasterCard	12/98	5123456789012345	false
MasterCard	12/05	42	false
American Express	4/05	341234567890127	true

要执行表 6.1 中的测试，团队中的程序员需要编写一些代码以响应简单的 FIT 命令。

① 集成测试框架，FIT，可在 fit.c2.com 找到。

② FitNesse 可在 fitnesse.org 找到。

③ 有关信用卡有效性信息，请参考 www.beachnet.com/~hstiles/cardtype.html。

代码将调用程序中需要测试的用来检查信用卡有效性的代码。客户只需要设计一些这样简单的表，然后向其中填充数据和期望的结果，即可完成验收测试的编写。

运行表 6.1 中的测试时，测试列(例子中最后一列)将标成绿色(测试通过)或红色(测试失败)。FitNesse 和 FIT 简化了客户或开发人员运行验收测试。

测试类型

测试类型有很多，客户和开发团队共同确保系统测试涵盖了项目需要的所有不同类型的测试。对于大多数系统来说，故事测试主要是功能性测试，用来确定应用程序是如期运行。不过，也应当考虑其他类型的测试。示例如下。

- 用户交互测试，确保所有用户交互组件如期工作。
- 可用性测试，确保程序好用。
- 性能测试，测量应用程序在各种负荷下的工作状况。
- 压力测试，使应用程序在用户和事务的极限值情况或其他任何让应用程序处在压力下的情况运行。

测试的是缺陷，而不是覆盖率

在一个敏捷的由故事驱动的项目中，测试并不像很多团队那样是一个对抗性的活动(译者注：与开发活动相对抗)。发现缺陷时，不该有“被我逮到了吧”这样的心态。在敏捷开发中，若有缺陷直到系统投产的时候才被发现，团队成员是不应该互相推卸责任的。高度协作的团队以及“我们共同负责”的心态能防范这种事情的发生。

在敏捷项目中，测试的目的发现并消除缺陷，所以没有必要追求 100%的代码覆盖率或测试所有的边界条件。我们运用我们的直觉、知识和过去的经验来指导测试。

选择最合适的人来执行测试。客户应定义验收测试，但是需要开发人员和专职测试人员的帮助和信息。例如，表 6.1 中的测试。测试中唯一过期的卡是万事达卡。如果我们争取完全覆盖，也势必会测试其他类型的信用卡。但在与客户和开发人员交流之后，他们知道(在这个例子中)所有的卡都是统一处理的，所以测试一个类型就够了。随着时间的推移，通过频繁的沟通和观察哪些类型的测试经常出现问题，项目中所有人都可以知道测试重点在哪些地方。



小结

- 验收测试可以用来记录客户和开发人员讨论的很多细节。
- 验收测试记录了有关故事的一些假设，这些假设可能还没有和开发人员讨论过。
- 验收测试提供了检查故事是否被完整实现的基本标准。
- 验收测试应由客户来写而不是开发人员。
- 验收测试应在程序员写代码之前就写好。
- 如果新的测试对阐明故事的细节或意图没有任何帮助，就不用再写。
- FIT 和 FitNesse 是写验收测试的优秀工具，它们用的是我们熟悉的表格或电子表格格式。

开发人员职责

- 若团队觉得有需要，则负责实现自动化验收测试。
- 开始开发一个新的故事时，负责考虑更多的验收测试。
- 负责为代码作单元测试，使验收测试就不必顾及故事的每个细节。

客户职责

- 负责编写验收测试。
- 负责执行验收测试。

问题

- 6.1 哪些人负责定义测试？哪些人负责提供帮助？
- 6.2 为什么要在写代码前定义测试？



第7章 优秀用户故事准则

至此，我们有了一个很好的基础，我们了解了什么是故事，如何拖网式捕捞以及编写故事，如何识别关键的用户角色以及验收测试在其中起到的作用。下面我们将了解一些额外的编写优秀故事的准则。

从目标故事开始

在一个大型项目中，尤其是有许多用户角色的项目，确定用户故事有时让人无从下手。我发现最好的办法是考虑每一个用户角色，了解用户使用我们软件的目的。例如，思考一下 BigMoneyJobs 例子中的求职者角色。她确实有一个最高优先级的目标：找到一份工作。但我们可以认为这个目标包括以下目标：

- 搜索她感兴趣的工作(基于她的技能、期望薪资、工作地点等)
- 自动搜索，以便不用每次都手动搜索
- 让她的简历可见，以便招聘公司能搜索到她
- 很容易申请她喜欢的任何工作

这些目标(实际上是高层次的故事)可以用来衍生出新的故事。

切蛋糕

当面临一个大的故事时，通常有许多方法可以将它分解成较小的故事。许多开发人员首先想到的是将故事按照技术路线分割。例如，假设团队觉得故事“求职者可以发布简历”在当前这轮迭代中太大了，就必须分割。开发人员可能想沿着技术边界分割，示例如下。

- 求职者可以填写简历表。
- 简历表上的信息被写入数据库。

在这个案例中，一个故事会在当前的迭代中完成，而另一个故事则(很可能)推迟到

下一轮迭代里。这种做法的缺陷是，没有一个故事是单独对用户很有用的。第一个故事说的是求职者可以填写简历表，但数据没有被保存。第二个故事说的是从简历表上搜集的数据会写入数据库。若没有第一个故事提供表格给用户，第二个故事就没什么价值。

一个更好的办法是换一种方式编写故事，每个故事都提供某种程度的完整(end-to-end)的功能。Bill Wake(2003a)将其称之为“切蛋糕”(slicing the cake)。根据这个原则，我们可以把故事“求职者可以发布简历”像下面这样分。

- 求职者可以提交简历，简历上只包括诸如名字、地址和教育背景这样的基本信息。
- 求职者可以提交简历，简历上包括雇主想看的所有信息。

在编写用户故事时，更倾向于编写像一块完整蛋糕那样功能完整的故事。具体有两个原因。首先，在开发中，及早涉及软件应用程序架构的每一层能够有效地降低最后时刻才发现层次架构方面问题的风险。其次，尽管不十分完美，即使只提供部分功能，但只要发布的功能可以跑，就可以放心地把应用程序发布给用户使用。

编写封闭的故事

Soren Lauesen(2002)在他的需求技术汇编中引入了任务闭包性的想法。他的想法同样适用于用户故事。一个封闭的故事是指那种随着一个有意义的目标的实现而结束的故事，能让用户使用后觉得她完成了某个任务。

例如，假设 BigMoneyJobs 网站项目包含故事“招聘者可以管理她发的招聘广告”。这不是一个闭合的故事：管理她发的招聘广告是没法彻底完成的事情。相反，它是一个持续进行的活动。这个故事可以更好地创建成一个闭合故事的集合，示例如下。

- 招聘者可以审核针对他发布的招聘广告发的简历。
- 招聘者可以更改招聘广告的过期日期。
- 招聘者可以删除不适合的申请。

.....

这种封闭的每一个故事都是原来那个非封闭故事的一部分。使用完这些封闭故事之后，用户可能会有一种成就感。

编写封闭故事其实是在相互冲突的各种需求之间权衡的结果。因为，故事也要小到能做评估，小到可以方便地安排到一轮迭代中。但故事也要足够大(译者注：这里的大指

的是粗粒度的、高层次的、抽象的), 从而避免过早捕获当下还不需要的细节。

卡片约束

Newkirk 和 Martin(2001)推荐过一种实践, 我觉得它是很有用的。他们引入的实践, 是对于任何必须要遵守而不需要直接实现的故事, 在其故事卡上标注“约束”(constraint)。故事卡 7.1 是一个例子。

故事卡 7.1 一个约束故事卡的例子

系统必须支持最大 50 个并发用户的峰值。

约束

其他约束例子如下。

- 设计的软件要便于今后实现国际化。
- 新系统必须使用我们现有的订单数据库。
- 该软件必须能在所有版本的 Windows 系统上运行。
- 该系统的无故障运行时间要求达到 99.999%。
- 该软件要很好用。

尽管约束卡不需要做估算, 也不会像普通卡片那样被安排到迭代中, 但它们仍然很有用处。至少, 可以把约束卡贴在墙上作为提醒。更妙的是, 可以编写验收测试来确保系统没有违反约束。例如, 为故事卡 7.1 编写测试不是一件难事。理想情况下, 团队可以在最初几轮迭代中的一轮中编写测试, 那时系统违反约束的可能性还很小。然后团队可以在后续的迭代中持续运行这些测试。只要可能(通常如此), 就要编写自动化测试来确保系统满足约束。

要想进一步了解如何将约束作为制定非功能性需求, 请参考第 16 章。

根据实现时间来确定故事规模

你想专注于最需要你关注的领域。通常, 这意味着要把注意力放在那些即将发生的



事情上，而不是放在更远的将来才发生的事情上。对故事而言，要基于故事实现的时间跨度，以不同的层次来编写故事。例如，对于下面几轮迭代的故事，它们的大小应该能够安排进那几轮迭代中，而对于更遥远的故事，它们可能会更大，但精确度则更低。例如，假设我们已经决定，BigMoneyJobs 网站最高层次的故事有以下 4 个。

- 求职者可以发布简历。
- 求职者可以搜索职位空缺。
- 招聘者可以发布职位空缺。
- 招聘者可以搜索简历。

客户决定第一轮迭代专注于允许用户发布简历。只有在简历发布功能的主体完成后，才把注意力集中于职位搜索、职位发布和搜索简历。这就是说，项目团队和客户将开始就“求职者可以发布简历”进行讨论。通过沟通，那个故事将扩展出细节，另外 3 个高层次的故事先放在一边。故事列表如下所示。

- 求职者可以添加一份新简历到网站上。
- 求职者可以修改已经在网站上的简历。
- 求职者可以从网站上删除她自己的简历。
- 求职者可以把简历标识为激活状态。
- 求职者可以对特定的雇主隐藏自己的简历。
- 求职者可以查看她的简历被浏览过多少次。
- ……关于发布简历的故事……
- 求职者可以搜索职位空缺。
- 招聘者可以发布职位空缺。
- 招聘者可以搜索简历。

在编写故事时，要利用故事灵活性的优势，让它们能够用于不同的层次。

不要过早涉及用户界面

一直困扰着软件需求方法的问题之一是将需求和解决方案混在一起。也就是说，在陈述需求的时候，也显式说明或暗示了解决方案。最常见的情况是用户界面。你想尽量让故事不包含用户界面。例如，考虑一下故事卡 7.2，它来自一个真实的系统。在项目早期开发这个故事是比较困难的，因为它包含太多用户界面的细节。这个故事描述了读者打印对话框、打印机列表和至少 4 种搜索方法。

故事卡 7.2 有太多用户界面细节的卡片

打印对话框允许用户修改打印机列表。用户可以对打印机列表增加或删除打印机。用户可以通过自动搜索，或者手动指定打印机 DNS 名或 IP 地址来添加打印机。“高级搜索”选项也允许用户使用 IP 地址和子网范围来限定搜索。

最终，用户界面细节不可避免地陷入故事。这种情况出现在软件变得越来越完整，并且故事从全新的功能转向修改或扩展现有功能的时候。

例如，考虑一下故事“用户可以在搜索界面上从日期小部件上选择日期”。不论它是在项目早期还是晚期完成，这个故事可能代表 3 天的工作。然而，在项目早期，还没有考虑用户界面的时候，这样的故事显然不是我们想要的。

有些需求并不是故事

尽管用户故事是一个非常灵活的格式，可以很好地描述许多系统的很多功能，但它们也并不是“万灵药”。如果需要，可以放心使用用户故事以外的其他形式来表达某些需求。例如，用户界面参考通常是以有很多截图的文档来描述的。类似地，除了用户故事外，也可以编写文档，对重要系统间的接口达成一致，尤其是那种由外包软件公司开发的系统。

如果发现系统某些方面更适合用其他表达方式，尽可放心使用。

在故事里包括用户角色

如果项目团队已经识别出用户角色，那么在编写故事时就要使用它们。所以不要写成“用户可以发布她的简历”，而要写成“求职者可以发布她的简历”。两者的差异是微小的，但这样编写故事，能让用户在开发人员脑子里保持最重要的位置。开发人员不会去想乏味的、不形象的、可切换的用户，他们会联想实际的、真切的用户，开发出满足用户需求的软件。

Connextra，最早采用极限编程的公司之一，通过使用一个简略的模板，将角色融入他们的故事。每个故事使用下面的格式编写：

我作为(角色)，想要(功能)，以此(商业价值)



你可能想尝试这个模板或使用自己的模板。像这样的模板有助于区分重要的故事和无关紧要的故事。

只为一个用户编写

当故事只为单一用户编写时，故事的可读性通常是最强的。对许多故事而言，针对的用户是一个还是许多个，没什么差异。然而，对有些故事而言，这种差异却是显著的。例如，考虑一下这个故事“求职者可以从网站上删除简历”。这可以解释为：求职者可以删除她自己的简历，也可能是删除其他人的简历。

一般来说，从单个用户的角度考虑故事时，这类问题会变得更清晰。例如，前面的故事可以写成“求职者可以删除简历”。如果这么写，求职者可能删除他人简历的问题会变得显而易见，然后就可以进一步改进，将故事写成“求职者可以删除她自己的简历”。

以主动语态编写

用户故事用主动语态编写时，更易于阅读和理解。例如，不要写成“简历可以被求职者发布”，而要写成“求职者可以发布简历”。

由客户编写

在理想情况下，故事由客户编写。在许多项目中，开发人员会帮忙编写故事，他们要么在最初的故事编写工作坊中进行实际编写，要么给客户建议新的故事。但是，编写故事的职责在于客户，不能转嫁给开发人员。

此外，因为客户有责任排列故事的优先级，将它们放入每一轮迭代中，所以客户理解每个故事是至关重要的。而要做到这一点，亲自编写无疑是最好的方法。

向故事卡编号说“不”

初次使用故事卡时，很多人会情不自禁地给它们编号。这么做的理由通常是，这有助于追踪单独的卡片，或者给故事加入了某种程度的可追踪性。例如，当我们发现 13

号卡片上的故事太大了，我们就把13号卡片撕碎，并且用卡片13.1、13.2和13.3取代它。然而，给故事卡编号会增加无谓的流程，并导致我们去抽象地讨论需要形象化的功能。我宁愿说“增加用户组的故事”，而不是“13号故事”。我特别不想说“13.1号故事”。

如果觉得必须要给故事卡编号，那么试着给卡片加上一个简短的标题，并在其余的故事描述文本中使用这个标题作为缩写。

不要忘记意图

不要忘记，故事卡的主要目的是用来提醒开发人员和客户团队对功能进行讨论的。既然仅仅是一个提醒，就要保持它的简洁性。加入需要的细节，以便联想到继续对话的切入点，但不要在故事卡上加入太多细节并以此取代对话。

小结

- 为了确定故事，从每个用户角色使用系统的目标开始考虑。
- 分割故事时，试着将它分割成贯穿应用程序所有层面的故事。
- 试着让故事的大小能够在使用后让用户感到可以去喝杯咖啡休息一下。^①
- 如果有项目领域和环境的需要，可以用其他需求搜集或文档技术来补充故事。
- 创建约束卡，将它们贴在公共的墙上，或者编写测试来确保系统没有违反约束。
- 为团队即将实现的功能编写小的故事，针对未来实现的功能编写宽泛的、高层次的故事。
- 不要让故事过早涉及用户界面。
- 实际编写故事时，要包括用户角色。
- 用主动语态编写故事。例如，要说“求职者可以发布简历”，而不要说“简历可以被求职者发布”。
- 为单个用户编写故事。不要写“求职者可以删除简历”，而要写“求职者可以删除她自己的简历”。
- 让客户，而不是开发人员，编写故事。

^① 译注：作者的意思是在故事被采用后，能让用户有些成就感。



- 用户故事要简短，别忘了，它们的目的是提醒开发人员和客户进行对话。
- 不要给故事卡编号。

问题

- 7.1 假设用户故事“求职者可以搜索职位空缺”太大了，不适合放入一轮迭代。你怎么分割它？
- 7.2 以下用户故事中，哪些故事的大小适中，并且可以认为是封闭的故事？
 - a. 用户可以保存她的偏好。
 - b. 用户可以修改用于购买的默认信用卡。
 - c. 用户可以登录系统。
- 7.3 应该怎样改进故事“用户可以发布他们的简历”？
- 7.4 应该怎样测试“软件要好用”这个约束？





第 II 部分 估算和计划

在对故事有了大体的理解后，我们将注意力转到如何使用用户故事进行估算和计划。在几乎所有的项目中，我们需要或者被要求估算项目所需时间。各种需要——市场推广的准备，用户的培训，硬件购买，等等——都依赖于项目计划。

在第 II 部分中，首先介绍如何估算故事以及如何为最高优先级的故事创建一个高层次的发布计划。然后介绍如何在每轮迭代开始时为当前迭代的工作做必要的计划，以此来细化发布计划。最后介绍度量和监控项目进展的方法，以便根据每轮迭代获得的知识持续调整计划。

第 8 章 估算用户故事

没有一个项目会在启动很久之后才会有人开始问“什么时候能完成？”估算故事的最好方法具有如下特点。

- 无论什么时候获得有关故事的新信息，都允许我们改变之前的想法。
- 适用于史诗故事和小故事。
- 不需要花很多时间。
- 提供进度和剩余工作的有用信息。
- 不太精确的估算也不会有太大问题。
- 可以用来制定发布计划。

故事点

有一种可以满足所有这些目标的估算方法，即用故事点(story point)估算。故事点有个很好的特性是团队可以定义自己认为合适的故事点。一个团队可能决定定义一个故事点为一个理想日的工作(也就是说，一天中没有任何干扰，没有会议，没有电子邮件，没有电话，等等)。另一个团队可能定义一个故事点为一个理想周的工作。还有一个团队可能把一个故事点作为故事复杂度的测量。因为故事点有很多意义，所以 Joshua Kerievsky 认为故事点代表时间的模糊单位，或叫 NUT(Nebulous Units of Time)^①。

我个人比较偏好将一个故事点的工作量看作是一个理想日的工作。我们极少会有这样的理想日，但是用理想时间考虑故事有两个好处。第一，相较于用连续时间估算，它更简单。用持续时间估算迫使我们考虑对时间的各种可能的影响，例如，星期二全公司会议，星期三约了牙医，每天花几小时回邮件，等等。第二，相较于用完全模糊单位，用理想日估算故事点可使我们的估算拥有更好的依据。估算的主要目的之一是知道整个项目的工作量，所以最后我们总是要将估算换算成时间。显然，相较于用完全模糊单位，

^① Joshua Kerievsky 在 2003 年 8 月 5 日的极限编程 Yahoo 讨论组中提到。

用理想时间更简单。

以团队估算

故事估算应该由整个团队集体完成。稍后，在第 10 章中，我们将看到一个故事包含多个任务，任务估算属于执行这个任务的个人。然而故事估算属于团队集体有两个原因。第一，还不确定团队中谁负责完成这个故事，所以应该把故事分配给整个团队而不是某个人。第二，团队决定的估算可能比个人估算更有用。

既然故事估算是团队的责任，那么团队大部分成员都参加估算是非常重要的。如果团队很大(可能 7 个或更多)，不需要每个开发人员都参加，但一般情况下，参加的开发人员越多越好。程序员估算时，客户也可以参加，但是他不能提出他个人的估算或者在听到自己不赞成的估算时发表意见。

估算

我喜欢的估算是来自 Boehm(1981)中的 Wideband Delphi 方法。与采用迭代方式进行开发软件的极限编程类似，我们也使用迭代方法进行估算。具体做法如下。

首先，把所有参与估算的客户和开发人员聚集在一起。带上故事卡和一些额外的空笔记卡——(即使你用电脑记录故事描述，还是应该带上一些空卡片)。发给每个参与者一些空卡片。客户随机抽取一个故事，读给开发人员听。开发人员根据需要尽可能多发问，客户要尽其所能解答。如果客户不知道答案，可以先猜猜看，或者要团队推迟估算这个故事。

所有事情都要花 4 小时

我最喜欢的电视剧之一是 *Mad About You*，主角是住在纽约的一对新婚夫妇。有一集中，妻子缠着丈夫去买沙发。她坚持认为只需要花一个小时。丈夫告诉她“这个世界上所有的事情都要花 4 个小时。你会去那，做各种事情，吃东西，谈论你其实应该在哪里进餐更好，然后回家。这至少是 4 个小时。”

程序员估算一个故事时，他们应考虑完成这个故事需做的所有事。他们要全盘考虑测试代码，和客户讨论，可能帮助客户计划或自动化验收测试等诸多因素。如果他们不将这些考虑在内，无异于他们期望只花一个小时买个沙发。

如果对故事没有疑问，每个开发人员在卡上写下一个估算值，先不要给其他人看。假如团队定义一个故事点为一个理想日的工作，开发人员则想想完成故事需要多少理想日。假如团队定义一个工作点为故事的复杂度，则估算值为开发人员所觉得的故事复杂度。

大家都写好估算值后，所有人翻开他们的卡片或者拿起展示给所有人看。这时，估算值很有可能相差很大。这其实是件好事。如果估算值不同，估算值高的和低的再解释一下估算依据。值得注意的是，这时不要互相攻击对方，而是耐心听取他们的想法。

举个例子来说，估算值较高的可能会说：“要测试这个故事，我们需要创建一个模拟数据库对象，这个可能需要1天时间。还有，我不确定我们的标准压缩算法是否能用，我们可能需要新写一个占用更少内存的算法。”估算值较低的可能回应：“我考虑把信息存在一个XML文件中，这样比用数据库简单。当然，我没有考虑到更多的数据，可能这是个问题。”

这时大家可以讨论几分钟。毫无疑问，其他人对两种极端估算的理由都有自己的看法。如果有什么问题，客户应该解释一下。可以在故事卡上加上一两个注释。也可写一两个新的故事。

在讨论完故事后，开发人员再次将估算值写在卡片上。当大家都写好修改的估算值，将卡再次展示给所有人看。在许多情况下，在第二轮估算值就差不多一样了。但是如果没有，重复让估高和估低的人解释他们的想法。多数情况下，估高和估低的人会改变他们的估算。事实上，有时我也曾碰到估高和估低的人在讨论中获得新的想法而走向另一个极端。

我们的目的是要为故事得到一个统一的估算值。这个过程很少会超过3轮，但是只要估算在不断接近一致，那我们就继续这个过程。没有必要让所有人都在卡上写下完全一样的估算值。要是我参与一个估算会议，在第二轮得到4，4，4和3个故事点，我会问估低的人是否愿意接受4的估算。点数要合理，而不是绝对精确。是的，开发人员可以畅所欲言，长时间讨论，最终在3或4个故事点上达成一致，但这样做并不值得。

三角测量

在做了几个估算后，我们可以(而且必要)对估算做三角测量。具体做法就是在估算一个故事时，根据这个故事与其他一个或多个故事的关系来估算。假定一个故事估算为

4 个故事点，第二个故事估算为 2 个故事点。把这两个故事放在一起考虑时，程序员应该都同意 4 个点的故事大概是 2 个点的故事的两倍。然后，如果一个故事估算为 3 个点，大家应该都同意这个故事大概比 2 个点的故事大，比 4 个点的故事小。

这些都不要太精确，但是三角测量是帮助团队验证他们没有逐渐改变一个故事点含义的有效方法。将故事卡根据他们的大小贴在墙上是个三角测量的好方法。在墙上划一些竖线，标出每列的故事点数，然后将故事卡贴到相应的列中，如图 8.1 所示。估算每个新故事后，将其放到相应的列中。可以非常快地比较出刚估算的故事是否和这列的其他故事大概相同。

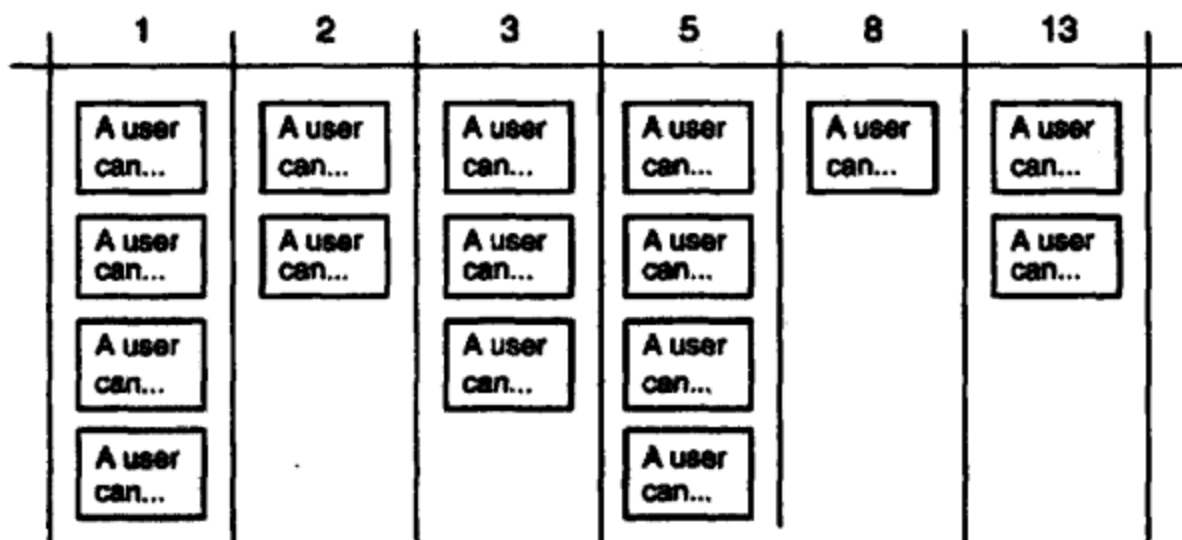


图 8.1 将故事卡贴在墙上以便于三角测量

使用故事点

在一轮迭代结束时，团队计算已完成的故事点数。因为即将到来的迭代也是同样的长度，这个点数可以作为下轮迭代将完成的故事点数的预报。举例来说，假如一个团队在一个 2 个星期的迭代中完成了 32 个故事点。那么很有可能他们下轮迭代也会完成 32 个故事点。我们用术语“速率”(velocity)来代表一个团队在一轮迭代中完成(或期望完成)的故事点数。

让我们看看如何用速率和故事点，以及为什么没有必要搞得那么精确。假如一个团队正在开始一个新项目。他们估算了项目的所有故事，一共 300 故事点。在开始第一轮迭代之前，他们计划一个星期完成 30 点，意味着他们需要 10 轮迭代(星期)完成项目。

在第一轮迭代结束时，团队将他们完成的故事点数加起来。他们得到 50 点，而不是

30 点。如果他们能每轮迭代都完成 50 点，那么他们一共需要 6 轮迭代来完成项目。他们应该按测量的 50 点作为速率计划么？是的，当然，基于以下 3 个条件。

首先，这轮迭代中没有发生异常事件(如加班，多了个程序员，等等)，因此生产力没有受到影响。加班或其他生产力因素对速率的影响是显著的。如果一轮迭代的速率是基于大家一个星期工作 60 个小时，当他们下轮迭代又回到一个星期工作 40 个小时时，速率会下降很多。

其次，必须采用前后一致的方式进行估算。这是非常重要的，因为这样能尽量减少从一轮迭代到下一个的速率的起伏。例如，在一轮迭代中，故事是被一个出了名的喜欢高估的人估算的。相对于之后的迭代由另一种倾向的人估算，团队在这轮迭代的速率会很高。保持估算一致性的最好方法是使用本章所描述的团队估算流程。

最后，第一轮迭代的故事必须是独立的。而且要按第 2 章建议的方式编写故事。想想一轮迭代中全是很糟糕的故事，如一轮迭代都在处理用户界面方面的事。我们将无法根据这轮迭代的速率来推断剩下的迭代。

为什么这个有用？

中央极限定理(Central Limit Theorem)告诉我们，任意分布的独立样本数量之和是符合正态分布的。

应用到我们这里，表达的意思就是团队的故事点估算可能倾向于低估、高估或任意方式的分布。但是当我们从这些分布中取出一轮迭代容量的故事时，取出来的故事是常态分布的。这意味着我们可以用一轮迭代的速率来预测未来迭代的速率。

一轮迭代的速率当然不是一个完美的预测。例如，包含一个 20 点故事的迭代比包含 20 个 1 点故事的迭代，预测精度显然要差一些。同样，随着团队学到新的技术、新的领域知识或者开始习惯与新的成员合作或习惯新的工作方式，速率可能会改变。

如果用结对编程呢？

团队用不用结对编程，对故事点估算并没有影响。例如，有个两个开发人员的团队按理想日来估算故事点。他们没有用结对编程。他们计划一个一星期的迭代，一共两个故事，每个估算 3 个故事点。在这轮迭代，他们完成了这两个故事并算出其团队速率



为 6。

假定他们用结对编程并且以理想结对日来估算。他们研究故事后决定每个故事需要两个理想结对日。在这个一星期的迭代结束时，他们完成了这两个故事并算出其团队速率为 4。

虽然数字不一样，但在这两种情况中，速率都是一样的。这表明这两个团队在迭代中完成了同样的工作量，所以速度是一样的。即，团队可以选择以理想结对日或理想个人日来估算故事点，区别会表现在速率值上。

精度随故事大小增加而降低

用故事点估算时，有个问题是有些数值之间的区别很难判断。例如，开发人员正在研究一个故事，一个说这个故事大概 2 个故事点。另一个说要 3 个点。这个讨论意义在于：3 个故事点意味着比 2 个故事点多一半。很有可能两个开发人员讨论这个故事并争论两个级别之间的区别。

然而，现在如果程序员争论一个故事是 7 个故事点还是 8 个故事点。在大多数情况下，大数之间一点的区别是很小的，没有必要再讨论。争论故事是 7 还是 8 个故事点，说明我们的估算过程在追求不必要的精确度。

为了避免这样的情况 and 简化问题，团队可以商定约束估算只用一些预定的值，例如：

1/2, 1, 2, 3, 5, 8, 13, 20, 40, 80

这是有意义的，因为它反映了一个事实，就是估算越大，我们对这些故事知道的越少。如果团队要考虑一个史诗故事，他们就得决定是 40 还是 80，但是他们不必考虑是 79 还是 80。

一些提醒

有时使用故事点会觉得比较困惑。通常是因为对于故事点太过思前想后或想让故事点发挥更多的作用。要正确使用故事点，请记住下面几点。

- 你的团队的故事点和我的团队的故事点是不一样的。你的团队估算的故事有 3 个故事点，可能我的团队估算有 5 个故事点。

- 一个故事(可能是一个史诗故事)分解成一些小故事后, 这些小故事估算的总和不需要与开始那个故事或史诗故事的估算相等。
- 类似地, 一个故事分解成一些任务。这些任务估算的总和不需要与故事的估算相等。

小结

- 用故事点估算故事, 故事点是故事复杂度、工作量或工期的相对估算。
- 应由团队估算故事, 估算属于团队而不是个人。
- 通过和其他估算进行比较来进行三角测量。
- 团队是否使用结对编程对故事点估算没有影响。结对编程影响的是团队的速率, 不是他们的估算。

开发人员职责

- 负责用一个方式定义故事点, 并且对团队可用和相关的。努力保证这个定义的一致性。
- 负责给出诚实的估算。不屈服于诱惑或压力而给出低的估算。
- 负责以团队估算。
- 负责估算应与其他估算一致。即所有 2 点的故事都应差不多。

客户职责

- 负责参加估算会议, 但是你的任务是回答问题并澄清故事细节。你不必参与故事估算。

问题


- 8.1 在估算会议上, 有 3 个程序员在估算一个故事。他们分别估算故事为 2, 4 和 5 个故事点。他们应该用哪个估算值?
- 8.2 三角测量估算有何用途?



8.3 请定义速率。

8.4 团队 A 在上一轮两周的迭代中完成了 43 个故事点。团队 B 在作另一个项目而且有两倍的开发人员。他们同样在上一轮两周的迭代中也完成了 43 个故事点。为什么会这样？





第9章 发布计划

大部分软件项目以每 2 到 6 个月为一个新发布周期，这是最好的。某些网站项目可能会更频繁地发布，但即便如此，搜集相关的新功能并放进一个发布中是有益的。以产品的开发路线图开始规划发布通常很有帮助，路线图展示未来几个新发布中关注的重点。这个产品的开发路线图毫无疑问会不断改变——这是我们所期望的，因为这些改变表明我们更加了解自己的产品、它的市场以及我们开发产品的能力。

产品的开发路线图可以很简单，它可以是未来几个发布要关注的重点列表，或者像 Kent Beck 所称的“主题” (Theme)。例如，对于 BigMoneyJobs.com 网站的下一个发布，我们可以列出以下主题故事。

- 为公司提供简历过滤和筛选工具。
- 为求职者提供自动搜索代理。
- 提高查询性能。

从一份笼统的开发路线图开始，我们使用以下两个问题来启动发布计划。

- 我们想在什么时候发布？
- 每个故事的优先级是什么？

一旦得到这些问题的答案，就可以通过估算团队能在每轮迭代中完成多少工作来计划发布了。利用我们能在一轮迭代中完成多少工作量的估算，我们可以做一个合理的预测，即完成符合用户期望的发布需要多少轮迭代。

我们想在什么时候发布

理想情况下，开发人员和客户可以谈一个日期范围，而不是一个具体日期：“我们希望在 5 月份发布，但只要我们能在 7 月的某个时候发布也可以接受。”可迭代的、由故事驱动的过程使我们很容易确定一个日期，确定在指定日期里交付哪些功能却比较困难。如果一个团队做发布计划时能以一个可接受的日期范围为起点，那么他们的发布时间将更灵活。例如，以一个日期范围开始，能让团队做出这样的承诺：“在 6 或 7 轮迭



代之后，我们应该有最基本的功能；或许 10 到 12 轮迭代后，我们会有 1.0 版本所有的功能。”

在某些案例中，日期确实是固定的。最常见的情况是，为行业展览准备的发布、关键客户的发布或者类似的其他里程碑式的版本。如果这样，发布计划实际上会更容易，因为需要考虑的因素较少。但是，决定在发布中包含哪些故事通常更加困难。

希望在发布中包含哪些功能？

为了计划一个发布，客户必须排列故事的优先级。把故事划分成诸如高优先级、中等优先级和低优先级这三种类型是很有用的，但这会导致乏味冗长的争论，会针对某个故事是高优先级还是中等优先级而争论不休。幸运的是，我们可以借用来自 DSDM 的方法，它是另一种敏捷过程^①。

DSDM 包括一个排列优先级的方法，称之为莫斯科(MoSCoW)规则。MoSCoW 是以下短语的缩写：

- 必须有(Must have)
- 应该有(Should have)
- 可以有(Could have)
- 这次不会有(Won't have this time)

“必须有的功能”是指系统的基本功能。“应该有的功能”是指很重要但短期内有替代解决方法的功能。如果项目没有时间约束，通常认为应该有的功能是强制性的。“可以有的功能”是指如果没时间就可以在发布中不予考虑的功能。列为“不会有的功能”是客户期望拥有但同时承认需要在后续发布中实现的功能。

排列故事优先级

我们可以通过多个维度来为故事排优先级。可以利用的技术要素如下。

- 故事不能如期完成的风险(例如，需要有预期的性能特点或全新算法时)。

^① 关于 DSDM 的信息，参考 *DSDM: Business Focused Development*, Stapleton 2003

- 推迟实现一个故事时对其他故事的影响(我们不想等到最后一轮迭代才知道,应用程序是三层结构,并且是多线程的)。

此外,客户和用户对故事进行优先级排序时,也会有他们自己的要素,如下所示。

- 故事对于广泛用户或客户的重要性。
- 故事对于少部分重要用户或客户的重要性。
- 故事与其他故事的内聚性(例如,故事“缩小”(Zoom Out)本身可能并不是高优先级的,但可以将它看成是高优先级的,因为它是高优先级故事“放大”(Zoom In)对应的功能)

总的来说,开发人员实现故事时会有一个顺序,就像客户所希望的那样。当客户和开发人员对这个顺序有不同意见时,最后每次都应该是客户说了算。

但是,客户在没有从开发团队那里获得某些信息之前,很难确定故事的优先级。至少,客户要知道每个故事需要大约多久才能完成。在确定故事的优先级前,应该先估算它们,并且在故事卡上写下估算,如故事卡 9.1 所示。

故事卡 9.1 提供链接, 退回先前查看过的项目

网站总是列出购物者最近查看过的 3 个项目,并提供链接退回。(即便购物者处在不同的会话里,这个功能也能正常工作)

估算: 3 天

此时,客户不会把对故事的估算加起来,然后决定一个发布中包括什么,不包括什么。而是利用这些估算,结合自己对于每个故事价值的评估,把故事进行优先级排序,使交付给自己公司的价值最大化。一个特别的故事对于客户公司可能极有价值,但需要花 1 个月时间来开发。一个不同的故事可能只有它一半的价值,但可能只要 1 天时间就能开发完。

成本改变优先级

几年前,我的团队为一个客户开发过一个 Windows 用户界面,这个客户是从一个老的基于 DOS 系统的大应用程序上转移过来的。在 DOS 系统里,回车键(Enter)用来在不同的字段中向前移动。客户想让我们在她的新 Windows 操作系统中保持这个功能。从她的客户角度来看,这是合乎逻辑的,无论是使用回车键还是跳格键(Tab),需要的开发时



间是相同的。然而，我们估计，使用回车键，将占用大约额外的 1 个人/周。在听到这个估计后，客户很快就降低了那个故事的优先级。当她认为这只需要几个小时的时候，那个故事就是高优先级的；而当这需要一周时间的时候，她宁可选择一些其他的功能。

混合优先级

如果客户在确定一个故事的优先级时遇到问题，可能需要分割这个故事。分割故事能使客户对独立的故事排列出不同的优先级。在一个项目中，我有一个故事(如故事卡 9.2 所示)。客户对故事的优先级举棋不定，因为根据作者和标题搜索是必需的功能，而其他搜索字段虽然不错，但不是必需的。这个故事被分成 3 个故事：一个故事根据作者和标题搜索；另一个根据出版物名称和日期搜索；第三个故事则允许根据组合条件搜索。

故事卡 9.2 搜索条件

用户可以根据作者、出版物名称、标题、日期或任意这些条件的组合来搜索杂志文章。

高风险故事

回顾软件开发的早期方法，很明显，大家一直在争论一个问题，即对一个项目来说，应该先做最有风险的部分，还是先做最有价值的部分。风险驱动开发的主要领导者也许是巴里·鲍依姆(Barry Boehm)，他的螺旋模型(spiral model)着重在早期消除风险(1988)。而另一个方向的领导者是汤姆·吉尔伯(Tom Gilb)，他提倡先做“油水最多”(“juicy bits”，1998)的部分。

敏捷方法旗帜鲜明地支持先做最有价值的部分。这让敏捷项目能够避免过早地解决风险，同时也推迟了可能并不需要的一些基础性代码的开发。赞同先做最有价值的部分，还能使项目可能尽早发布，那时只提供最有价值的功能。

但是，即使以价值优先为导向，我们在排列故事优先级时仍然需要考虑风险。许多开发人员倾向于先做风险最高的故事。有时这是恰当的，但这仍然必须由客户决定。然而，在排列故事优先级时，客户应该考虑技术团队的意见。

在一家生物技术公司最近的项目中，有些故事需要给一个称为“期望最大化”

(Expectation Maximization)标准统计方法做全新的扩展。由于这个工作确实是全新的，所以团队无法确定是否，或者需要多少时间完成。即使不包括这些故事，产品仍然是有销路的，所以客户把它们排列在中等优先级范围。但是，一旦客户意识到这些故事伴随的高风险，就会为这些故事的大部分排列更高优先级，期望由此来确定在开发新算法时会牵涉到哪些部分。

根据架构需要安排优先级

高风险故事经常与基础性或非功能性需求有关，如性能需求。我曾经参与开发过一个网站程序，它能显示股票价格图。其中有一个故事如故事卡 9.3 所示。对于已经指定的基准网络服务器机器而言，这个级别的性能要求是一个很大的挑战。满足这个性能需求的难度，对我们系统架构方面的决策有着深远的影响。

我们已经承诺使用 Java 作为我们的服务器端开发语言，但我们用 Java 能实现每秒 50 幅图像吗？我们要使用原生的 C 或 C++ 代码来生成图像吗？或者我们能否使用一个强大的缓存算法，对于相隔几秒的请求，输出相同的图表，以此来达成我们的目标。

故事卡 9.3 每秒产生 50 幅图像

每秒能产生 50 幅股票走势图。

这个案例中，客户已经为我们编写了故事卡 9.3。然而，她将这个故事的优先级排列得相当低。我们计划最初的几轮迭代先开发那些能展示前景并且能用于体现产品的最初卖点和好处的功能。客户的理由是，我们总可以在今后增加产品的可扩展性。在某些情况下，很容易重构系统以改善其扩展性。但在另一些情况下，重构可能会非常困难。开发人员应该帮助客户识别那些可以推迟实现，但越晚实现开发成本可能会非常高的故事。但是，开发人员不能滥用这种影响力，引导客户同意尽早实现他们喜欢的技术性功能。

在另一个项目中，客户明确表示应用程序以三层结构发布，有数据库服务器、客户端机器以及中间层用以路由客户端和服务端之间的请求和数据。客户和开发团队在不同的会议中讨论过这个问题，客户准备的销售文档也将系统描述为三层结构。但是，客户没有写下任何故事以表明需要增加中间层。

对于技术团队而言，这是一种困扰。他们并不介意从一个简单的二层结构系统开始



开发(数据库服务器和客户端机器),但在几轮迭代过后,他们越来越担心还没有加入中间层。他们知道加入中间层依然容易,但这会随着迭代的增加变得越来越困难。同时,由于用户故事完全专注于终端用户的功能,何时加入这样的基本需求并不是很清楚。

解决方法是,为帮助团队安排工作优先级的客户编写一个高优先级的故事,该故事要求系统有三层结构的能力。在这个案例中,我们加入故事“在安装过程中,用户可以决定在她的本地 PC 上安装所有组件,或者分开安装客户端程序、中间层程序和服务器端程序。”

选择迭代长度

开发人员和客户共同选择适合他们的迭代长度。迭代长度通常为 1 至 4 周。短迭代允许项目更加频繁地做出调整,项目进度也会更加透明;但是每轮迭代会有少许额外开销。假如不确定迭代长度,请选择短迭代而不是长迭代,使用长迭代更容易犯错。

在项目开发期间,尽可能地坚持固定的迭代长度。有了一致的迭代长度,项目会有固定的节奏,这有利于团队的开发速度。当然有时需要改变迭代长度。例如,有一个团队以 3 周作为一轮迭代,他们被告知要为 8 周后的重要行业展览准备下一个版本。不要在两轮 3 周的迭代后,就停止迭代,在展会前留下 2 周时间,他们可以从两轮正常的 3 周迭代开始,然后做一轮缩短的 2 周迭代。这不会有什么问题。不过需要避免随意改变迭代长度。

从故事点到预计工期

假设客户已经安排好所有故事的优先级。团队累加每个卡片的估算,总共有 100 个故事点。使用故事点使得估算故事变得更加容易,但现在我们需要一种方法,将故事点转换成项目的预计工期。

答案当然是使用速率。就像我们在第 8 章学到的那样,速率代表一轮迭代中能完成的工作量。一旦知道团队的速率,就可以用它将理想日转变成日历日。例如,假设估算项目需要 100 个理想日,如果速率是 25,我们就可以估算出完成项目需要 $100/25=4$ 轮迭代。

初始速率

可以通过下面三种方法获得初始速率。

1. 使用历史值。
2. 执行一轮初始迭代，使用那轮迭代的速率。
3. 猜测。

使用历史值无疑是最好的选择，但仅适用于现有团队正好刚刚做过类似的项目且没有成员加入或离开时。不幸的是，同一个团队接手两个连续且相似的项目非常罕见。

执行一轮迭代以获取初始速率是一个很好的方法。但很多时候，这个方法并不可行。例如，假设你的老板过来告诉你有新产品的点子。她写下了她认为在第一版中需要的故事。她已经对这些故事做过市场调查，认为这个产品第一年可以赚 50 万美金。如果这个产品的开发成本很低，公司就会开发它。如果不低，就不进行研发。老板询问开发成本如何时，你不能总是自如地说：“让我先执行一轮为期两周的样例迭代，然后再回来告诉你。”像这样的案例，需要一种方法来猜测速率。

猜测速率

如果我们需要猜测速率，这种方法至少应该是言之有理的，能清楚地跟别人解释。幸运的是，如果你遵循了第 8 章的建议，确实是有合理方法的，并且定义大约 1 个理想工作日为 1 个故事点。

如果故事点是一个理想工作日，我们可以通过估算完成一个完整的理想工作日实际需要多少天来估算初始速率。在迭代过程中，团队显然会受到许多干扰，阻止团队享受理想日。他们的实际工作日会与理想日有所不同，因为需要花时间回复电子邮件、打电话、参加全公司的会议、部门会议、培训、做演示或参加演示、给老板洗车、面试新的候选人、生病、放假等。因为所有这些干扰，把一轮迭代三分之一到一半的开发日作为预计速率是很常见的。例如，一个由 6 人组成的团队，使用为期 2 周的迭代长度(10 个工作日)，每轮迭代会有 60 个开发日。取决于他们预计的工作日与理想日之间的差异，他们可能想把速率估算为每轮迭代只能完成 20~30 个故事点。

当然，随着项目进行过几轮迭代以后，团队对于项目开发工期会获得更多经验。他们会知道在一轮或两轮迭代里，实际速率与估算速率相差多少，并且能够以此完善估算，



沟通计划时也会更加自信。

创建发布计划

因此，如果项目有 100 个故事点，若估算的速率是每轮迭代 20 个故事点，则可以预计总共需要 5 轮迭代。发布计划的最后一步是把故事分配到每轮迭代中。客户和开发人员一起协作，选择优先级最高的 20 个故事点，并且将它们放入到第一轮迭代中。下一组次高优先级的 20 个故事点放入第二轮迭代，如此进行直到分配完所有的故事。

取决于团队是否在同一地点工作(包括高层经理等利益相关者)以及组织所需的规范，有许多方法可以沟通发布计划。例如，我用过以下方法。

- 对于工作在一起的团队，我把故事卡钉在墙上，用列来表示迭代。
- 对于记录在电子表格中的故事，我根据它们的迭代进行排序，然后在每轮迭代的最后一个故事后画一条厚重的粗线。
- 对于有兴趣的远程利益相关者，我复印记录卡给他们(三张为一页，或者减小尺寸，6 张为一页)。
- 对于有兴趣的，比较讲究形式的远程利益相关者，我给他们创建简单的甘特图(Gantt chart)。创建诸如“迭代 1”的入口，然后在下方列出那轮迭代中所有故事的名字。

警告

小心，不要太迷信发布计划！本章描述的方法将帮助你估算项目所需的大致工期，让你可以声明“在 5~7 轮迭代后，可以准备发布产品”。但是，这些方法不足以精确说明“我们会在 6 月 3 日完成”。

利用发布计划可以设立初始期望，但之后如果获得新的信息，应该不断重新调整期望。监控每轮迭代的速率，只要了解到有其他因素影响估算，就应该重新估算故事。

小结

- 在计划发布时，有必要知道客户预期的大致发布日期和故事的相对优先级。
- 故事应该以明确的顺序排列(第一个、第二个、第三个，等等)，而不是利用诸如“非常高、高、中等”模糊顺序的分组。

- 故事的优先级由客户确定，但也要考虑开发人员的想法。
- 使用速率将以理想日为单位的估算转换成日历日。
- 估算团队的初始速率是很有必要的。

开发人员职责

- 负责提供信息(有时包括基本假设和可能的替代方法)给客户，以帮助她排列故事优先级。
- 负责在基础性需求或者架构性需求与其他客户需求之间取得权衡，避免不切实际地提高基础性需求或架构性需求的优先级。
- 建立发布计划时，负责在实际估算的基础上，适当包括一定长短的时间用以项目缓冲。

客户职责

- 负责以自己对故事价值的估计来确切排列用户故事的优先级。把故事排列为高、中、低这三个优先级是不够的。
- 负责诚实地表达发布期限。如果在7月15日需要，请不要为了保险起见而告诉开发人员6月15号就需要。
- 负责理解理想日和日历日的不同。
- 在想对故事的不同组件排列不同的优先级时，负责分割故事。
- 负责了解为何不应该谴责或批评一位个人速率为0.6的程序员，只因为她的速率小于1.0。

问题

- 9.1 估算团队初始速率有哪三种方法？
- 9.2 假设迭代以1周为长度，团队里有4位开发人员，如果团队的速率是4，项目总共有27个故事点，那么完成项目需要多少轮迭代？



第10章 迭代计划

利用发布计划，我们顺利地将粗粒度的故事分配到发布中的多轮迭代。这种层次的计划——不包含很多细节，可以避免给出精确需求的错觉，却足以根据它开始行动——适合作为发布计划。然而，在开始一轮迭代的时候，再做进一步的计划也很重要。

迭代计划概览

整个团队通过举行迭代计划会议来为下一轮迭代做计划。客户以及团队中的所有开发人员(也就是程序员，测试人员和其他人)都要出席参加这个会议。由于团队将仔细研究用户故事，所以毫无疑问他们会有一些问题。他们需要客户随时回答这些问题。

迭代计划会议的一般内容如下。

1. 讨论故事。
2. 从故事中分解出任务。
3. 开发人员承担每个任务的职责。
4. 讨论过所有故事，并且接受所有任务后，开发人员单独估计他们承担的任务，以确保他们不会做出过于乐观的承诺。

下面将逐一讨论每个活动。

讨论故事

团队获得一个已经排好优先级的故事集合，以此作为迭代计划会议的输入。正如程序员可能改变他们对实现一个故事难度的看法，客户一样可能改变她对故事优先级的想法。迭代计划会议是客户为团队调整故事优先级的最佳时机。

会议开始时，客户从最高优先级的故事开始，读给开发人员听。然后由开发人员提问，直到他们充分理解故事，能从故事中分解出任务。没有必要理解故事的所有细节，过分地深入每个故事的细节会让会议变得冗长、低效，因为会议中不是每个人都需要聆

听所有故事的所有细节。在计划会议后，开发人员仍能和客户一起理清故事的关键细节。

改变优先级

在迭代进行时，客户最好能保持克制，不去改变故事的优先级。如果客户在迭代期间频繁地改变想法，很容易使团队走上弯路。例如，在一个项目中，客户和程序员就数据库搜索功能如何工作达成了一致。10天的迭代过了5天后(搜索功能的编码工作完成了大约三分之二)，客户想出了一个与原先已部分实现的方案截然不同的方案，她觉得新的方案更好。那时，她认为自己正在比较两个都还没有开始编码工作的方案，显然，她偏爱自己觉得更好的方案。她敦促团队放弃目前的方法，并马上开始实施新的方法。我们礼貌地让她等到本轮迭代结束后再说，她同意了。到那个时候，她可以用一个已经完全运行的方案与另一个尚未开发的版本进行比较。前者可以做她想要的搜索功能的大部分事情，后者毫无疑问更好，但需要10天时间开发。

虽然她(以及团队中的大部分成员)认为新的搜索功能会更好，但将它与已经完全可以运行且功能足够完整的方案进行比较之后，发现这时再加入新的搜索功能并不值得。让开发人员集中精力于其他全新的功能，从而为用户提供更好的服务。

分解任务

将故事分解为任务没什么窍门。许多开发人员在他们的职业生涯中一直在那么做。由于故事已经比较小了(一般只占用项目普通程序员1~5个理想日)，所以通常没有必要进行更多分解。

事实上，为何要做分解呢？为何不直接把故事作为独立的工作单位呢？

尽管故事的确可以小到作为工作单位，但将它们分解出更小的任务，一般更符合项目的需要。首先，对于很多团队来说，实现故事的开发人员不止一个(或者是一对开发人员)。需要由多个开发人员共同完成故事，这要么是因为开发人员在某些特定技术上的专业性，要么是因为工作划分是完成故事的最快途径。

其次，故事是对用户或客户有价值的功能的描述，它们并不是开发人员的待办事项(to-do list)。把故事分解成任务常常是有用的，因为这有助于发现那些可能会被遗忘的任务。由于整个小组一起来划分任务，依靠的是所有人的集体智慧，尽管某个开发人员可能会忘记“有必要更新安装程序”是故事的一部分，但所有人都忘记却不太可能。

敏捷过程为人诟病的地方之一，是它没有像瀑布过程那样的前期设计步骤。这是事实，敏捷没有前期设计阶段，敏捷过程的特点是做频繁的短期设计。当脑海里至少有一个最小的设计方案时，才可能从故事中分解出任务。所以，一个故事的任务分解其实是即时设计(just in time design)中的一次短脉冲，而这些短脉冲的集合取代了瀑布过程的前期设计阶段。

当不同的团队成员说明构成故事的任务时，团队中需要有人记录下这些任务。我个人喜欢写在团队共享会议室的白板上。

作为从故事中分解出任务的例子，假设我们有一个故事“用户可以根据不同的字段搜索酒店”。该故事可以转化为以下任务。

- 编写基本搜索界面。
- 编写高级搜索界面。
- 编写搜索结果的界面。
- 为支持基本搜索查询数据库编写调试 SQL 语句。
- 为支持高级搜索查询数据库编写调试 SQL 语句。
- 在帮助系统和用户指南里写下新功能的文档。

尤其需要注意，任务中包含更新用户指南和帮助系统的任务。尽管这个故事没有明确说明任何有关文档的事情，团队从先前的迭代中得知系统有帮助系统和用户指南，在每轮迭代结束时，它们应该是准确的。如果对此有任何问题，团队应在迭代结束前询问客户。

准则

因为故事已经很小了，所以没有必要围绕任务的期望大小设定非常精确的准则。从故事分解任务时，使用以下准则。

- 如果故事的某个任务特别难于估算(例如，系统支持的数据格式列表，需要得到远程副总裁的批准)，就把那个任务从故事的其余任务中分离出来。
- 倘若有些任务可以很容易安排给多名开发人员共同完成，就分割它们。例如，在前面的案例中，编写基本和高级的搜索界面是分开的任务。尽管让同一个程序员或相同的结对(pair)程序员来完成这两个任务可能会有一些自然的协同作用，但这不是必须的。这样分解任务是有帮助的，因为这能让多个开发人员合作完成同一个故事。在迭代快结束时，这通常是必要的，因为就要没时间了。类似地，如果团队正在使用用户界面设计师或者专门的界面设计小组，任务“编

写基本的搜索界面”可以分成两个任务：“为基本搜索界面设计布局”和“编写基本搜索界面”。

- 若有必要让客户了解故事某一部分的完成情况，可以把那部分拿出来作为一个任务。在前面的例子中，编写基本的和高级的搜索界面是作为单独任务列出的。这将允许编写数据库接入代码的开发人员在搜索界面可用时，把她的 SQL 语句和搜索界面连接起来。这意味着，高级搜索界面如果推迟完成，并不会延误基本搜索界面两个任务的完成。

承担职责

一旦确定故事的所有任务，就需要有团队成员自愿执行每个任务。如果任务写在白板上，开发人员可以简单地把名字写在他们认领的任务旁边。

即使要采用结对编程，每个任务通常也最好只关联一个人的名字。此人将承担完成任务的责任。如果他需要从客户那里获得其他信息，他去负责好了。如果他选择结对编程，让他去找结对者。不过，归根到底，确保在迭代期间完成任务是他的职责。

实际上，确保完成任务是团队中每个人的责任。团队要有一种“同舟共济”的心态。而且，在迭代快要结束时，如果有开发人员不能完成他接手的所有任务，团队中的其他成员应该尽量勇于承担。

虽然任务是由每个人认领并承担责任的，但在迭代期间，这并不是一成不变的。在迭代中，随着开发取得进展，他们会更加了解任务，可能有些工作比预想的简单，但有些却比预想的困难，因此任务的认领及承诺也需要做出调整。在迭代结束时，不应该有人说“我完成了我的工作，但是 Tom 还有一些任务没有完成”。

估算并确认

假如一个项目团队的速率是每轮迭代 40 个故事点，那么不断重复前面的步骤——讨论故事、分解任务——直到团队讨论完客户提供的前 40 个故事点的故事。这时每个开发人员负责估算自己承担的工作量。最好的方法仍是以理想时间来估算。

此时任务应该足够小，以便做出可靠的估算。但即使不行，也不必担心。预测一下任务完成需要多长时间，然后继续前进。如果像本章早先推荐的那样，任务以及承担任

务的开发人员的名字写在白板上，每个开发人员就可以在白板上加上他或她的估算。结果看起来会像表 10.1 那样。

表 10.1 追踪任务很容易，开发人员执行每个任务，然后在白板上估算

任 务	责 任 人	估算时间(故事点, 单位: 个)
编写基本搜索界面	Susan	6
编写高级搜索界面	Susan	8
编写搜索结果界面	Jay	6
为支持基本搜索查询数据库编写调试 SQL 语句	Susan	4
为支持高级搜索查询数据库编写调试 SQL 语句	Susan	8
在帮助系统和用户指南里为新功能写下文档	Shannon	2

一旦开发人员估算好自己的每个任务，就需要把这些估算加起来，进而做出实际的评估，看看在迭代中能否完成所有的任务。例如，假设为期 2 周的迭代开始了，我已经承担了一个任务，目前我估算完成任务实际需要 53 个小时。算上我必须做的其他事情，我没有把握在这些任务上投入那么多时间。这种情况下，我有以下选择。

- 留着所有任务，寄希望于一切顺利。
- 请求团队中其他成员接手一些我的任务。
- 与客户讨论，放弃一个故事(或者分割故事，然后放弃其中一部分)。

假如有开发人员完成了自己工作量的估算，并且她觉得可以接手我的任务，完成这些任务的职责就转移到她身上。但是，如果没人有额外的时间来接手这些任务，那么客户就有必要帮忙从迭代中移除一些工作。每个开发人员必须能够放心地承诺完成自己将要承担的工作。而且，由于整个团队必须同舟共济，所以每个人都必须对整个团队做出的承诺有把握。

小结

- 迭代计划是发布计划的进一步计划，但只在迭代即将开始时才开始做迭代计划。
- 迭代计划中，团队讨论每个故事，然后从故事中分解出任务。
- 任务的大小没有强制的范围(例如，3 到 5 小时)。相反，从故事中分解出任务，用来帮助估算或鼓励多个开发人员合作完成一个故事。
- 每个任务都有开发人员承担。



- 开发人员通过估算他们承担的任务，评估他们是否承诺过度。

开发人员职责

- 负责参加迭代计划会议。
- 负责帮助把所有故事划分为任务，而不只是自己想做的故事。
- 负责为认领的任务承担责任。
- 负责确保承担适当工作量的工作。
- 在整轮迭代中，负责监控自己剩余的工作，同时也要监控队友剩余的工作。如果很快就能完成自己的工作，就有责任帮助队友承担部分工作。

客户职责

- 负责对迭代中包含的故事排列优先级。
- 负责指导开发人员交付他们能提供的最大商业价值。这意味着，从发布计划设定之后，若有更高价值的故事，要负责调整优先级以交付最大的商业价值。
- 负责参加迭代计划会议。

问题

- 10.1 从这个故事中分解出任务：用户可以查看酒店的相关详细信息。



第 11 章 测量并监控速率

正如第 9 章所讲到的，我们将项目分成一系列迭代来做发布计划，每轮迭代中安排一定故事点的任务。一轮迭代完成的故事点就是项目的速率。为这个项目做计划时，我们可以用已知的速率(如果有这样的速率，可能来自于另一个类似的项目)，我们也可以自己假想一个速率。速率是一个有用的管理工具，所以在每轮迭代结束和迭代中监控团队的速率是很重要的。

测量速率

因为速率是一个非常重要的度量，所以怎样测量它就变得很重要。多数故事很容易清点：团队在迭代中完成了这些故事，所以他们的点数全部计算在内。例如，假定在一轮迭代中一个团队完成了表 11.1 中的故事。正如表中所示，团队的速率是 23，这是一轮迭代完成的故事点数的总和。如果发布计划假定的速率和 23 差别很大，就有必要重新审视项目计划。但是，注意不要过早地调整发布计划。不仅仅因为是最初的速率往往不准确，而且速率在初期的迭代中也很不稳定。可能需要两三轮迭代之后，才能获得一个长期的、比较稳定的速率。

表 11.1 在一轮迭代中完成的故事

故 事	故 事 点
用户可以……	4
用户可以……	3
用户可以……	5
用户可以……	3
用户可以……	2
用户可以……	4
用户可以……	2
速率	23

但是，尚未全部完成的故事怎么办呢？是否应该把未完成的故事包括在计算中？

不行，不能将部分完成的故事也计算在速率中。这有几个原因。首先，一个显然的问题就是没法计算故事已完成的百分比。其次，我们不想使用像 43.8 这样带小数的值为速率引入错误的精度。第三，没完成的故事通常并不能给用户或客户带来任何价值。所以，即使我们为故事编写了部分代码，交付软件给用户时，这些故事将不包含在正式的迭代末版本中。第四，既然一个故事大到包含一个影响速率的子故事，如 41~50 之间，那只能说明这个故事确实太大了。最后，我们要尽量避免许多故事 90% 完成，没有多少故事 100% 完成的情形。最后 10% 的工作可能非常复杂，因此只计算全部完成的故事是很重要的。

如果实在想在计算速率的时候把部分完成故事的工作计算在内，可以先评估一下故事的平均大小，争取以后把故事划分得更小。在迭代结束计算速率时，忽略 1 个故事点故事的一半要比忽略一个 12 点的故事容易得多。另外，如果经常发现迭代结束时有太多尚未全部完成的故事(哪怕这些都是半点的故事)，这可能是团队内部缺乏合作的一个信号。用集中全部力量完成一个故事的方法会提高团队意识：大家一起先完成一些故事比所有故事都只完成一部分更有价值。

不用实际小时作为速率

注意，计算速率是用迭代开始前分配的故事点数。一旦迭代完成，就不要改变迭代中团队获得的任何故事点数。举个例子来说，假如一个故事估算是 4 个故事点，但其实更大。后来团队发现他们应该估 7 个故事点。在计算速率时，这个故事应该算 4 个点，而不是 7 个点。

通常情况下，应鼓励团队在为下轮迭代计划速率时不要超过上轮迭代的速率。然而，如果团队确实认为有个故事被严重低估，在下轮迭代中他们能做更多，就应该让他们计划一个略高的速率。

虽然团队不能返回修改一个已完成故事的点数，但他们应该用这类信息调整后续故事的估算值。

计划速率和实际速率

监测实际速率与计划速率的偏差，或者说，是否需要采取什么措施保证合理的速率，

这是很重要的。一个比较好的方法是为每轮迭代画出计划速率和实际速率。如图 11.1 所示，计划速率一开始低，但是之后开始增长并在第 3 轮迭代趋于稳定。

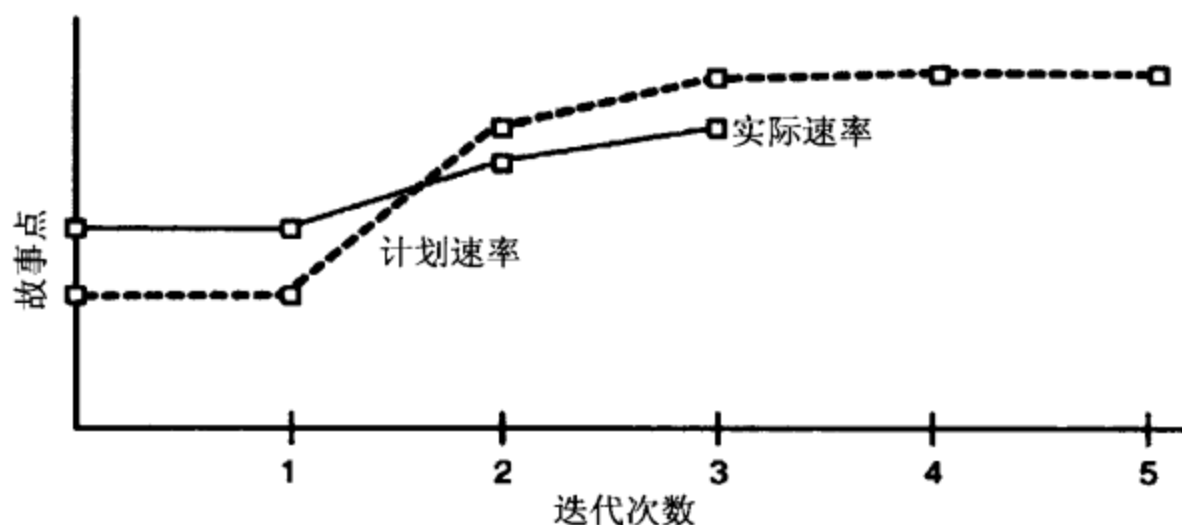


图 11.1 前 3 轮迭代后的计划速率和实际速率

图中画的第 3 轮迭代的实际速率超过了第 1 轮迭代的计划速率。然而，第 2 轮和第 3 轮迭代的实际增长不如计划，所以实际速率比计划速率略低。

图 11.1 中的团队如果按照第一轮迭代得出结论并告诉客户他们超过了计划速率可以将交付日期提前，那他们就错了。那看看 3 轮迭代后怎样？团队是不是可以说他们应降低客户对发布计划的期望？要回答这个问题，团队不仅需要图 11.1 中的速率图，还有图 11.2 中的累计故事点图。

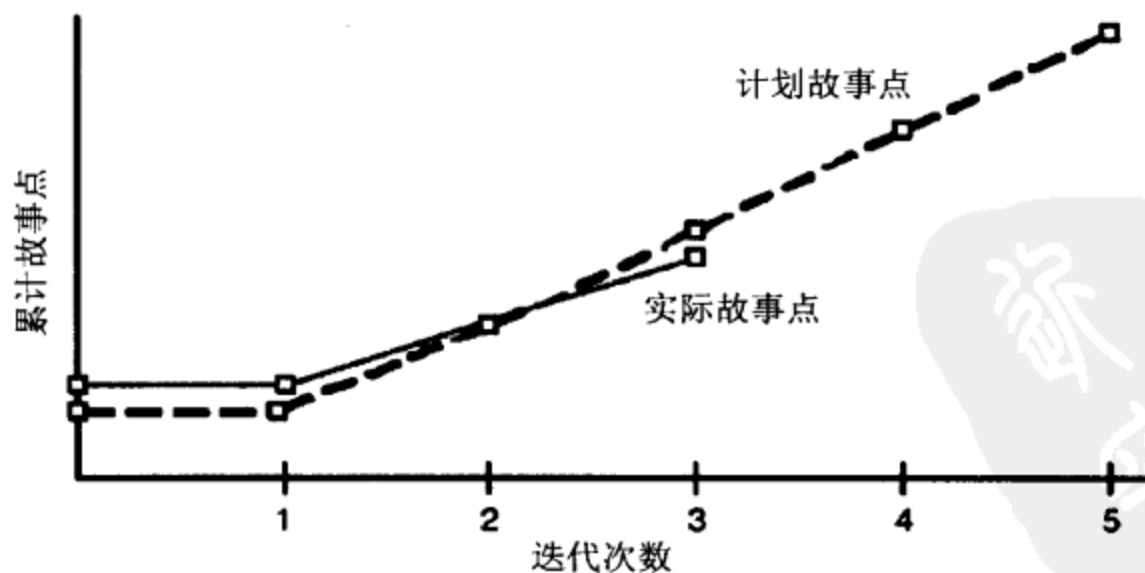


图 11.2 绘制累计计划故事点和实际故事点

累计故事点图表明了每轮迭代结束时总共完成的故事点数。由此，我们可以在

图 11.2 中看出尽管第 2 轮迭代的进展比计划慢得多，第 2 轮迭代结束后，团队实际完成的点数还是比计划的多。但是，在第 3 轮迭代结束后，团队在第一轮迭代建立起来的优势被第 2 轮和第 3 轮迭代的较慢进展蚕食殆尽。

在第 3 轮迭代结束时，团队很有可能不能按照计划完成那么多的功能。如果客户不能在每天与团队的沟通中意识到这点，团队应当及时让客户了解目前的状况。

迭代燃尽图

另外一个监控进展的好方法就是迭代燃尽图(Burndown Chart)。迭代燃尽图展示了以故事点表示的在每轮迭代末剩余的工作量。图 11.3 就是一个例子。

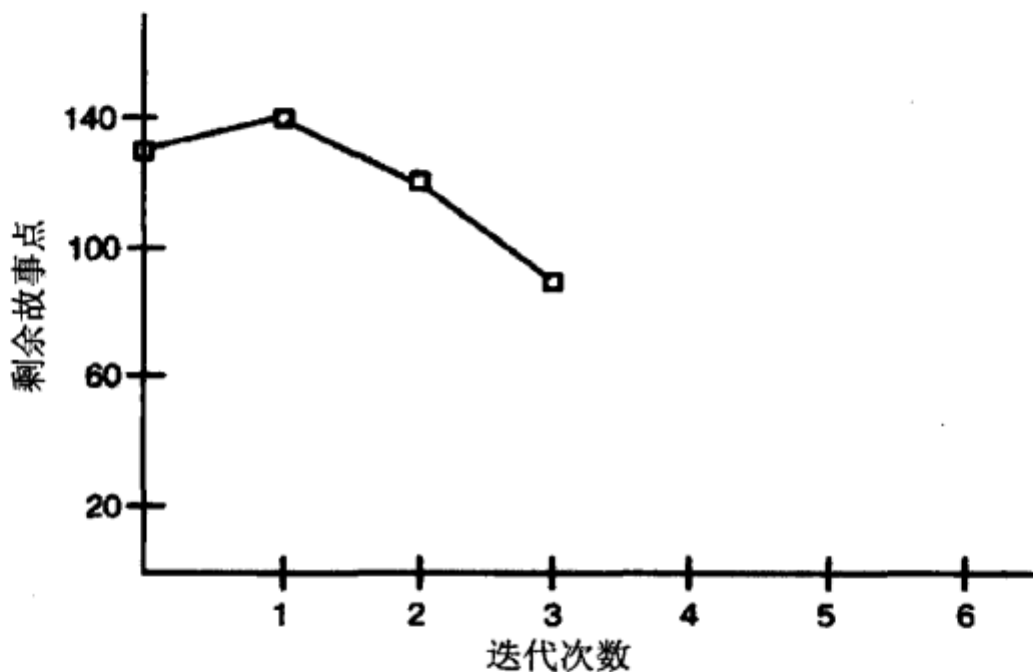


图 11.3 迭代燃尽图

燃尽图有一个有趣的特性，既可以从中了解到用已完成的故事点表示的进展，也可以从中了解到对发布剩余的计划故事点数的改变。例如，假定一个团队在一轮迭代完成了 20 个故事点，但是客户给项目加了 15 点的新任务。20 个故事点最后只得到了 5 点；当开发人员正在以最佳状态工作，如果客户期望项目能尽早完成，她可能不得不减慢引入新任务的速度。

从图 11.3 中可以看出，团队在第 1 轮迭代实际取得了负的进度。他们在第 1 轮迭代开始时需要完成 115 个故事点，在迭代结束时变成了 120 个。经理和客户需要注意，不

要不关注像图 11.3 中的燃尽图就去对团队嚷嚷。我们不能从燃尽图得到团队前进的速度。图 11.3 中的团队可能完成了 90 个故事点，然而可能加了 95 个故事点。想知道团队完成了多少故事点，应该看速率图(如图 11.1)或累计故事点图(如图 11.2)。

即使燃尽图不能表示团队的开发速度，但它还是很有用，因为它能更好地展现项目的整体进展。敏捷软件开发的一个优点就是项目开始时不需为项目需求写冗长完整的说明。敏捷团队都承认客户不可能预先知道所有事情的事实。因此，敏捷团队会要求客户提供尽可能多的信息，并允许客户在项目过程中修改或精练他们的想法，大家在一起学习如何构建软件。也就是说不断会有新故事涌现，也会有旧的故事因为变得没有价值而被取消，故事的规模和重要性也会不断进行调整。看看表 11.2 中的例子。

表 11.2 4 轮迭代中的进度和改变

	迭代 1	迭代 2	迭代 3	迭代 4
迭代开始时故事点	130	113	78	31
在迭代中完成的	45	47	48	31
修改后的估算	10	4	-3	
新加故事的故事点	18	8	4	
迭代结束时故事点	113	78	31	0

这个项目的团队认为他们可以每轮迭代做 45 个故事点。他们计划 3 轮迭代完成 130 个故事点。第 1 轮迭代他们确实完成了 45 个故事点。但是，在完成这些故事的同时，他们发现剩下的一些故事比一开始估算的要大，决定将这些还没开始的故事估算增加 10 点。另外，客户写了 6 个新的故事，每个估算为 3 点。所以即使团队完成了 45 个故事点，他们的净进度也为 $45 - 10 - 18$ ，即 17 点。也就是说在第 1 轮迭代结束时，他们还剩 113 个故事点。此时团队可以认为他们不能按计划 3 轮迭代内完成项目。即使没有加入新的故事，他们剩下的故事也超过了 90 点，90 点是他们在剩下两轮迭代能完成的合理期望值。客户和团队商量，考虑在 3 轮迭代后停下来——删除一些功能——但是，他们同意为该项目再多做一轮迭代。

第 2 轮迭代延续了第 1 轮迭代的趋势，团队完成了 47 点，但是没开始的故事估算增加了 4 点。客户减慢了变更速度，但依然加了 8 个点的故事。第 2 轮迭代的净进展为 $47 - 4 - 8 = 35$ 个故事点。

团队进入第 3 轮迭代时还剩 78 个故事点。一切都还顺利，团队完成了 48 个故事点。



同时还减少了剩下故事估算 3 个点。(想想前两轮迭代里, 剩余故事的估算一直在增加) 客户加了一到两个故事, 共 4 个故事点。第 3 轮迭代的净进展为 $48+3-4=47$ 。这样第 4 轮迭代就只剩 31 点, 只要再没什么变更, 团队应该可以完成所有任务。

图 11.4 是这个项目的燃尽图。如图所示, 从第 1 轮迭代的燃尽线的斜率可以看出项目不能在 3 轮迭代里完成。

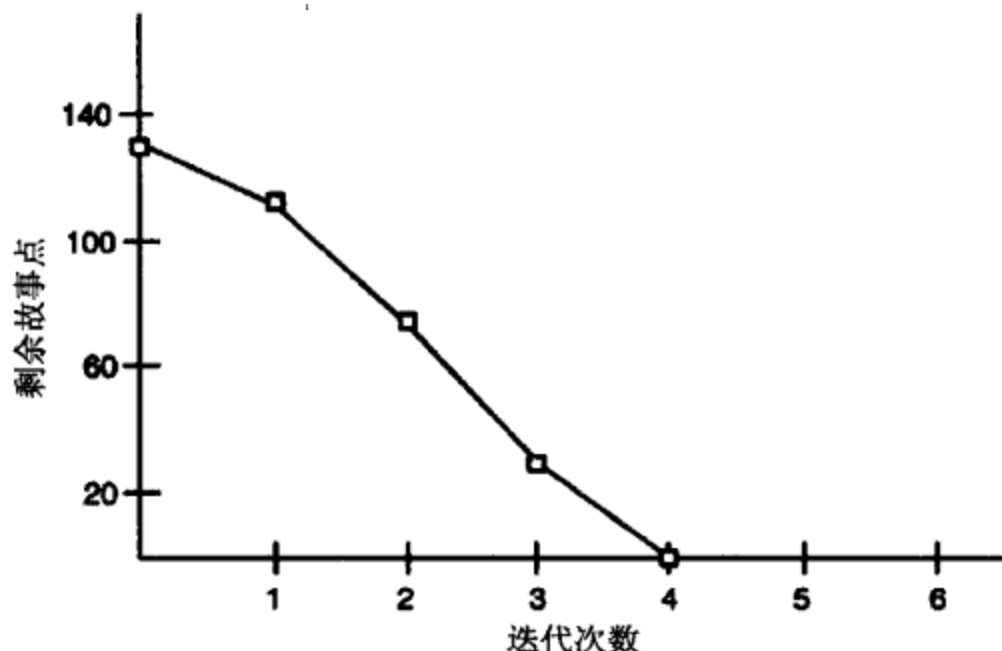


图 11.4 表 11.2 中项目的燃尽图

迭代中的燃尽图

燃尽图不仅可以用于在迭代结束时跟踪进展, 它还是迭代期间一种很好的团队自管理的工具。在迭代中, 每日燃尽图可以展现在迭代内剩余的估算小时。例如, 图 11.5 展示了一轮迭代中每日所剩的小时数。

我个人比较喜欢让团队成员在一个公共的白板上调整他所剩的小时数, 这样我就可以收集剩余工作量的信息。当迭代计划完成时, 在白板上加上写注释。此时板上有一个故事列表以及每个故事的任务。在任务旁有一块区域让程序员签名以认领任务。大约每天一次我会把白板上的数字加起来, 然后画到迭代的燃尽图上。在迭代开始时, 板上有类似图 11.6 的一块区域。

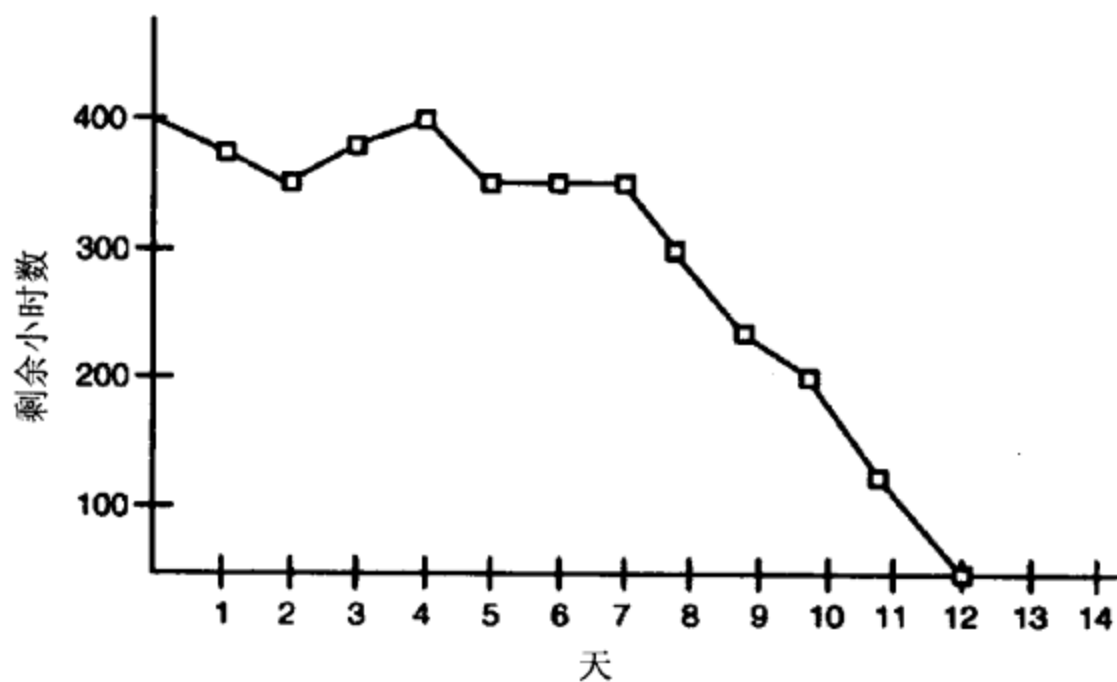


图 11.5 一个每日燃尽图

创建HTML页面	Thad	2	0
从HTML表单上读取搜索字段	Thad	2	
创建更好的样例数据	Mary	2	4
编写servlet来执行搜索	Mary	4	
生成结果页面	Thad	2	

图 11.6 在白板上写下估算并经常修改

从图 11.6 中可以看出，任务 Create HTML page(创建 HTML 页面)已经完成，剩余工作的估算从 2 个小时降至 0。然而，Mary 增加了任务 Create better sample data(创建更好的样例数据)的估算。不论 Mary 是还没开始就改变想法，还是她已经花了 2 个(或 4 个或 6 个)小时，觉得还需要 4 个小时，这些都没关系。重要的是白板上的估算反映出她现在对于还剩余多少工作的看法。

团队成员都明白需要保持白板上的数字反映相对最新的情况。最好是在完成任务时或者在每天结束时更新白板。这样就能保证反映最新的进展。应鼓励大家尽量准确地估算。让程序员觉得增加剩余时间估算和减少估算是一样正常的。

是剩余小时，而不是花费的小时

注意，每日燃尽图反映的是剩余工作量，而不是在一个故事或任务上所花费的工作量。可能跟踪花费小时数有些好的理由(如比较实际和计划的时间以提高估算准确度，或监测每个星期所花的生产时间)。然而这些好处远远不能抵消记录花费时间所带来的负面作用(如开发人员会觉得被管得很严，很死，以及这些数字并不准确)。

当然，也可以加新的任务。但是，只有在某人发现一个遗忘的任务，而且这个任务是当前迭代中完成某个故事必需的时候才能添加这个任务。如果只是某人想在迭代中加些新的东西，就不应该加这样的新任务，这样的调整应当通过在下次计划会议中通过调整故事优先级把该任务排入到某一轮迭代中。

使用大而且可视的图表

一个很有效的方法是把本章中所有的图做得很大并且放在大家都可以看见的地方。如果公司内部有绘图仪，就用绘图仪把图表打印出来并挂在团队公共区域或经常经过的走道的墙上。如果没有绘图仪，可以考虑在墙上挂一些大的白板，然后在上面画这些图。

在公共区域的一个地方，我们挂三个大的、4尺×6尺的白板。我用从文具店买来的黑胶带，在白板上画坐标。这样我们就得到了固定的坐标轴线，在修改图的时候不会被擦掉。每个星期，我们在三个图上加一个点。隔一段时间没有地方画的时候，我们就把图全部擦掉重画。

小结

- 计算速率时只考虑那些已完成的故事，即通过验收测试的故事。不要计算迭代中团队未全部完成的故事。
- 为每轮迭代计划和实际完成的故事点数画图是监测实际和计划速率区别的好方法。
- 不要在一两轮迭代后就忙着预测速率趋势。
- 完成一个任务或故事所花费的实际小时数对速率没有影响。
- 在大家都能看到的公共区域贴一些大而可见的图。
- 累计故事点图(如图 11.2)很有用，因为我们可以从中看出每轮迭代中完成的故事点。

- 迭代燃尽图(如图 11.3)展示了用完成的故事点表示的进度和剩余故事的改变。
- 每日燃尽图在迭代中十分有用,它展示了迭代中每天剩余的小时数。
- 设计及检测一个平均每个故事点出现的缺陷数目的图表可以帮助我们发现团队速率的提高是不是以牺牲质量为代价。

开发人员职责

- 尽量在完成一个故事后再去做下一个故事。我们更希望看到少量已完成的故事,而不是较多未完成的故事。
- 清楚所做的任何决定对项目速率的影响。
- 理解本章所讲的所有图。
- 经理或者极限编程项目中的追踪者,应知道怎样以及在什么时候画本章中的这些图。

客户职责

- 理解本章所讲的所有图。
- 知道团队的速率。
- 知道实际速率和计划速率的差别和是否需要调整计划。
- 为发布添加或删除故事,以确保团队在种种限制条件下尽量多实现项目目标。

问题

- 11.1 一个故事估算为 1 个故事点,实际花了两天完成。在迭代结束计算速率时应该计入多少呢?
- 11.2 哪些信息可以从每日燃尽图中得到,却不能从迭代燃尽图中得到?
- 11.3 从图 11.7 中能得出什么结论?像这样的项目是会提前完成,延期完成,还是按时完成?

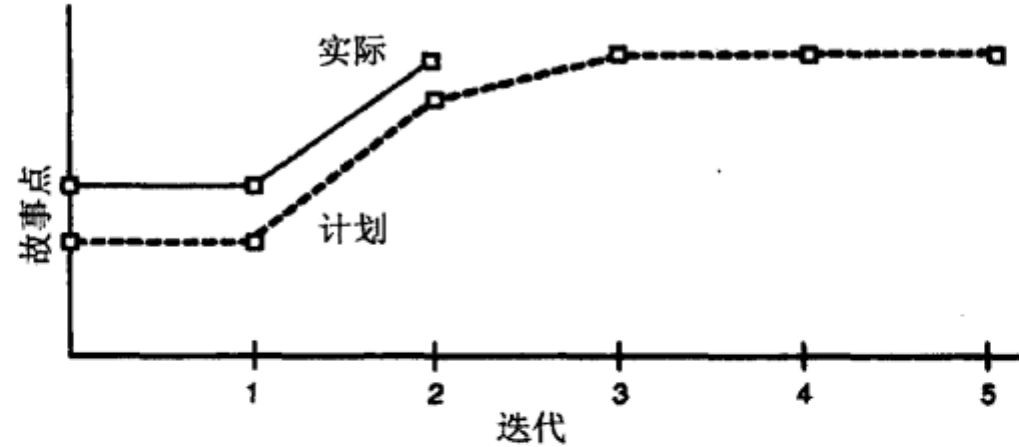


图 11.7 这个项目是会提前完成，延期完成，还是按时完成？

11.4 如表 11.3 所示的迭代，团队的速率是多少？

表 11.3 在迭代中完成的故事

故 事	故 事 点	状 态
故事 1	4	完成
故事 2	3	完成
故事 3	5	完成
故事 4	3	部分完成
故事 5	2	完成
故事 6	4	还没开始
故事 7	2	完成
速率	23	

11.5 在哪些情况下迭代燃尽图会有向上的趋势？

11.6 填写表中空缺的值，完成表 11.4。

表 11.4 填写空缺的值


	迭代 1	迭代 2	迭代 3
迭代开始时故事点	100		
在迭代中完成的	35	40	36
改变的估算	5	-5	0
新加故事的故事点	6	3	
迭代结束时故事点	76		0



第 III 部分 经常讨论的话题

到目前为止，我们已经讨论过用户故事的定义，怎样运用用户故事，怎样进行估算与计划。在本书第 III 部分，首先把注意力集中于用户故事与其他需求方法的区别，包括需求规格文档、场景以及用例。然后阐述用户故事相较于这些方法的优势。

对于任何方法，总会碰到不顺的情况，我们会看看发生问题时的一些不良征兆或者信号。用户故事起源于极限编程，并在其中得到比较多的应用和关注。在这一部分，我们也会探讨如何在另一个敏捷过程 Scrum 中使用故事。最后，我们以一系列其他比较小的常见主题收尾，例如用户故事应该写在纸上还是电子化，是否也用用户故事来描述缺陷，等等。



第 12 章 故事不是什么

为了帮助我们更好地理解用户故事是什么，首先应了解它们不是什么，这一点很重要。本章会解释故事如何区别于其他三种常见的需求方法：用例(use case)、IEEE 830 软件需求规格(software requirements specifications)和交互设计场景(interaction design scenario)。

用户故事不是 IEEE 830

电气与电子工程师协会下面的计算机学会出版过一本关于如何编写软件需求规格的指南(IEEE 1998)。这份众所周知的 IEEE 标准 830 文档，最后一次修订是在 1998 年。IEEE 的建议覆盖了诸如如何整理需求规则文档、角色原型和良好需求的特征等主题。IEEE 830 样式的软件需求规格，最突出的特征是使用短语“系统应该……”，这是 IEEE 建议的编写功能性需求的方法。一段典型的 IEEE 830 需求规则如下：

- 4.6) 系统应该允许公司使用信用卡支付招聘广告的费用。
 - 4.6.1) 系统应该接受 Visa 信用卡、万事达信用卡和美国运通卡。
 - 4.6.2) 系统应该在工作信息发布到站点之前从信用卡中支取费用。
 - 4.6.3) 系统应该给用户一个唯一的确认编号。

按照这种样式来记录系统需求乏味、容易出错，而且非常费时。此外，这样编写的需求文档，坦率地说，读起来很无聊。仅仅由于阅读起来很乏味就放弃一种方法，这个理由是不充分的。然而，如果你正在处理 300 页像这样的需求(这只是一个中等大小的系统)，就必须假设并不是每个人都将完整地读完。读者会略读或者索性跳过无聊的部分。另外，这种层面的文档，常常使读者无法理解全局。

警告信号

因为使用项目需求规格书而误入歧途的警告信号是，规格文档在开发小组和其他小组(如市场小组或产品管理小组之间)来回打乒乓。通常的情形是产品管理(或类似)小组编写了需求规格给开发人员。然后开发人员重写这份文档，以便表达他们对原来产品管理

组所写需求的理解。开发人员总是小心翼翼地给他们的文档起一个完全不同的名字(或许类似“功能规格”),以此掩饰它其实与原始的文档相同,只是从另外一个小组的角度来写罢了。

两个小组都知道,无论是否重要,任何项目的需求规格都是很难阅读和完全理解的,而且根本不可能把需求描述到想要的精度。因此,无论是哪个小组编写了最终的需求,它都能宣称握有该文档意图的归属权。当项目完成后,开始检讨过失责任时,他们会翻出文档的一部分,指出漏掉的功能在某处已经暗含了。或者根据文档里面的一句话,他们断言期望的功能明显超出了范围。

大部分时候,当我看到两个小组为基本相同的文档编写了单独版本时,我已经知道他们正把自己拽入到项目最后的责任推诿会议中,并辩称自己了解文档的意图。使用用户故事时,不会犯这种愚蠢的错误。随着用交谈代替文档,团队会发现没有必要追求一成不变。看起来像合同的文档总是让人觉得它是不可以改变的,交谈则不会给人这种感觉。假如我们今天讨论过了,然后下个月发现了新情况,没关系,我们可以再次讨论。

人们总是很倾向于通过思考,思考,再思考,构想出一个好的系统,然后把所有需求编写成“系统应该……”听起来会比“如果可能,系统将……”好得多,更不要说相对于“如果我们有时间,我们会试着……”的说法了。但第二种说法却能更好地刻画绝大多数项目的实际情况。

不幸的是,以这种方式编写系统的所有需求实际是一个不可能的任务。用户看到正在开发的软件时,会有有效和重要的反馈循环。用户看到软件时,他们会想出新的点子,并且改变他们之前的想法。需要修改用需求规格描述的软件时,我们习惯地称之为“范围变更”(change of scope)。这种想法是不正确的,有两个原因。首先,它隐含的意思是,在某个时候我们已充分了解了软件,其范围已经完全确定。不论前期投入了多少精力去考虑需求,我们知道,一旦用户看到软件,他们会有不同的(或更好的)意见。其次,这种想法迫使我们相信,当软件满足了需求列表,而不是满足了用户目标时,软件就算完成了。如果用户的目标范围改变,或许可以称作“变更范围”,但是事实上即使是对于特定软件解决方案的细节做改动,我们也经常用这个术语。

IEEE 830 样式的需求已经使许多项目误入歧途,因为它们侧重于关注需求的检查清单(checklist),而不是用户的目标。需求清单不会像故事那样给读者一个对产品的整体理解。读需求列表时,你会情不自禁地去考虑解决方案。Carroll(2000)认为“设计师可能只

会为他们遇到的前几个需求想出解决方案”。例如，考虑以下需求^①。

3.4) 产品应该有一个汽油发动机。

3.5) 产品应该有 4 个轮子。

3.5.1) 产品的每个轮子上应该安装橡胶轮胎。

3.6) 产品应该有一个方向盘。

3.7) 产品应该有一个钢制躯体。

此刻，我推断你的脑袋里浮现出汽车的图像。当然，汽车满足了前面列出的所有需求。你脑海中浮现的可能是一辆鲜红的敞篷车，而我构想的可能是一辆蓝色皮卡车。你的敞篷车和我的皮卡车之间的不同，很可能会在其他需求说明中涉及。

但是，假设用户没有编写 IEEE 830 样式的需求规格，而是告诉我们她对产品的以下目标：

- 该产品可以让我方便快速地修剪草坪
- 使用该产品时，我会觉得很舒服

通过察看用户目标，我们得到一个截然不同的视角，而且认识到客户实际上想要可骑的割草机，而不是汽车。虽然这些目标还不是用户故事，IEEE 830 文档其实是需求列表，故事则描述用户目标。通过从客户角度关注新产品的目标，而不是新产品的特征列表，我们往往能够设计出更好的解决方案，更好地满足用户的需求。

用户故事与 IEEE 830 样式需求规格的最后一个不同在于，对于后者而言，在写下所有需求前，每个需求的成本是不可见的。典型的情况是，一个或几个分析师花两三个月时间(通常更长时间)编写出冗长的需求文档。随后把文档交付给程序员。这时程序员告诉分析师(消息会被转告给客户)，完成项目要 24 个月，而不是分析师所希望的 6 个月。在这种情况下，分析师在编写团队没有时间开发的四分之三部分的文档上浪费了很多时间，而且在开发人员、分析师和客户之间反复讨论需要在有限的时间内先开发哪些功能上会浪费更多的时间。每个故事开始都会有一个估算。客户知道团队的速率，也知道每个故事的故事点数。编写完足够填满所有迭代的故事后，她知道自己的任务已经完成了。

Kent Beck 用登记结婚的比喻解释这个不同之处^②。当你登记婚礼时，你看不到每一

① 改编自 The Inmates Are Running the Asylum(Cooper, 1999)。

② 个人通信(Personal Communication)，2003 年 11 月 7 日。



项花费。你只是列出一个愿望清单。这对于婚礼可能行得通，但对于软件开发就不行。当客户在她的项目愿望清单中放入一项条目时，她要了解它的成本。

如何以及为什么要用该功能

需求列表更大的问题是，列表上的项目描述了软件的行为，而不是用户的行为或目标。需求列表很少会回答“但是，如何使用以及为什么有人需要这个功能？”

Steve Berczuk，极限编程开发者，《软件配置管理模式》的作者，认为这个问题十分重要：“拿着功能列表让客户给出使用那些功能的应用场景，多次让我节省了很多无用工作。客户经常会发现有些功能其实是不需要的，你应该把时间花在有附加值的事情上。”^②

用户故事不是用例

用例(use case)最初是由 Ivar Jacobsen(1992)采用的，如今用例已经被广泛地应用于统一过程(Unified Process)中。用例是对系统之间以及一个或多个用户之间交互的一般性描述，使用者要么是用户，要么是另外的系统。用例可以编写为非结构化的文本，或者符合结构化的模板。Alistair Cockburn(2001)提出的模板是最常用的模板。参见图 12.1 的示例，这个例子其实与用户故事“招聘者可以用信用卡支付招聘广告的费用”相同。

因为这不是一本关于用例的书籍，我们不去包括图 12.1 所示用例的所有细节。然而，还是需要探讨一下主要成功场景和扩展部分。主要成功场景是用例成功路径的描述。在前述例子中，在完成所有的 5 个步骤之后，系统成功完成了该场景。扩展部分定义了用例的其他路径。通常情况下，扩展部分用于错误处理；但是，也可以在扩展中描述第二条成功路径，比如图 12.1 中的 3a。用例的每条路径称之为场景(scenario)。因此，从 1 到 5 的序列代表了主要成功场景，1，2，2a，2a1，2，3，4，5 的序列则代表其他场景。

故事与用例之间最明显的区别是它们的范围。两者的大小都以交付商业价值为目标，但故事的范围更小，因为我们对它们的大小有限制(如不超过 10 个开发工作日)，以便用于安排工作。用例覆盖的范围几乎总是比故事大。例如，用户故事“招聘者可以使用信用卡支付招聘广告费用”，我们可以看到它同图 12.1 所示的主要成功场景类似。这导致了用户故事与单个用例场景类似的言论。每个故事未必等同于主要成功场景，例如，我

② Steve Berczuk 于 2003 年 2 月 20 日 extreamprogramming@yahoogroups.com 的发言。

们可以编写故事“当用户试图使用过期信用卡时，系统提示她输入不同的信用卡”，这等同于图 12.1 的扩展 2b。

用例标题：支付招聘广告的费用

主要使用者：招聘者

级别：使用者目标

前置条件：工作信息已经录入，但还不可以浏览。

最低保证：无

成功保证：发布工作；从招聘者信用卡中支取了费用。

主要的成功场景：

1. 招聘者提交信用卡卡号，日期和授权信息。
2. 系统验证信用卡。
3. 系统通过信用卡支取全部费用。
4. 求职者可以看到招聘广告。
5. 给招聘者一个唯一的确认编号。

扩展：

2a: 如果信用卡不是系统可以接受的类型：

2a1: 系统提示用户使用不同的信用卡。

2b: 如果信用卡过期：

2b1: 系统提示用户使用不同的信用卡。

3a: 如果信用卡没有足够的信用额度支付发布费用。

3a1: 系统从当前的信用卡上支取尽可能多的金额。

3a2: 系统告知用户这个问题，并要求输入第二张信用卡以支付剩余的金額。转到用例的第二步。

图 12.1 示例用例——支付招聘广告费用

用户故事和用例的不同还在于它们的完整性。James Grenning 注意到故事卡上的文本“加上验收测试基本相当于用例”^①，Grenning 的意思是故事对应于用例的主要成功场景，而故事测试对应于用例扩展。

例如，在第 6 章中，我们看到对于“招聘者可以用信用卡支付招聘广告费用”这个故事，比较好的验收测试如下。

- 用 VISA 信用卡，万事达信用卡和美国运通卡测试(通过)。

^① James Grenning 在 2003 年 2 月 23 日的 extreamprogramming@yahoogroups.com，指出过这点。

- 用大来卡(Diner's Club)测试(失败)。
- 用正确的、错误的和空的卡号测试。
- 用过期的信用卡测试。
- 测试不同的交易金额(包括超过信用卡额度限制)。

我们很容易看出这些验收测试与图 12.1 中扩展的相关性。

用例和故事之间另一个更重要的区别是它们的寿命。只要产品在开发或维护,用例常常作为永久性的“工件”持续存在。另一方面,故事不会超过包含它们的迭代。虽然可以对故事卡存档,许多团队却采用一种简单的处理方式,即撕掉它们。

另外的区别是用例比较容易包括用户界面的细节,尽管 Cockburn(2001), Adolph 和 Bramble 等(2002)告诫大家要避免这种情况。发生这种情况的原因有几个。首先,用例经常导致大量纸张的使用,并且没有其他合适的地方存放用户界面的需求时,它们最终便会放到用例上。其次,用例编写者会过早地关注软件实现,而不是去关注商业目标。

包含用户界面细节肯定会导致问题,尤其在项目早期,先入为主的想法往往使得用户界面设计变得更加困难。我最近碰到了图 12.2 所示的用例,其中描述了撰写和发送电子邮件的步骤。

用例标题: 撰写并发送电子邮件

主要的成功场景:

1. 用户选择菜单“新建邮件”。
2. 系统给用户展示“撰写新邮件”对话框。
3. 用户编辑电子邮件内容,主题和收件人。
4. 用户点击“发送”按钮。
5. 系统发送邮件。

图 12.2 撰写并发送电子邮件

这个用例贯穿设想的用户界面。它假设系统有“新建邮件”的菜单项,有撰写新邮件的对话框,对话框上有主题和接收人的输入字段,而且还有发送按钮。很多这样的假设似乎不错,而且没有问题,但它们可能没有考虑到一个用户界面,从这个页面,我可以通过点击接收人姓名而不是键入姓名来初始化一份邮件。此外,图 12.2 的用例排除了使用语音识别作为系统接口的可能。

诚然，绝大多数电子邮件客户端软件只支持使用手工输入，而不支持语音识别，但要注意的是在用例中对用户界面加以详细说明是不合适的。考虑一下替代图 12.2 的用户故事：“用户可以撰写并发送电子邮件”。没有任何关于用户界面的假设。使用故事，用户界面会在同客户的交谈过程中产生。

为了解决在使用用例过程中的用户界面假设问题，Constantine 和 Lockwood(1999)提出使用基本用例(essential use case)。基本用例剥离了关于技术和实现细节的隐藏假设。例如，表 12.1 展示了一个撰写发送电子邮件的基本用例。关于基本用例，一个比较有趣的地方是，用户意图可以直接诠释为用户故事。

用例和用户故事还有一个区别是，它们的目的不同(Davies 2001)。用例编写成客户和开发人员都可以接受的格式，如此大家都可以读懂并达成一致。其目的是记录客户和开发团队之间的协议。而编写用户故事是为了更方便发布计划和迭代计划，并且它充当着用户具体需求对话的占位符。

表 12.1 一个基本用例

用户意图	系统职责
撰写电子邮件	
注明接收人	
	收集电子邮件内容和接收人
发送电子邮件	
	发送邮件

并非所有用例都像图 12.1 所示那样编写成供填写的表单。有些用例编写成非结构化的文本。Cockburn 称之为用例摘要(use case brief)。用例摘要与用户故事有两点不同。首先，由于用例摘要必须依旧涵盖与用例相同的范围，用例摘要的范围通常比用户故事的范围要大。一个用例摘要通常可以陈述为几个故事。其次，用例摘要的生命周期与产品生命周期相同。而用户故事则会被扔掉。

最后，用例一般写成分析活动的结果。用户故事则写成注释，用以启动分析谈话。

用户故事不是场景

除了参考用例的单一路径，人机交互设计师也使用单词“场景”(scenario)。在这里，

场景是用户与计算机交互的详细描述。交互设计场景与用例场景是不同的。事实上，交互设计场景通常比用例更大或更全面。例如，思考一下这个场景：

Maria 正在考虑转行。自从辉煌的互联网时代泡沫时代开始，她一直在 BigTechCo 担任测试工程师。由于她原先是一所高中的数学老师，Maria 觉得假如回到学校教书，自己可能会更快乐。Maria 去访问 BigMoneyJobs.com 网站。她用用户名和口令创建了一个新账户。接着她创建了自己的简历。她希望在爱达荷州找一份数学教师的工作，但她偏向于靠近 Coeur d'Alene，这是她现在工作的地方。根据她的搜索条件，Maria 找到少数几个符合条件的工作机会。她最感兴趣的是北岸学校，一所地处 Boise 的私立高中。Maria 在 Boise 有个朋友叫 Jessica，她觉得 Jessica 应该认识北岸学校的人。Maria 输入 Jessica 的电子邮件地址，加上一段注释询问她是否认识学校的人，然后转发给 Jessica 这份工作的链接。第二天早上，Maria 收到一封来自 Jessica 的电邮，Jessica 说自己并不认识这个学校的人，但她知道这个学校，它有良好的声誉。Maria 点击按钮将自己的简历提交给北岸学校。

Carroll(2000)说场景包括以下特征性元素：

- 应用环境(setting)
- 使用者(actor)
- 目标或目的(goals or objective)
- 行动和事件(actions and event)

应用环境是故事发生的地方。在关于 Maria 的故事中，故事很可能发生在她家中的电脑上，但由于故事中没有提及，因此故事发生的地方也可能发生是工作日在她的办公室中。

每个场景至少包含一个使用者(actor)。一个场景可能有多个使用者。例如，在我们的场景中，Maria 和 Jessica 都是使用者。Maria 可以称为主要使用者(primary actor)，因为场景大多描述的是她与系统的交互。但是，由于 Jessica 收到来自系统的电子邮件，然后使用网站查看招聘广告，我们认为她是次要使用者。不像用例，交互设计场景中的使用者总是人，而不会是所有系统。

场景中的每个使用者可以寻求一个或多个目标。就像使用者，可以有首要目标和次要目标。例如，Maria 的首要目标是在心仪的地方找一份合适的工作。在努力达成首要

目标的同时，她也会有诸如查看酒店细节信息，或者与好友分享信息之类的次要目标。

Carroll 所说的行动和事件就像是场景的故事情节(plot)。它们是使用者采取的步骤用以达成她的目标或系统的响应。搜索在爱达荷州的工作是 Maria 执行的行动。对这一行动的响应是显示匹配工作清单的系统事件。

用户故事和场景的主要区别是范围和细节。场景包含更多细节，它们的范围通常涵盖多个故事。前面的示例场景包含许多可能的故事，示例如下。

- 用户可以通过电子邮件给好友发送工作信息。
- 用户可以创建简历。
- 用户可以对合适的工作提交她的简历。
- 用户可以根据地理区域搜索工作。

尽管场景往往包含更多的细节，它(像故事一样)也是鼓励通过讨论获得更多的细节。示例如下。

- Maria 用用户名和口令登录到站点。是否所有用户都需要登录？或者登录后是否允许 Maria 使用某些功能(或许发送邮件的功能)？
- 当 Jessica 收到邮件时，电子邮件是否包含工作信息，还是只包含一个到网站工作信息页面的超链接？

小结

- 用户故事有别于 IEEE 830 软件需求规格、用例和交互设计场景。
- 不管预想得多么全面，我们都无法事先完全定义一个完整的具有相当规模的系统。
- 在定义需求和用户早期频繁接触软件之间，有一个有价值的反馈循环。
- 考虑用户的目标比列出方案的特性更重要。
- 用户故事与用例场景类似。但用例往往仍然比单个故事要大，更容易包含关于用户界面的嵌入式假设。
- 此外，用户故事与用例的完整度和寿命不同。用例比用户故事完整得多。用例存在于整个开发过程中。用户故事往往只是暂时的，它们的生命周期仅仅限于开发它们的迭代中。
- 用户故事和用例以不同的目的编写。用例被编写成方便开发人员和客户讨论并




达成共识。用户故事编写成方便计划发布，并用于提醒需求细节的讨论。

- 不像 IEEE 830 规格和用例，用户故事不是分析活动的产物。相反，用户故事是进行分析的支持工具。
- 交互式设计场景比用户故事具体得多，它们提供的细节类型也不同。
- 典型的交互式设计场景比用户故事大得多。一个场景可以由多个用例组成，相应地，它可以组成许多用户故事。

问题

- 12.1 用户故事和用例的主要区别是什么？
- 12.2 用户故事和 IEEE 830 需求声明的主要区别是什么？
- 12.3 用户故事与交互设计场景的主要区别是什么？
- 12.4 对于一个重要的项目，为什么不能在项目启动时编写所有需求？
- 12.5 与琢磨待开发软件的特性列表相比，考虑用户的目标有哪些优势？





第 13 章 用户故事的优势

有那么多各种各样的处理需求的方法，为什么我们要选择用户故事？本章我们会看看与其他方式方法相比，使用用户故事到底能带来哪些好处。

- 用户故事强调口头沟通。
- 人人都可以理解用户故事。
- 用户故事的大小适合做计划。
- 用户故事适合于迭代开发。
- 用户故事鼓励延迟细节。
- 用户故事支持随机应变的开发。
- 用户故事鼓励参与性设计。
- 用户故事传播隐性知识。

讨论完用户故事的种种好处之后，本章结束时会给出用户故事几个潜在的不足。

口头沟通

人类曾经有非常出色的口头表达传统；最早的神话故事与历史都是通过一代一代口口相传下来。直到有一位雅典统治者为了防止遗忘，开始把荷马的《伊利亚特》写下来，像荷马这样的故事都是口口相传而不是通过阅读来传播的。过去人们的记忆力一定比现在好很多。从上世纪七十年代人们的记忆力开始衰退，要不然为什么我们经常连“系统应该提示用户输入登录名称及密码”这样的短句都记不住。我们只好开始把这些都记录下来。

从那时起我们就误入歧途。我们开始只关注一份共享的文档，而不是共有的认识。

我们想当然地以为：一旦我们写下一切我们所同意的东西，就不会再有任何不同的意见，开发人员确切地知道要开发什么，测试人员确切地知道怎样测试，而且最重要的是，客户以为他们一定会得到他们确切想要的系统。不幸的是，恰恰相反：客户只会获得开发人员根据他们对写下来的文档的理解所开发出来的系统，而这可能并不是他们

真正想要的。

在尝试之前，你或许会天真地以为只需要把一堆软件需求写下来，开发人员就能够精确地开发出你所想要的。然而，我们有时候连简单的午餐菜单都无法准确地描述，写出软件需求的难度可想而知。有一天午餐时，我看到这样一份菜单：

主菜配有汤或色拉和面包。

理解这句话本应没有任何难度，然而我可以选择的究竟是下面哪一种？

汤或(色拉和面包) 还是
(汤或色拉)和面包

我们常常以为白纸黑字的描述就是精准的，但事与愿违。与菜单上的描述不同，想像一下如果服务员直接问你：“你喜欢汤还是色拉？”更妙的情况是，为了完全消除任何疑惑，她可以在我点单前往我桌上送上一篮面包。

我们看到糟糕的文字表达可以代表多种意思。一个极端的例子，请看这两句话：

Buffalo buffalo buffalo.

Buffalo buffalo Buffalo buffalo.

我倒，这是什么意思！英语里 Buffalo 可以是指一种大型毛皮动物(也称作野牛，bison)，或者是纽约州的一个城市(水牛城)。又或者指“威慑”(intimidate)，譬如“开发人员在威慑之下承诺一个提早交付的日期”(“The Developers were buffaloed into promising an earlier delivery date.”)。因此，第一句话的意思是“一只野牛威慑另一只野牛”。而第二句话的意思是“一只野牛威慑一只来自水牛城的野牛”。

除非我们是在为野牛写软件，否则毫无疑问这是一个绝不会出现的例子，但是，难道它真的比下面这个典型的需求声明差吗？

- 当用户输入不正确的数据时，系统应该(should)突出地显示警告信息。

“应该”是否意味着如果我们愿意，就可以忽略这个需求？例如，我应该每天吃 3 份蔬菜，但我从不。另外，“突出显示”代表什么意思呢？对于写下需求的人和开发与测试人员来说，“突出”可能会有不同的理解。

我最近碰到另一个例子，涉及在一个数据管理系统里用户命名文件夹的功能。

- 用户可以输入一个名称。它可以是(can be)127 个字符。

单看这句话，我们不清楚用户是否一定需要为文件夹输入一个名称。可能系统提供一个默认的名称。第二句几乎没有任何明确的意思。文件夹名称究竟是一定要 127 个字符长度，还是可以有不同的长度？

把东西写下来确实有一些好处：写下来的文档可以帮助克服短期记忆的限制、干扰以及中断等。但是，如果我们更多地关注讨论需求而不是写下需求，很多起因不同的困惑都会消失，不论这些困惑是由文字的不精确性还是由其多重意思导致的。

不可否认，有些语言的问题在口语及书面中都存在。但我们知道当客户、开发人员及用户之间谈论需求时，短时间的即时反馈能促进相互学习与理解。文档所表现出的精准和精确的假象不会在谈话中出现。没有人会对一次交谈签名确认，所以也不会有人在后来拿出来指着说：“就是那次三个月前某周二，你说过密码不能有数字的。”

使用用户故事的目的并不是让我们事无巨细地记录下想要的特性；相反，用户故事作为提示语句提醒开发人员在将来需要与客户进行沟通与交谈。我经常通过电子邮件进行沟通，我无法想像离开它将能如何工作。我每天发送及接收数百封电子邮件。尽管如此，当我需要与人讨论复杂的事情时，我总是会拿起电话或者直接到对方的办公室或工作场所。

最近有一个关于传统需求工程的会议。会议里包括一个半天的环节指导大家怎样写“完美的需求”(perfect requirement)并且承诺传授如何写出更精准的语句以实现完美需求的技巧。但我认为，写出“完美的”需求永远都是高不可攀、遥不可及的。

就算需求文档里的每一句话都能够实现完美，还是存在两个问题。首先，随着软件的开发进程，用户会进一步了解软件，从而精炼他们的观点。其次，完美的部件并不能保证一定能带来完美的整体。《精益软件开发工具》作者 Tom Poppendieck 曾经提醒，100 只完美的左脚鞋不可能产出 1 双完美的鞋。相较于追求完美的需求，更有价值的是运用频繁的沟通来加强恰当的故事。

用户故事容易理解

相比 IEEE 830 样式的软件需求，用例(use cases)和场景(scenario)的好处是：用户和开发人员都能够理解。IEEE 830 样式的文档常常一方面有太多的技术术语令用户摸不着头脑，同时又用了不少的领域术语让开发者无所适从。

用户故事发扬了这个优点并且比用例及场景更加容易理解。Constantine 及 Lockwood(1999)察觉到场景方法由于过分强调现实与细节,从而使其更加宽泛的要点变得难以理解,这增加了人们尝试使用场景来理解人机交互本质内容的难度。而故事是简洁明了的,并且向用户或客户显示价值,业务人员及开发者理解起来通常很容易。

另外,上世纪 70 年代末的一次研究发现,若组织成故事,人们增强对各种事件的记忆(Bower, Black 及 Turner 1979)。而且,研究参与者能够更加清楚地回忆起事件中讨论清楚的情节以及潜在暗示的情节。这表明,故事不仅仅对回忆讨论清楚的情节有帮助,而且对回忆潜在暗示的情节方面也有用。我们可以写出比传统的需求规格甚至用例更加简洁明了的故事。需求是通过用户故事来展现以及进行沟通,从而加强了大家的记忆。

用户故事的大小适合做计划

用户故事的大小适合做计划,不太大也不太小。绝大多数开发人员都会碰到需要让客户或用户对 IEEE 830 样式的需求进行优先级排序的情形。通常的结果是:90%的需求是必需的;5%是非常想要的,尽管可以稍稍延迟;余下的 5%可以延迟久一些。这是因为对数千条以“这个系统应该……”开头的语句进行排序是非常不切实际的要求。例如,请考虑下面的需求:

4.6) 系统应该允许使用信用卡预订一个房间。

4.6.1) 系统应该接受 Visa 卡、万事达卡及美国运通卡。

4.6.1.1) 系统应该验证信用卡没有过期。

4.6.2) 系统应该在预订确认之前从信用卡中收取所有入住晚上的所示费用。

4.7) 系统应该给用户一个唯一的确认号码。

IEEE 830 需求规格中每一层次的嵌套层次都显示需求声明间的关系。上述例子中,想当然地认为客户能够分别对 4.6.1.1 及 4.6.1 进行优先级排序显然是不切实际的。既然这样的几个需求不能分别独立地被开发,可能就不应该分开这些需求。如果只是为了能够独立测试,还不如直接写测试。

考虑到某个典型产品的软件需求文档里成千上万条语句(以及它们之间的关系),要给这个需求列表按优先级排序显然是不可能的。

另一方面的问题是用例及交互设计场景通常太大了。对数十个用例或者场景进行排

序有时候不难，但其结果常常没有什么意义，因为通常并不是所有的最高优先级的事项都比下一级的一些事项更重要。为克服这个问题，很多项目尝试写出很多小粒度的用例，但这往往又有矫枉过正之嫌。

与用例和场景不同，用户故事的大小可以掌控，可以很方便地用于做发布规划以及进行编程和测试。

用户故事适合于迭代开发

与迭代开发相辅相成使用户故事具有很明显的优势。在开始编码之前，我并不需要写出所有的用户故事。我可以写出一部分故事，就进行编码与测试，然后按需要的节奏重复这个过程。写故事时，我可以按照认为合适的粒度去写。正是由于我们很容易对故事本身也进行迭代，所以用户故事很适合迭代开发。

例如，假设我正在开始考虑一个项目，我可能写出一个史诗故事“用户可以撰写并发送电子邮件”。这可能对非常早期的计划有用。后来我会把该故事分割成大概一打新的故事。

- 用户可以撰写一个邮件消息。
- 用户可以在邮件消息里包含图片。
- 用户可以发送邮件消息。
- 用户可以设定邮件发送的一个具体时间。

场景及 IEEE 830 样式的文档不支持这种递进式的细节层次。通过写出它们的方法，IEEE 830 类文档暗示如果一个需求语句不是以“系统应该……”开头，就可以认为“系统不应该……”这使我们根本无法知道究竟一个需求是漏掉了呢，还是只是没有被记录下来。

场景的强大之处在于其细节。如果我们想先写出一个没有细节的场景，然后期待按照开发人员的需要渐进地逐步增加细节，这是毫无意义的。这完全没有体现出场景本来的用处。

另外，我们可以写出不同层次的渐进式细节的用例，Cockburn(2001)提出了许多非常好的方法。可是，不同于使用自由格式，大部分的组织定义一个标准的模板，然后强行规定所有用例都需要符合这个模板。当人们觉得被迫填满一个表时会产生问题。



Fowler(1997)把这种现象称为“完整主义”(completism)。实际上,很少组织能够同时写出一些概要性层次的用例及另外一些包含详细信息的用例。用户故事适合于完美主义者,因为到目前为止,没有人建议使用一个充满字段的模板写出每一个故事。

用户故事鼓励延迟细节

使用用户故事的另一个优势是鼓励团队延迟收集细节。我们可以先写出一个起始的目标层面及占位意义的故事(例如“招聘者能发布一个新的工作”)。在这个故事在后来对于开发进程变得重要时,才需要用更多细节来代替这个简单的描述。

因此,用户故事非常适用于有时间限制的项目。团队可以非常迅速地写出数十个用户故事,让大家对要开发的系统有一个整体的概念,然后就能讨论前几个故事的细节并且开始编码。这远比觉得被迫先要完成一份 IEEE 830 样式的软件需求文档进展快许多。

用户故事支持随机应变的开发

相信我们能够写下一个系统的所有需求,然后从上往下想出解决方案,这向来就是一个美丽的陷阱。约 20 年前,Parnas 及 Clements(1986)已经告诉我们根本不可能有这样的好事儿,理由如下。

- 用户及客户通常都不会准确地知道自己的实际需求。
- 即便软件开发者知道所有的需求,很多需求细节只能随着他们的开发进展变得清晰。
- 就算假设所有的细节可以在前面弄清楚,人们也没有能力理解这么多的细节。
- 就算我们真的能理解所有的细节,产品和项目也会经常变更。
- 人非圣贤,熟能无过。

如果我们没有能力从上往下地开发软件,应该怎样做呢? Guindon(1990)的研究表明了软件开发者是怎样看待问题的。她请一小批开发人员解决一套楼宇电梯控制系统的设计问题。她把他们解决问题的过程进行了录像与观察。她发现他们完全没有遵从自上而下的方式。相反,开发人员使用一种随机应变的方式。这种方式中,他们对需求的思考不断地创建及讨论使用场景,在不同(opportunistic)层面的抽象的设计之间来回切换。一旦发现需要切换抽象层次,他们立刻就会切换思路。

使用故事承认并解决了 Parnas 及 Clements 提出的问题。由于用户故事十分重视口头交流，而且很容易写出或者重写出不同粒度的细节，所以用户故事让我们获得了一个解决方案。

- 不再需要依赖于用户预先完全了解及沟通他们的确切需求。
- 也不再需要依赖于开发人员能够完全事无巨细地掌握需求。
- 能够拥抱变化。

因此，用户故事提示我们用随机应变的方式开发软件。因为我们承认从高层次的需求到代码并没有一条严格线性的路径，用户故事是软件团队能够轻松地在高层及底层思考间来回切换并且不断地讨论需求。

用户故事鼓励参与性设计

像场景一样，用户故事很引人入胜。把交谈的重点从一个系统的特性转移到描绘用户使用该系统目标的各个故事，确实会使讨论变得更有趣。不少项目由于缺乏用户参与而失败；用户故事是一个吸引用户参与设计他们所需软件的便捷手段。

在参与性设计(Kuhn 及 Muller 1993;Schuler 及 Namioka 1993)中，系统的用户作为软件行为设计团队的一分子。他们的参与不是因为管理层的指令(“你们应该组成一个包括用户的跨职能团队”)，而是因为他们被所使用的需求及设计技巧大大吸引，从而自动成为团队密不可分的一部分。例如在参与性设计中，从一开始，用户就在帮助建立用户界面的原型，而并不是等到审核最初的原型后才参与进来。

与参与性设计相对立的是经验性设计，其间新软件的设计者通过观察未来的用户以及发挥软件效用的各种情形。经验性设计高度依赖访谈与观察，但用户不会成为软件设计的真正参与者。

因为在用户故事及场景中完全没有任何的技术术语，所以用户及客户可以完全理解。即便出色的用例可以避开技术术语，用例的读者通常必须掌握怎样解读用例的格式。对于一些用例的常见字段，例如扩展、前置条件及保证(extensions, preconditions and guarantees)，第一次看到用例的人很少能无师自通。而典型的 IEEE 830 文档在理解难度上更加变本加厉，既布满技术术语，又非常累赘冗长。

相比之下，故事及场景易读、易懂，因而能激发用户的积极性，成为软件设计的参与者。同时，随着用户渐渐掌握如何使用对于开发人员直接有用的故事来描绘他们的需



求时，开发人员与用户间的关系会变得更加密切和主动。这个良性的循环使所有开发中涉及的人或者软件使用者都获益良多。

用户故事传播隐性知识

缘于对面对面沟通的重视，故事能够促进团队内隐性知识的积累。开发人员与客户之间以及他们内部的沟通越密切，越多的隐性知识能得到传播与加强。

用户故事的不足

探讨过用户故事是敏捷需求推荐方式的诸多原因后，让我们看看用户故事的不足。

首先，在拥有大量用户故事的大型项目中，故事之间的关系可能错综复杂，难以捉摸。我们可以使用角色来淡化此问题，尽量保证用户故事不要过于细节化，直到团队开发这些故事时才开始细化。面对大量的需求时，用例的固有层级性会使需求收集比较方便。一个单一的用例，通过其核心成功场景及延伸，可以把相当多数量的用户故事汇聚为一个实体。

使用用户故事的另一个问题是，如果开发过程规定要有需求的可追溯性，必然离不开额外的文档。幸运的是，这可以用非常轻量级的方式解决。举个例子，在一个项目中，我们承担一个比我们大许多的 ISO-9001 认证过的公司的分包合同开发任务。他们要求我们呈现需求到测试间的可追溯性。我们采用一个非常轻量级的办法：在每一轮迭代开始时，我们生成一份文档，在其中列出该迭代中计划开发的每一个故事。随着测试的不断出现，我们把测试的名称添加到该文档中。在迭代进行期间，挪入或挪出故事时我们保持文档的更新。我们大概只需要在每月花费一个小时完成这一项额外的工作。

最后，虽然故事在一个团队内部能大大促进隐性知识的积累，但还是不适用于特大规模多团队的结构。这时，确实需要把有些交流记录下来，不然难以保证信息在大型团队中充分共享。当然，我们要在两种情况间取得平衡：很多人都知道一点点信息(通过低带宽的书面文档)；或者一小群人知道非常多信息(通过高带宽的面对面交流)。

小结

- 用户故事促使我们重视口头交流。与其他完全依赖书面文档的需求方法不同，用户故事认为开发人员与用户之间的交谈有重大的意义。
- 重视口头交流这一转变提供了迅速的反馈周期，往往能促成对需求的充分理解。
- 开发者与用户都能理解用户故事。IEEE 830 软件需求规格往往容易充斥着太多的技术或商业术语。
- 用户故事的典型范围规模比用例及场景小，但比 IEEE 830 文档大，这个大小适合于做计划。计划、编码及测试，都可以不再需要合并或分解故事就能完成。
- 用户故事适合于迭代开发，我们很容易从一个史诗故事入手，并在后来需要时把它分成多个小故事。
- 用户故事鼓励延迟细节。可以很快地写出独立的用户故事，而且写出不同大小的故事也很容易。对于不太重要的或者一开始不会被开发的需求，我们可以选择先用史诗故事来代表，而其他故事包含有更多的细节。
- 故事鼓励随机应变的开发。在此方式中，随着不断地发现机会，团队视线能迅速地在高层及低层细节思考间来回切换。
- 故事提升团队内隐性知识的水平。
- 用户故事鼓励参与性设计。与经验性设计不同，用户成为软件行为设计的活跃的参与者，做出有价值的贡献。
- 虽然使用故事有不少的好处，不可否认也有一些缺点：大型项目中难以组织好成千上万的用户故事；有时候需要额外的文档以实现可追溯性；尽管面对面的沟通大大促进隐性知识的共享，但在大型项目中，单单依赖这种交谈难于实现有效的扩展来完全替代书面文档。

开发人员职责

- 理解选择任何方法的原因。如果团队决定使用用户故事，需要明白为什么。
- 了解其他需求方法的优点以及知道什么情况下适合使用。例如，如果有客户无法理解一个特性，可能需要与其讨论交互设计场景或者开发一个用例。



客户职责

- 用户故事与其他需求管理方式相比的一大优点是其鼓励参与性设计。应该积极参与软件的设计。

问题

- 13.1 列举使用用户故事描述需求的 4 大优势。
- 13.2 列举使用用户故事的两个不足之处。
- 13.3 参与性设计与经验性设计之间有哪些主要区别？
- 13.4 “所有多页报表应该编号”这个需求语句有什么问题？



第 14 章 用户故事不良征兆一览

本章列出了使用用户故事时的一些不良征兆(smell)。这些不良征兆往往暗示着在项目中应用用户故事时出现了一些问题。我们将分析每一种不良征兆并给出一个或者多个解决方案。

故事太小

症状：经常需要调整估算。

讨论：小故事规模估算十分依赖于故事实现的顺序。实现顺序不同，规模估算值可能也有很大的差异，从而导致故事估算和计划出现问题。例如下面两个小故事。

- 搜索结果可以保存为 XML 文件。
- 搜索结果可以保存为 HTML 文件。

显然，这两个故事可以在很大程度上共享同一部分实现。实现了一个故事，只需要再花很短的时间就可以实现第二个故事。在做计划的时候，应该把这样的故事合并起来。如果把这个组合故事放到一轮迭代中，就可以把故事分成两个故事，但是，只要没有必要，最好不要划分这个组合故事。

故事互相依赖

症状：由于故事之间有依赖，所以很难做迭代计划。

讨论：两个甚至多个故事之间相互依赖，以至于很难把单独的故事放入迭代中。团队发现如果想要在一轮迭代中完成一个故事，就必须把另外一个故事也加入迭代中。如果要在迭代中加入这第二个故事，还要把第三个故事加进来，以此类推。故事过小或者划分不恰当的时候，容易出现这种问题。

如果觉得故事过小，解决方案很简单，就把互相依赖的故事合并成一个故事。如果

故事的规模刚刚好，就需要看一下故事划分是否恰当。第7章给出了如何像“切片蛋糕”那样划分故事的建议。好的故事应当包括整个系统的各个层面的功能。

镀金

症状：开发人员在迭代中实现了计划外的功能，或者仅仅凭借自己的感觉实现故事，实现的功能超出了实际的需要。

讨论：镀金指的是开发人员实现了不需要的功能。很多开发人员都愿意开发一些超出客户实际要求的功能。可能有几个原因。首先，很多开发人员总是希望给客户带来惊喜，但是敏捷软件团队客户的参与度往往很高。每天客户都会与团队有密切的交流，因此很难给客户带来较大的惊喜。

其次，在敏捷软件开发项目总是采用基于故事驱动的短迭代模式，开发人员总能感受到需要不断生产的压力。镀金功能使得开发人员得到一定程度的解脱。毕竟，即使不能按时完成这样的功能关系也不大，其他人甚至不知道已经着手开发这个功能。

最后，开发人员总是愿意在项目中留下自己的印记，添加一些“宠物功能”^①就是一种留下自己印记的方式。

一个镀金的例子

我曾经参与过一个项目，项目中的一个故事是重写一个十分拥挤的页面，把它写成基于标签的对话框，从而提高可用性。开发人员实现这个功能后，他把整个底层标签的代码也做了优化，实现了一个新的功能。可以从当前的位置脱离标签，并被挪到页面上任意的地方。这不是客户需要的功能。开发人员应该只关注客户认为的高优先级的故事。如果开发人员有了好的想法，她可以向客户提出建议，客户同意后，才能放到下一轮迭代中实现。

如果项目组中的开发人员花太多的精力去开发镀金功能，一个有效的解决方法是提高项目组中每个人的任务可见性。例如，可以要求每个人在每天会议上说明自己手头上的任务。每个人任务的可见性提高后，团队就自我约束，从而减少镀金。

① 译者注：客户并没有要求，但开发人员觉得炫的功能。

类似的，在迭代结束时的演示会议中，开发团队给用户以及其他利益相关者详细演示刚刚实现的功能的时候，也能发现哪些功能是一些镀金功能。尽管看起来比较晚，功能已经实现，但是可以让团队更意识到镀金的问题，从而避免以后再次发生。

最后，项目中测试团队也可以帮助发现镀金问题，尤其是测试人员能够及早与程序员和客户一起进行讨论需求的情况之下。

细节太多

症状：在实现故事之前花太多的功夫去收集整理故事细节。也有可能是花太多的时间去写故事而不去讨论故事。

讨论：把故事写到小卡片上的一个好处是在小卡片上只能记录下很有限的东西。很难把很多具体细节记录到小卡片上。如果在故事中包括太多的细节，这说明团队过于重视文档，而忽视了口头交流。

Tom Poppendieck (2003)评论说“如果总是需要写满小卡片，那下一次就用一个更小一点儿的卡片。”这是个好主意，因为这能够迫使故事作者有意识地减少记录过多的故事细节。

过早考虑用户界面细节

症状：在项目(尤其是开发新产品这样的项目)早期阶段编写的故事就已经包含用户界面细节。

讨论：在项目的某一阶段，开发团队一定需要用户界面的直接构想(或者直接的知识)。例如，“求职者可以在工作描述页面查看招聘公司的信息。”然而，只要有可能，尽量避免写这样包含这类具体信息的故事。

在项目早期，你不知道未来会有一个“工作描述页面”，因此尽量避免写关于这个页面的故事。取而代之，应该把故事写成“在查看工作描述的时候，求职者也可以看到关于招聘公司的信息。”



想得太远

症状：有几种表现形式。例如，小卡片空间小得难以记录用户故事的全部，又例如不是出于团队规模或者地域的考虑，希望用软件而不是小卡片来记录故事，又或者是某人建议用一个故事模板来捕捉故事相关的所有细节，又或者是建议用更精确的方法来估算(例如，用小时而不是用天为单位)。

讨论：这种不良征兆普遍存在于喜欢在项目前期花很多功夫整理需求的团队。这样的团队需要一个关于用户故事优点的进修课程。利用用户故事的关键在于承认事先不可能发现所有需求。好的软件是在不断的迭代中发展而来，而在每轮迭代中都会发现新的细节需求，然后被添加到软件中。用户故事与这种开发模式十分契合，就是因为我们可以不费力气地在未来的故事版本里添加细节。团队应该意识到采用用户故事的原因正是由于我们原来开发过程中存在着种种问题。

故事划分太过频繁

症状：在做迭代计划的时候为了确保迭代工作量合适，频繁地划分用户故事。

讨论：开发人员与用户一起为下一轮迭代选择故事时，有时需要把一个故事划分成两个或者多个子故事。在出现下列情况时，需要这样做。

1. 故事太大以至于不能在一轮迭代中完成。
2. 故事包括高优先级和低优先级的子故事，客户只希望在下轮迭代中完成高优先级的子故事。

这不是问题。考虑到 Sprint(Scrum 框架中迭代的称呼)长度和团队的历史速率的因素，很多项目和团队总会碰到需要划分用户故事的情况。但是如果太过频繁，就说明出现了一些问题。

如果感觉划分故事过于频繁，应该考虑扫描剩余的故事，找出真正需要划分的故事。

客户很难为故事安排优先级

症状：选择故事和给故事安排优先级并不是一件容易的事情，但是有时候如果安排优先级太困难，也可以认为是一种不良征兆。

讨论：如果客户认为很难给故事安排优先级，第一个可能的原因是故事太大。大的用户故事很难安排优先级。假设一个极端情况，BigMoneyJobs 网站仅仅包括三个用户故事。

- 求职者可以搜索工作。
- 公司可以发布职位。
- 猎头可以搜索求职者。

要给这三个故事安排优先级的客户真令人令人同情！她的第一反应可能是，“我不能只要这个故事的一部分，再要另外一个故事的一部分？”遇到这种情况时，需要扔掉这些故事，试着用一些小的故事代替，让客户可以选择她需要的功能。

此外，试图为体现不出商业价值的用户故事排优先级也很困难。比如下面两个用户故事。

- 用户通过连接池连接到数据库。
- 用户可以从日志文件中看到详细的错误信息。

客户会感到为这样的用户故事安排优先级很痛苦，因为对客户来说故事的商业价值太不清晰。需要重写这样的用户故事。但具体怎样重写，还要看客户技术能力的高低，这就是为什么最好让客户自己来写用户故事。比如，可以把前面的用户故事写成下面这样。

- 用户在启动系统时，感觉不到有明显的连接数据库的延迟。
- 系统出现错误时，用户能够获得足够的信息，找到修正错误的方法。

客户不愿意写用户故事，也不愿意为故事安排优先级

症状：项目的客户不愿意承担写用户故事和安排优先级的责任。

讨论：在一个总是互相指责的传统组织中，很多人认为最好能够不承担任何责任。如果不用为某件事负责，也就不会由于事情的失败而被指责。更有甚者，即使是获得成功，有些人也会吹毛求疵地去指责不足之处。处在这种文化氛围中的个人往往不愿意去做类似于把功能排入计划或者排除在发布计划之外艰难的决定。他们会说“你们能不能在项目截止日期前完成所有任务不是我的问题，你们自己想办法吧。”

我觉得对于类似问题最好的一个解决方案就是想办法让客户摆脱这样的困境。我找到一种让他轻松表达观点的方法。根据个体不同，可以通过私下讨论。如果我需要与多



个客户合作，我会跟每一个人说我也会从其他客户那里收集信息，但是由我自己为最终决定负责，尤其是在结果证明他们所说的并不正确时。

小结

在本章中，我们了解了以下一些不良征兆：

- 故事太小
- 故事互相依赖
- 镀金
- 故事中包含太多的细节
- 过早包含用户界面细节
- 想得太远
- 故事划分太过频繁
- 很难为故事安排优先级
- 客户不愿意写用户故事，为故事安排优先级

开发人员职责

- 与客户一起了解这些使用用户故事时的不良征兆，当然还包括其他种种不良征兆，然后找到这些不良征兆的解决方案。

客户职责

- 与开发人员一起了解这些使用用户故事时的不良征兆，当然还包括其他种种不良征兆，然后找到这些不良征兆的解决方案。

问题

- 14.1 如果团队总是发现很难做下一轮迭代的计划，应该怎么办？
- 14.2 如果团队总是觉得小卡片太小以至于写不下用户故事，应该怎么办？
- 14.3 客户在哪些情况下会觉得很难为用户故事安排优先级？
- 14.4 在什么情况下，划分的故事会被视为太多？

第 15 章 Scrum 与用户故事

本用户故事源自于极限编程，所以故事能够很自然地与极限编程的其他实践形成一个体系。不过，用户故事作为一种管理需求的方法，也可以应用到其他类型的软件过程中。

在本章中，我们来探讨怎样有效地把用户故事和另一种敏捷过程 Scrum^①结合在一起。Scrum 术语在本章中第一次出现时，将用斜体标注。

Scrum 是迭代和递增的

与极限编程类似，Scrum 也是一种迭代递增的软件过程。首先我们需要定义什么是迭代和递增。

一轮迭代的过程是一种持续改进的过程。开发团队首先针对系统的一部分开始开发，团队十分清楚系统还是不完备的，有些地方甚至比较差。接下来，团队需要逐步地完善整个系统，直到满意为止。通过一轮迭代，不断地给软件添加更多的细节，软件的功能也越来越完备。例如，在第一轮迭代，搜索页面可能仅仅支持最简单类型的检索。第二轮迭代，进一步支持更多的检索条件。最终，到了第三轮迭代，支持错误处理功能。

一个比较好的类比是雕刻。首先，选择一块大小合适的石头。接下来，雕刻家会在石头上雕出大体的轮廓。这时候，我们可以区分出头和躯干，可以分辨出最终会雕成一个人体而不是一只小鸟。然后，雕刻家会通过不断地添加细节来改进这件作品。然而，只有当整个作品都完成后才会认为作品的所有部分都已经完成。

一个递增的软件过程是指团队按照功能点开发和发布软件。每个功能点，或者称为功能增量，代表一个完整的功能子集。可能是大的功能增量，也可能是小的，小到简单的系统登录页面，大到一个高度灵活的数据管理页面。每一个功能增量都被完整地实现以及测试通过。因此，一轮迭代的工作往往不需要返工。

① 如果想要了解 Scrum，可以参考 Ken Schwaber 《Scrum 敏捷项目管理》一书(Schwaber 及 Beedle 2002)。

一个采用递增方式工作的雕刻家首先会确定作品的一部分，然后专注于这个部分直到完成为止。她可能选择小的递增(开始鼻子，然后眼睛，接下来是嘴，等等)或者大的递增(头，躯干，腿，接下来胳膊)。然而无论递增的规模大小，这个雕刻家总是尽量完成每一部分工作。

Scrum 和极限编程都是基于递增和迭代方式的过程。这两种过程都在一轮新的迭代开始之前为迭代做计划，并在后续迭代中改进以前的交付，而且总是在每轮迭代中把当前迭代所计划的工作做完，贯穿整个项目总是可以持续地交付。

Scrum 基础

实施 Scrum 过程的项目往往采用 30 天为周期的迭代，称为 *Sprint*。在每个 Sprint 开始的时候，团队需要确定这个 Sprint 需要完成的工作。所有工作内容放在一个称为产品 *Backlog* 的排好优先级的列表中。团队根据自己的经验从产品 *Backlog* 中选择下个 Sprint 能够完成的任务放到另外一个称为 *Sprint Backlog* 的列表中。团队每天都会有一个简单的会议，称为每日 Scrum 简会(*Daily Scrum*)，在会议上所有成员审查团队的进度，并根据需要做出调整。图 15.1 给出了一个基本的 Scrum 模型，这个模型是从 Ken Schwaber 的网站 www.controlchaos.com 上的模型修改而来的。

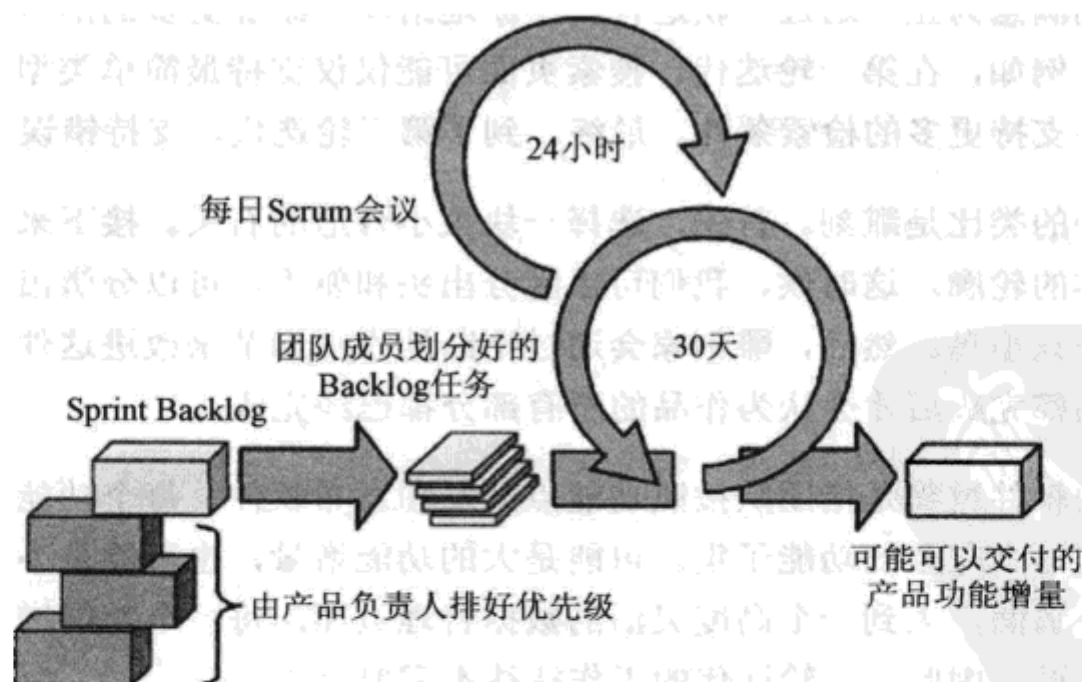


图 15.1 Scrum 过程的图例

Scrum 团队

一个 Scrum 团队通常由 4~7 个开发人员组成。尽管在 Scrum 团队中可能某些人有特长，如测试人员和数据库管理员，但整个团队总是同舟共济。如果需要完成测试任务，但是没有空闲的测试人员，其他的开发人员也会参与测试。大家共同负责结果。Scrum 团队往往是自组织的。也就是说，不会有管理人员指派 Mary 做编码，Bill 做测试，在 Scrum 团队中通常没有区分架构师和测试人员角色的说法。团队根据实际情况，自主决定怎样完成剩余的任务。

开发团队还有两个承担特殊角色的人员：产品负责人和 ScrumMaster。Scrum 产品负责人主要负责管理 Product Backlog 的内容以及排列优先级。ScrumMaster 的职能类似于项目经理，只不过他不是管理者的角色，更像一个领导者。由于 Scrum 团队采取自组织的方式，团队成员自己决定如何完成当前 Sprint 的任务，因此 ScrumMaster 更多的是为 Scrum 团队服务，而不是指手画脚。ScrumMaster 主要负责为团队排除障碍，保证开发的顺利进行，确保整个团队按照 Scrum 的简单规则进行开发。

产品 Backlog

产品 Backlog 是所有待开发产品功能的列表。在项目初期，一般不需要投入很大的精力写出所有的功能。通常，产品负责人和团队一起写下一些比较显而易见的功能，一般情况下，在第一轮迭代中不可能完成所有这些功能。随着开发的不断进行，产品负责人会根据产品的发展和从客户那里得到的反馈，不断对产品 Backlog 进行调整和扩充。

表 15.1 是一个实际项目中的产品 Backlog 示例。从这个表中可以看出，产品 Backlog 中有技术类的任务（“重构 Login 类，让它抛出异常”），也有面向客户的任务（“可以从安装页面取消安装”）。

产品负责人负责按照优先级对产品 Backlog 中的条目进行排序。甚至有些有经验的产品负责人会每个月根据产品优先级的变化，按照优先级对产品 Backlog 中的所有条目重新排列。



表 15.1 一个产品 Backlog 例子

编 号	描 述
1	完成数据库版本化
2	从数据库中删掉不用的 Java 代码
3	重构 Login 类, 让它抛出异常
4	支持并发用户授权
5	支持评估授权
6	在搜索时支持通配符
7	可以保存用户设置
8	可以从安装页面取消安装

Sprint 计划会议

在每个 Sprint 的开始是 Sprint 计划会议。这个会议通常持续一整天。会议的参加者包括产品负责人、ScrumMaster 和团队的所有开发人员。如果有兴趣, 其他的管理人员和客户代表也可以参加。

在 Sprint 计划会议的前半段, 产品负责人会把待开发的高优先级的功能介绍给 Scrum 团队。在第二个阶段, 团队成员可以针对第一阶段中介绍的每一个待开发功能提出问题, 如果团队有信心完成某一个功能, 就把这个功能从产品 Backlog 移到 Sprint Backlog。

产品负责人没有必要在 Sprint 计划会议上介绍产品 Backlog 的所有条目。根据产品 Backlog 中内容的多少和团队的速率(一个 Sprint 能够完成的工作量), 通常只需要介绍产品 Backlog 中高优先级的条目。较低优先级的条目可以放到以后的 Sprint 计划会议中去讨论。如果团队发现继续增加的工作量已经超过了整个团队一个 Sprint 能够完成的总量, Scrum 团队会立刻建议产品负责人停止。

团队和产品负责人一起确定整体的 Sprint 目标, Sprint 目标通常是关于下一个 Sprint 团队要完成工作的概述。在 Sprint 结束时的 Sprint 评审会议中, 团队会审视自己是否达到了 Sprint 目标, 而不是太关注于上个 Sprint 中完成的每一个具体条目。

在 Sprint 计划会议的下半段, 团队会讨论这些用户故事, 决定下一轮迭代能够完成的工作量。虽然从理论上说, 团队按照优先级次序, 从最高优先级选故事一直选到团队

认为足够为止。实际情况中，有时团队也会选择最高优先级的五个故事，然后又选了两个低优先级但与前五个故事有关的故事。一般情况下，团队会和产品负责人沟通，但是通常是由团队决定一个 Sprint 能够完成多少。

Scrum 的主要规则

- 在 Sprint 开始的时候召开 Sprint 计划会议。
 - 每个 Sprint 必须发布可以工作的、经过测试的代码，这些代码能够完成对最终客户有价值的一些功能。
 - 产品负责人为产品 Backlog 排列优先级。
 - 团队一起决定一轮迭代完成多少故事。
 - 在任何时候都可以向产品 Backlog 中添加故事，或者重新排列优先级。
 - 每天有一个 Scrum 短会。每个项目成员回答三个问题：我昨天做了什么？我今天打算要做什么？我有什么困难？
 - 只有团队成员能在每日 Scrum 简会中发言，其他人包括对项目感兴趣的观察者或利益相关者都不能发言。
 - 在 Sprint 结束时的 Sprint 评审会议中，团队会演示完成的成果。
 - 团队演示的是可以工作的软件，而不是幻灯片。
 - 准备 Sprint 评审会议的时间不得超过两小时。
-

一旦 Sprint 开始，只有团队成员才能向 Sprint 中添加工作。即使是 CEO 也不能让团队去做产品负责人没有提出的工作。销售人员也不能为了某一个特殊的客户而要求实现更多的功能。产品负责人也不可以改变主意，往该 Sprint 添加更多的功能。只有在团队觉得自己已经把 Sprint 承诺的所有任务都完成的前提下，才可以向 Sprint 中增加新的故事，此时，团队会询问产品负责人，要求再添加一两个用户故事。

在团队承诺完成 Sprint 计划会议中所选的全部工作的同时，组织也需要信守承诺不在 Sprint 期间改变 Sprint 的工作内容。如果有十分重要的事情发生，组织发现需要做出调整，就会把现在的 Sprint 取消，重新开 Sprint 计划会议，然后启动一个新的 Sprint。

团队从产品 Backlog 中选择故事之后，会从故事中划分出任务，形成 Sprint Backlog。在产品 Backlog 上的每个条目都可能会分解成 Sprint Backlog 上的一个或者多个条目。这样做有利于团队成员之间更好地共同分担任务。



Sprint 评审会议

每个 Sprint 都要发布一个“潜在可以交付的产品功能增量”。这就意味着在每一个 Sprint 结束时，团队都会发布完全实现了的、经过测试的并且可以使用的一部分功能。这部分软件应该是可以上市的，这说明组织可以根据实际情况，衡量软件升级的成本，以此来决定是否升级到新版本。尽管商业软件分销商可能考虑到升级的困难，而不愿意每个月都给客户升级。但 Scrum 团队还是必须每个月都发布可以上线的版本。

在每个 Sprint 结束时，都会有一个 Sprint 评审会议。在会议上，团队会演示在 Sprint 中完成的工作，经常是新功能的演示。

有意保持 Sprint 评审会议的非正式会议性质，尽量不用幻灯片，准备时间也不要超过两个小时。不要让团队感觉 Sprint 评审会议是一种干扰或者负担，它应该是 Sprint 自然而然的结果。

整个团队包括产品负责人和 ScrumMaster 都要参加 Sprint 评审会议。其他任何人(如管理层、客户和其他项目组的成员)如果感兴趣，也可以参加。

在 Sprint 评审会议中，大家一起评估是否达到了在 Sprint 计划会议上设定的 Sprint 目标。理想情况下，团队应该完成所有计划的工作，但是更重要的是，看团队是否达到了 Sprint 的整体目标。

每日 Scrum 简会

Scrum 是有记载的第一种包括每日短会的软件过程(Beedle 1999)，这个短会叫每日 Scrum 简会(Daily Scrum)。其他很多敏捷过程包括极限编程和特征驱动开发(FDD - Feature Driven Development)，很快都借鉴了这种做法。每日 Scrum 简会不需要花太多时间，不会过多地影响日常工作，但是我们可以很快地了解和分享项目信息。

等所有项目组人员到齐后，每日 Scrum 简会安排得越早越好。一般安排在 9:00 或者 9:30。所有项目人员(包括开发人员、测试人员、产品负责人和 ScrumMaster)都必须参加。会议必须简短，一般在 15 分钟内结束，最多不超过 30 分钟。为了保证会议时间不至于过长，很多团队要求参加者站着开会。

在每日 Scrum 简会中，每个团队成员要求回答以下三个问题。

1. 你昨天做了什么？
2. 你今天打算做什么？
3. 有什么困难？

注意，不要让每日 Scrum 简会演变为成员向 ScrumMaster 汇报工作的会议。这个会议的一个重要目的是让每个人在自己以及同事面前做出承诺。这个承诺不是向经理或者公司的承诺而是团队成员之间的承诺。

在每日 Scrum 简会中，尽量只回答这三个问题，不要让每日 Scrum 演变成为一个讨论系统设计或者解决问题的会议。一些问题会被记录下来稍后解决。有时候，团队的一部分成员需要留下来解决手头的问题，但注意，不要在每日短会中解决问题。例如，在每日 Scrum 简会中，有人会问什么时候开始使用我们供应商新发布的 5.0 版应用服务器。在这种情况下，大家决定在每日 Scrum 简会之后，我们需要安排一个有以下人员参加的会议。

- 技术架构师，讨论引入新的应用服务器对技术方面的影响。
- 产品负责人，根据从市场部门得到的信息，决定客户是部署应用服务器的新版本还是旧版本。
- 测试团队代表，可以从他们的角度评估对测试影响。

作为会议的组织者，ScrumMaster 需要保证会议的快节奏，确保所有人只回答这三个问题。产品负责人也需要向其他团队成员一样汇报进度，例如“我昨天完成了‘把书放入购物车’这个故事的测试，今天我会做一下市场调研确定我们接受哪些信用卡。今天结束时，我应该能够完成这个任务。”

每日例会的一个好处是它可以作为一个随机的检查点(checkpoint)，高层管理人员和其他人如果有兴趣了解项目状况，都可以参加。会议每天都会在相同的时间召开，所有有兴趣的人都可以参加，因此其他很多劳神的会议，如每月项目评审会议，就变得没有必要了。注意，如果是团队以外的人员参加会议，需要遵守一个规则，即在会议中只有项目组内部人员才能够提问。因此，大老板可以参加并留心倾听，但是她不能在会议上提任何问题，因为这样会干扰会议的顺利进行。

在每日 Scrum 简会中，团队成员每天都能够了解到项目的进展。因此团队也可以重新审视目前任务分配的情况。比如，Randy 报告说他现在的工作量比当初预想的大得多，而 Andrew 报告说他的进度已经大大超前了。在这种情况下，Andrew 可以当天就和 Randy

一起结对完成 Randy 的任务，Andrew 也可以去帮 Randy 完成一些其他的工作。

ScrumMaster 在每日 Scrum 简会中必须要把握好尺度。她需要保证会议的快节奏，同时也不要给大家一个感觉，好像这个会议仅仅是为她一个人开的。有一个问题尤其不应该问“完成‘订购一本书’这个故事还需要多长时间？”这个信息很重要，但是在每日 Scrum 简会中问这个问题很容易让大家开始讨论估算和数字。除了每日 Scrum 简会，我会让团队在共用的白板上更新他们的估算，当然如果大家没有坐在一起的话，就需要在软件中更新估算。

在 Scrum 中使用用户故事

介绍完 Scrum 的基本框架后，接下来我们看一下怎样在 Scrum 中应用用户故事。

Scrum 和产品 Backlog

在我的论文(Cohn 2003)中，我介绍了如何成功地使用用户故事描述 Scrum 的 Backlog 条目。在产品 Backlog 中的条目不再需要分为新功能、要调研的问题以及要解决的缺陷等，所有的条目都变成用户故事。产品 Backlog 中的每一个故事必须对客户或者产品负责人有价值。

如果限制产品 Backlog 中只有用户故事，排列优先级对产品负责人来说就很简单。她可以了解 Backlog 中的每个条目，从而很容易在不同的功能之间做出取舍。

与极限编程一样，在 Scrum 中，产品负责人也不需要一开始就确定所有的需求。但是在最初把尽可能多的故事记录下来还是有一定好处的。Scrum 不要求，甚至也不建议尽早维护很大的产品 Backlog。通常情况下，最初由产品负责人、ScrumMaster 和一个或者几个工程师一起完成最初需求的收集。我发现首先确定用户角色，然后根据角色来收集故事，在 Scrum 中应用十分有效。

在 Sprint 计划会议中使用用户故事

在 Sprint 计划会议中，产品负责人和团队一起讨论产品 Backlog 中那些高优先级的条目。团队确定在下一个 Sprint 中需要完成的条目，然后向产品负责人作出承诺。记下来，他们会把这些故事划分成小的任务，方便在 Sprint 中由程序员认领。

由于产品 Backlog 中的每一个条目都必须为用户故事，所以能确保每一个条目对客

户来说是有价值的。因而，我会觉得 Sprint 计划会议变得更容易，也更加有效，因为团队必须向产品负责人讲明白那些纯技术类的任务(如“重构 Login 类，让它抛出异常”)的好处。

在 Sprint 计划会议中使用用户故事很容易，就像我们在第 10 章那样，从用户故事中划分任务很容易。

在 Sprint 评审会议中使用用户故事

用户故事给 Sprint 评审会议带来的好处就是很容易评估 Sprint 中哪些部分已经完成。如果 Scrum 的产品 Backlog 使用各种各样的技术任务、需求、问题和缺陷修正，由于这样的 Sprint 任务很多并不可见，所以团队很难向产品负责人演示。使用用户故事则不会出现这样的问题。

在每日 Scrum 简会中使用用户故事

用户故事给每日 Scrum 简会带来的好处是，确保整个团队关注于完成余下面向客户和最终用户的任务。由于在 Sprint 之前没有需求或者分析阶段，所以在 Sprint 开始的时候，团队对要完成的任务只有一个大体概念。团队可能知道需要加一个搜索页面，不过他们可能不确定可以用哪些关键字进行搜索，是否支持检索条件组合，等等。用户故事的好处是作为一个提醒，团队始终明确要做什么，幕后的动机是什么。在 Sprint 中，团队不断与产品负责人交流，故事能帮助团队很方便地确定他们是否已经完成了某一个特定故事的足够功能，或者做得过多。

一个案例

接下来以我参加过的一个项目为例，介绍一下如何在 Scrum 中使用用户故事。为方便起见，我们姑且将该公司称为 Cosmodemonic，这是一家生物科技公司，规模不大的上市公司，主要开发生命科学行业方面的软件。这家公司刚刚用 9 个月的时间完成了一个人类遗传方面的新产品。就在公司刚刚发布最早的 beta 版网站时，它被另外一家公司收购了。

收购它们的公司对此新产品的用户很感兴趣，但是他们觉得应该重写这个软件，原因如下。

- 最初产品使用的客户端技术(HTML)与新公司的技术战略不匹配。
- 产品的目标市场从超大型的需要几千兆字节(TB, multi-terabyte)数据库的制药



公司，变成对数据库要求不高的实验室和小的生物科技公司。

- 第一版产品的设计实现比较糟糕。

最初这个产品是约一百个开发人员花 9 个月时间，利用十分严格的瀑布模型开发出来的。其实，一个 7 人以下的团队就可以完成所有的功能。

为了达到这个目标，我们使用了 Scrum 和用户故事。项目取得了巨大的成功。一个不到 7 人的 Scrum 团队花 12 个月完成了原来 100 多号人的瀑布团队花 9 个月完成的功能。整个团队，甚至连产品负责人和 ScrumMaster 在内也不到 7 个人。整个项目总共花费了 54 个人月，而瀑布版本花费了 540 个人月。

不包含注释，瀑布团队一共有 58 000 行 Java 代码，Scrum 团队用更少的代码完成了更多的功能，一共写了 51 000 行代码。这意味着瀑布团队平均生产率是每人月 120 行 Java 代码，而 Scrum 团队的是 840 行每人月。表 15.2 对两种方式进行了对比。

表 15.2 用两种不同方式完成相同一个项目的比较

	瀑 布	Scrum 和用户故事
用例页数	3 000	0
故事数	0	1 400
日历月	9	12
人月	540	54
Java 代码行数	58 000	51 000
Java 代码行数 / 人-月	120	840

小结

- Scrum 是一种迭代和递增的过程。
- Scrum 每 30 天一轮迭代，称为 Sprint。
- ScrumMaster 相当于传统的项目经理，但更像是领导者和组织者，而不是经理。
- 一般的 Scrum 团队包括 4~7 个开发人员。
- 产品 Backlog 是一个待开发的功能需求列表，里面的条目要么还没有实现到产品中，要么还没有计划在当前 Sprint 中开发。
- Sprint Backlog 是一个团队承诺在当前 Sprint 完成的任务列表。
- 在极限编程里面的客户角色，在 Scrum 中称为产品负责人。

- 产品负责人负责给产品 Backlog 排列优先级。
- 在 Sprint 的开始，团队从产品 Backlog 中选择下一个 Sprint 要完成的任务。
- 在每日 Scrum 简会中，每个团队成员需要回答三个问题：我昨天完成了什么？今天我要做什么？我碰到了哪些问题？
- 每个 Sprint 都要完成一部分可以潜在交付的产品功能增量。
- 在 Sprint 结束时，团队在 Sprint 评审会议上演示所完成的功能。

问题

- 15.1 描述递增过程与迭代过程的区别。
- 15.2 产品 Backlog 和 Sprint Backlog 有何关联？
- 15.3 什么是潜在可交付的产品功能增量？
- 15.4 谁负责排优先级？谁负责替团队选择下一个 Sprint 的任务？
- 15.6 在每日 Scrum 简会中，每个成员都要回答哪些问题？





第16章 其他话题

通过本书这部分内容的介绍，我们已经阐述过一些用户故事的常见主题。我们已经讨论过用户故事与其他需求管理方法的区别，讨论了在哪些情况下更适合使用用户故事。我们也讨论过用户故事的一些常见不良征兆或问题，讨论了如何纠正它们。在本章中，我们一起探讨一些其他主题。

- 处理非功能性需求。
- 团队应该使用纸质笔记卡还是软件工具。
- 用户故事在用户界面方面的影响。
- 在开发完成后，是否应该保留用户故事。
- 缺陷报表和故事之间的关系。

处理非功能性需求

团队开始使用用户故事时，最常见的障碍是，他们会觉得所有的东西都必须转化为用户故事。大多数项目一般都会有一部分需求无法恰当地以故事形式来表达。这些往往是系统的非功能性需求。

非功能性需求可以表达各种系统需要。常见的非功能性需求类型如下：

- 性能(performance)
- 准确性(accuracy)
- 可移植性(portability)
- 可重用性(reusability)
- 可维护性(maintainability)
- 互操作性(interoperability)
- 可用性(availability)
- 易用性(usability)
- 安全性(security)
- 容量(capacity)



许多非功能性需求可以视为系统行为的约束。例如，项目中包含诸如“系统须用 Java 语言编写”的需求并不少见。这无疑是系统剩余部分设计的约束。正如第 7 章所讨论的那样，处理约束最好的办法是在卡片上写下约束，并将卡片标注为“约束”卡。大多数情况下，编写自动化测试(并且至少每天运行一次)可以确保系统遵守约束。有一些约束无法进行实际测试或者不值得测试。约束“系统须用 Java 语言编写”就是这样。当然，我们有确保满足这个约束的更简单的方法。

表 16.1 展示了一些常见的约束范例。除了在卡片上写下约束，如果系统确定要有更多非功能性需求，可以用任何合适的或传统的形式来沟通。例如，如果采用数据字典来表示系统所有变量的大小和类型有不错效果，就创建一个数据字典。

表 16.1 常见非功能性需求的示例约束

方 面	示例约束
性能	80%的数据库查询结果应该在两秒钟之内显示到屏幕上
准确性	软件能以至少 55%的准确率预测出足球赛的赢家
可移植性	系统不应该使用任何会使得移植到 Linux 系统变得困难的技术
可重用性	数据库以及数据库访问代码应该可以在日后的应用程序中重用
可维护性	所有组件都必须有自动化的单元测试 自动化单元测试至少每 24 小时完整运行一遍
互操作性	系统须用 Java 语言编写 所有配置数据应以 XML 文件存储 数据应存储在 MySQL 数据库中
容量	数据库在特定的硬件环境上，应该能够存储 2000 万个会员，同时还能满足性能目标

纸质还是软件？

故事应该写在纸质卡片上还是存储在软件系统中？这个问题甚至比在百货店里问“纸包装还是塑料包装？”还普遍。许多极限编程社区的人主张使用纸质笔记卡，因为它们简单。极限编程重视简单的解决方案，纸质笔记卡绝对简单。此外，卡片可以鼓励交流和讨论。在做计划时，它们可以以各种形式放在桌面上，可以堆放它们，也可以对它们分类，任何会议都可以带入它们，等等。

另一方面，有专门为追踪故事而设计的软件产品(VersionOne^①、XPlanner^②和 Select Scope Manager^③)，也有通用的软件可用于故事的管理(电子表格、缺陷追踪软件以及维基)。

卡片相对于软件工具的主要优点之一，是它们技术含量低的本质，可以不断提醒人们故事是不精确的。在软件中使用时，故事可能就要采用 IEEE 830 样式需求的外观，由此，那些故事可能会增加额外的、不必要的细节。

典型的笔记卡只能容纳有限数量的文字。这可以给故事描述文本一个很自然的上限。而大多数软件则没有这种限制。另一方面，使用笔记卡的一种常见做法是，在卡片背面写一些验收测试的样例。许多情况下，卡片的尺寸不利于写测试用例。

ClickTactics 选择使用软件

ClickTactics 是一家营销解决方案提供商，他们专门编写网络访问软件的组件。他们起初使用笔记卡，而后转向软件，即 VersionOne 的 V1:XP。

Mark Mosholder 是 ClickTactics 的资深产品经理，他说转变的一个原因是他们的销售队伍和高层管理人员分布在多个地点。他们无法对远程利益相关者说“去看看白板就知道了”，因此他们花了很多时间与高层管理者和其他远程利益相关者同步信息。同时，他们用笔记卡时，偶尔会有卡片丢失的情况，几个星期以后却在桌子底下成堆的东西中找到了。

在 VersionOne 软件中使用故事，让 ClickTactics 得以把极限编程的使用作为一个销售工具。使用软件，可以给客户看到有限的迭代信息。然后他们就可以推广他们的交付能力，通过告诉客户“我们可以让你在三周后获得新的功能”，迅速地交付给客户新的版本。

Mark 说他们决定使用软件来管理故事没有什么障碍，并表示以后还会使用软件。

推行 ISO(International Organization for Standardization, 国际标准化组织)或类似认证的项目，需要从需求声明到代码以及测试的可追溯性，因此项目组很可能会喜欢使用软

① 可查看 www.versionone.net。

② 可查看 www.xplanner.org。

③ 可查看 www.selectbs.com/products/products/select_scope_manager.htm。



件。使用手写的笔记卡应该可以达到 ISO 认证标准，但把一堆卡片放到合适的位置，又要保证足够的变更控制，步骤过于繁琐，因此相比卡片的其他优势就变得无足轻重了。

同样，不在同一地点工作的团队可能会更喜欢软件而不是笔记卡。当一个或多个开发人员，或者尤其是客户，在远程时，更不可能使用纸质卡片。

笔记卡的另外一个优势在于，很容易用来排序，而且可以用多种方式进行排列。故事集合可以排列到高优先级、中等优先级和低优先级这几堆中。或者也可以用更加精确的顺序排列故事，第一个故事的优先级高于第二个故事的优先级，并且第二个故事的优先级高于第三个故事的优先级，以此类推。

卡片的拥趸很多，但软件工具的支持者也不少。但我的看法是，这两种方法都适合。我建议从卡片开始，看看是否适用于具体环境。然后，如果有令人信服的理由使用软件，放心切换好了。

Diaspar 软件服务公司使用 Wiki

Diaspar 软件服务公司是一家软件开发及顾问公司，J. B. Rainsberger 是该公司的创始人。作为一名顾问，J. B. 无法总是跟客户在一起。在类似情况下，J. B. 使用 Wiki 加强自己与远程客户之间的交流。Wiki 是特殊网页的集合，任何查看页面的人都可以编辑它。J. B. 和 Diaspar 软件使用 FitNesse 作为他们的 Wiki。他们不会为每个故事编写一张故事卡，而是为每个故事创建一个新的页面。

J. B. 报告说，在最近的一个小项目中，这种方法工作得非常好。一旦对一个故事有疑问，他就会在页面上记录下问题，并加上文本“待办事项”(todo)。每周，他的客户会检查几次 Wiki，搜索“待办事项”，并回答问题。紧急的问题通过电话解决，但因为使用 Wiki 比较高效，所以很少有紧急的问题。在其他项目里，J. B. 也使用笔记卡，但他说在这个项目中，由于远程客户的关系，虽然他希望同时在卡片和 Wiki 上记录故事，但从来就没有时间那么做。

尽管 J. B. 使用 FitNesse wiki 为每个项目编写可执行的测试，他指出“假如每个人都在一个屋子里，就没有必要把故事放到 Wiki 上。”

用户故事和用户界面

很多人认为，敏捷方法在很大程度上会忽略用户界面的设计问题。从某种程度上说，

这是可以理解的：敏捷过程是高度迭代的，而传统的用户界面设计方法严重依赖于前期的设计。对于一个具有重大或重要用户界面的应用程序而言，理解使用基于故事的敏捷方法有潜在的风险是很重要的。

我们可以反复地完善系统，这是敏捷开发的原则之一。用户故事允许我们延迟讨论，直到开发人员准备好实现故事。有时候，延迟讨论会导致开发人员对现有应用程序的一部分进行轻微的返工，但我的看法是，这些轻微返工的成本是非常合乎情理的，因为这不仅可以节省时间，不讨论那些未来可能会被抛弃的功能的需求，同时还使得客户可以通过许多小的阶段性修正来控制产品的发展方向。

假如这些变化发生在应用程序的用户界面背后，那么这种想法可能是正确的。但是，当这些变化影响到用户界面时会怎样？Larry Constantine(2002)如此描述：

对于用户界面，架构的设计(包括总的结构组织、导航以及观感)，必须可以覆盖各种各样的任务。当架构涉及到用户界面，那么后期改进是无法接受的，因为这意味着对那些已经学会或掌握了先前界面的用户更改系统。即使是对布局或者窗口特征做轻微的调整，都可能产生问题。

Constantine 和 Lockwood(2002)提出了解决方案，即敏捷版本的以使用为中心的设计(usage-centered design)。敏捷版的以使用为中心的设计由基本用例或任务案例(task case)驱动，而不是用户故事驱动。然而，我们可以用故事替换基本用例，这就产生了以下基于故事的敏捷版以使用为中心的设计：

- 用户角色建模
- 捕捞高层次的用户故事
- 排列故事优先级
- 精炼高优先级和中等优先级的故事
- 对故事整理分组
- 建立书面的原型
- 精炼该原型
- 开始编程

第一步是举行用户角色建模会议，正如第 3 章描述的那样。为了完成后续的步骤，启动第 4 章所说的故事编写工作坊。在工作坊里重点捕获最高层次的故事，很可能不会超过两打。

接着，把高层次的故事排列成三组：高优先级故事肯定会包含在最近一个发布中，期望中等优先级故事有望放入最近一个发布，而低优先级故事可以延迟到后续发布中。把高优先级和中等优先级的故事精炼成更小的故事时，把低优先级的故事放在一边。这些故事的大小是适合做发布计划的。

然后，把高优先级和中等优先级的故事整理为有望一起执行的组。随后，给每个组的故事在纸上画出原型。在创建好书面原型后，展示给用户(或者有必要的话展示给用户代理)看，并根据他们的意见改进原型。

如果在项目中加入这些步骤，记得尽量保持这个过程的轻量性。因为还有一些已识别好并且已经画出用户界面原型的故事，最终在开发前会被删除。避免付出任何没有必要的时间。对于大多数应用程序，少则可能需要几天，多至几周(对于有远程用户的商业软件)。

编写两套

几年前，我参与了一个项目，我们请来 Ward Cunningham(译者注：此人是 Wiki 概念的发明者)来咨询评估那个项目。那时团队在关于用户界面的问题上陷入困境。有许多激烈的辩论是关于用户会喜欢基于浏览器界面还是他们会更喜欢本地窗口程序。我们的营销小组已经询问过用户，但我们不确信他们做的调查是否充分，或者他们做调研的方式是否正确。

Ward 解决这个问题的方法是告诉我们“写两份用户界面”。他的逻辑是，两种界面都不难编写，而且在这两者之间，他们可以真正地保证应用程序的中间层是独立的。由于有两份用户界面，功能代码不会被不合时宜地放到客户端，如果那样，同样的代码就要写两次。

Ward 的建议是正确的。当然，我们没有听取他的建议，认为开发两份完整用户界面的成本太高。产品完成后，客户让我们知道我们确实选择了错误的用户界面技术。然后迅速启动另一个项目，为那个产品增加第二份用户界面。

保留故事

关于是否应该保留故事的争论很普遍。正方认为，欣然撕掉已完成卡片的好处大于保留卡片的任何价值。反方是比较保守的群体，他们宁愿保存故事卡而不要冒扔掉之后

又需要的风险。

如果使用软件来保存故事，就没有理由丢弃已经完成的故事。可以从删除一个电子故事获取一些快乐和成就感，但这很可能不如把纸质卡片撕成两半来得痛快。

若使用的是纸质笔记卡，的确可以从完成一张卡片以及撕成两半中获取真实的愉悦。我使用卡片来引导编写本书，每次完成或修订好一个新的小节，我就撕掉卡片。但是，当我工作的对象是软件项目而不是书时，我更喜欢保留卡片，用橡皮筋归档到书架上。

多年来，我确实遇到过少数几次得感谢保留了需求的情况。这里有一些案例。

- 我就职的公司被收购。收购方对我们轻量级的开发过程很有兴趣，但他们有自己重量级的面向瀑布的方法，每个项目都要通过众多的门槛和签署点。因为我能实际地给他们展示我们的开发过程(从故事到代码和测试)，他们允许我们沿用自己的过程，而不采用全公司的标准。更妙的是，我们最终能够取得一些进展，在公司其他部门传播我们的过程。
- 我有几次参与过商业产品完全重写的情况。在一个案例中，某个产品的第一版因为一些劣质技术的选择失败了。后来该产品被完全重写，取得了一定的成功。另外有个客户端-服务器应用程序产品，取得了惊人的成功。5年之后，公司想重写这个产品，并发布到网上。即便存有任何过期的故事或者需求，有时也会派上用场。
- 我涉及过的另外一种情况是，一家刚刚创建的小公司企图与一家大公司完成交易。完成交易后公司会盈利，老板也高度承诺给所有开发人员发放五位数的奖金。对方要求我们“提供一份需求副本”。我开始走上一条不归路：对他们描述我们实际上为何不关注编写需求，而关注谈话和合作。我可以感觉到这种对话不会有好的结果，公司的利润和开发人员高额的奖金都会蒸发。我转变说法，告诉他们我们如何以用户故事的形式编写需求。他们喜欢这个主意。幸运的是，那个项目的故事存储在电子系统里。我们把它们从那个系统里剪切出来，粘贴到 Word 文档里，增加了一个封面和签名页，其结果是皆大欢喜。

考虑到在不同场合保留故事的好处，我的建议是你也这样做。如果使用的是软件，要么始终保证安装了软件，要么从软件里打印出一份报表，并把报表归档在某处。假如在使用卡片，要么保存卡片本身，或者一次复印 3 个卡片到纸上。

缺陷的用户故事

一个非常普遍的问题是故事与缺陷报告之间的关系。我发现最好的方法是把每个缺陷报告当成自己的故事。如果修正缺陷很可能会花费完成典型故事所需的时间，就可以像任何其他故事那样对待那个缺陷。但是，对于那些团队期望能够快速修复的缺陷，应该把那些缺陷合并到一个或多个故事中。这一点很容易用故事卡来实现，可以把故事卡装订在一起，加上封面故事卡。然后，出于做计划的目的，缺陷集合可以当成一个单一的故事。

使用彩色怎么样？

有些团队喜欢使用不同颜色的卡片来表示不同类型的故事。例如，传统的白色卡片可以用来表示普通的故事。红色的卡片可以用于缺陷，蓝色的卡片可以用来表示工作任务，等等。

对我来说，使用彩色的卡片在理论上来说似乎总是一个好点子；但实际上这实在是画蛇添足。我无法仅仅在口袋里放白色的卡片，我必须确保始终携带一些每种颜色的卡片。如果我用完了某种颜色的卡片，我就得在其他颜色的卡片上写下故事，这之后还得重写故事。但是，假如觉得彩色卡片可以帮助你整理故事，尝试一下也不错。但我会坚持简单，即一大堆白色卡片。

小结

- 非功能性需求(诸如性能、准确性、可移植性、可重用性、可维护性、互操作性和容量等)都可以通过创建约束卡的方式来处理。假如系统的需求比这些更复杂，但无论其他格式还是方法，只要它能充分表示那些需求，就可以用作用户故事的补充。
- 笔记卡和软件系统都不是适用于所有情况下编写故事的最佳方式。使用适合项目和团队的工具。
- 迭代过程会导致用户界面的反复变动。习惯于特定界面的用户，不会喜欢用户界面变动，因为这会影响他们已经学会的操作软件的方法。可以考虑加入一些敏捷版以使用为中心的设计实践，以避免用户界面的反复变化。

- 一旦故事完成，撕毁故事卡会有一定的乐趣。但同样也有保留卡片的理由。宁可谨慎一些，保留故事。
- 把小的缺陷报告用一个封面故事卡装订在一起，并把它们当成一个单一故事。

开发人员职责

- 在适当的时候，建议并使用替代技术和方法来表示需求。
- 共同决定适用于项目的方式：笔记卡还是软件系统。
- 共同理解在项目开始时考虑所有用户界面的优点和缺点。

客户职责

- 如果觉得用户故事无法准确地反映需求的一部分，应该要求使用替代技术和方法来表示需求。
- 共同决定适用于项目的方式：笔记卡还是软件系统。
- 共同理解在项目开始时考虑所有用户界面的优点和缺点。

问题

- 16.1 如何处理让系统支持 1000 个并发用户的扩充性需求？
- 16.2 你喜欢在笔记卡还是在软件系统里编写故事？请解释一下。
- 16.3 迭代过程对应用程序的用户界面有何影响？
- 16.4 给出一些例子，说明在系统前期考虑用户界面比典型敏捷项目的做法具有更多好处。
- 16.5 一旦开发完故事，你建议保留故事还是销毁故事？请解释一下。



第 IV 部分 一个完整的实例

第 IV 部分通过一个例子综合呈现前面论述过的所有知识和技巧。在后面各章中，我们会看到南海岸航海用品(South Coast Nautical Supplies)的案例，它的一些员工创建了一个网站以扩充它们现有的目录销售业务。你将有机会和 Lori(南海岸的销售和市场副总裁以及这个项目的客户)一起定义用户角色，编写故事，估算故事，创建发布计划，最后为初始计划中的故事编写一些验收测试。



第 17 章 用户角色

在接下来的 5 章中，我们将开始一个假设的小项目。在本章中，我们将从定义用户角色开始。在后续各章中，我们将编写用户故事，估算故事，做发布计划，为故事编写验收测试。

项目

我们公司南海岸航海用品(South Coast Nautical Supplies)，30 年以来一直用产品目录销售航海用品。我们的产品主要包括全球定位系统、钟、气象设备、导航及绘图设备、救生筏、救生衣、图表、地图和书。到目前为止，我们的网站仅仅只有一个简单的网页，告诉人们拨打免费电话索要产品目录。

我们老板决定我们应该与时俱进，开始在网上卖东西。但是，他不希望我们一开始就卖大件物品，而是先从卖书开始。因为有些物品价值超过 10000 美元，所以除非我们认为我们的网站一切正常并且不会丢失订单，否则我们不愿意在这些贵重物品上冒任何风险。但是如果我们发现客户喜欢在网上下订单，而且网站做的不错，我们也会考虑在网站上卖其他产品。

噢，老板最后还说了网站要在 30 天内上线，这样我们就有望在夏天航海旺季提高销量。

定义客户

项目需要一个客户来帮助我们定义和编写故事。这个项目的客户是需要买书的航海者，他们都不是公司员工。所以，我们需要一个内部客户作为最终客户的代理。老板指定了 Lori 来充当这个客户，她是销售和市场副总裁。

在第一次会议中，Lori 提供了一些系统的背景信息。她需要一个“典型的书店/电子商务网站”。她希望客户能通过各种方法(此时我们还没有要求她澄清具体内容)来搜索书籍，她希望用户能维护一些书籍列表，这样他们就能记着以后再买，她希望用户能对

他们买的书进行打分和评论，她还希望用户可以查看订单状态。我们已经见过许多这样的网站，所以我们告诉 Lori 我们随时可以开工。

定义一些角色雏形

首先，我们把一些开发人员和 Lori 聚集到一个有大桌子的房间里。Lori 已经做过市场调查，了解典型客户的特征。Lori 和开发人员一起写下如下用户角色卡片，如图 17.1 所示摆放：

- 狂热(铁杆，或称骨灰级)航海者
- 初级航海者
- 新航海者
- 礼物购买者
- 航海者的不出海配偶
- 管理员
- 销售副总裁
- 船长
- 有经验的航海者
- 航海学校
- 图书馆
- 教练

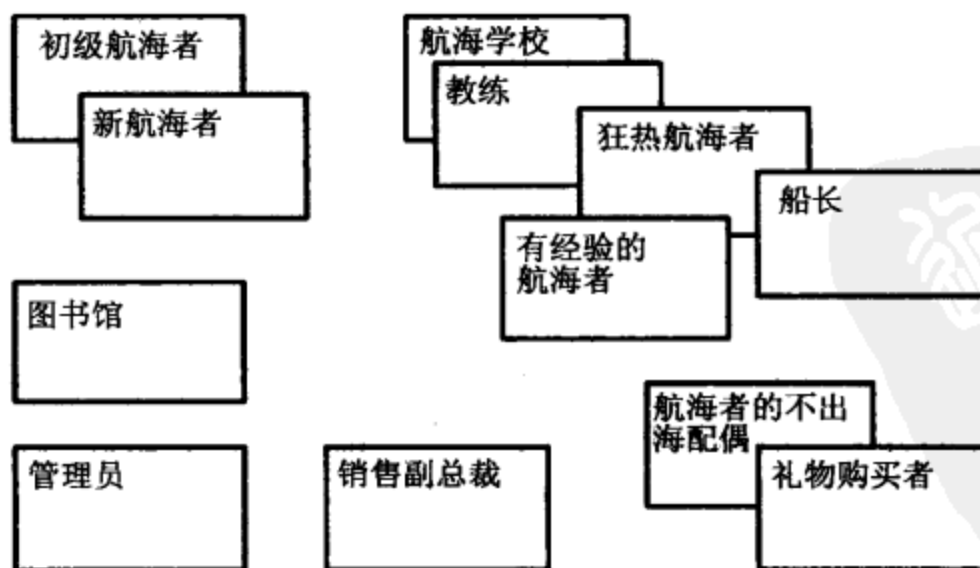


图 17.1 用户角色卡片的摆放位置

整合与提炼

把用户角色名字写在卡片上后，我们需要去掉重复的或相似的，看看有哪些角色应该合并，从而得到一个精简的用户角色列表，我们可以用这个列表开始我们的项目。最简单的方法是去掉那些完全覆盖其他卡片的卡片，表明写这个卡片的人认为卡片重复了。

在这个项目中，“新航海者”卡片放在“初级航海者”卡片上面。写这些卡片的人解释这些卡片的意图，任何人都可以补充自己的想法。然后我们发现了“新航海者”和“初级航海者”的区别。“新航海者”是刚开始航海的人，可能她正参加航海课程或者才出海几次。写“初级航海者”卡片的人认为这个角色可能已经航海几年了，但是并不经常出海，所以还不熟。大家讨论后决定虽然这两个角色有一些区别，但是区别不大，因此没有必要为此定义两个角色。它们被合并成一个角色“初级航海者”，从而扔掉“新航海者”卡。

接下来，大家开始考虑“航海学校”和“教练”的重叠部分。写“教练”卡的人解释说这个角色代表了教授航海课程的航海者。她认为教练经常给她们的学生买书，或者可能整理一个学生需要阅读的书籍列表。写“航海学校”角色卡的人指出这也是她写这个角色卡的部分原因。但是，她认为一般是学校的管理人员来做这些事，而不是航海教师自己来做。客户 Lori 为此向我们解释：即使是学校管理人员，她也和教师有很多相同的特征。因为教师比航海学校更明显是指一个个体，因此“航海学校”卡片被撕掉。

“狂热航海者”和“教练”、“有经验的航海者”、甚至“船长”角色卡片部分重叠。接下来大家讨论这些角色，发现“狂热航海者”角色代表那些知道确切需要哪本书的航海者。比如，“狂热航海者”知道导航方面最好的书的书名。她的搜索模式显然不同于那些知之甚少的航海者甚至是“有经验的航海者”。“有经验的航海者”代表那些很熟悉网站产品的人，但是她们想不起最好的书的书名。

经过讨论后，团队决定撕掉“船长”角色卡，因为这个角色本质上和“狂热航海者”是一样的。

此时，团队已经决定保留“初级航海者”、“教员”、“狂热航海者”和“有经验的航海者”。舍弃了“新航海者”、“船长”和“航海学校”。他们仍然需要考虑“礼物购买者”、“不出海的配偶”、“管理人员”和“销售副总裁”。写这些卡片的人阐述了他们的想法。



“礼物购买者”角色代表那些本身不是航海者但是要给航海者买礼物的人。写“航海者的不出海配偶”角色的人指出这也是写这张卡的想法。经过讨论过后，团队决定把两个角色卡片都撕掉，用“不出海的礼物购买者”来代替他们。

写“管理员”角色的人解释说这个角色负责向系统导入数据，保持系统持续运行。这个角色是团队讨论的第一个不在网站上买东西的角色。讨论过后，他们决定，这是一个重要的角色，Lori 说她会加一些故事，包括如何维护系统和如何向系统添加新的商品。

接下来讨论“销售副总裁”这个角色。这是另一个不购买的角色。但是，CEO 明确指示要仔细观察新系统，看它对销售的影响。团队认为不用关心这个角色，因为他们认为不会有很多专门为这个角色的故事。最后他们决定保留这个角色，但是改为更通用的名字“报表查阅者”。

接着是“图书馆”角色。团队认为这个和航海学校，甚至“航海者的不出海配偶”是类似的。然而，大家驳回了这些想法，决定保留“图书馆”角色。但是，根据角色代表实际用户的准则，“图书馆”卡片被替换为“图书馆管理员”卡片。

至此，团队确定了如图 17.2 所示的角色。

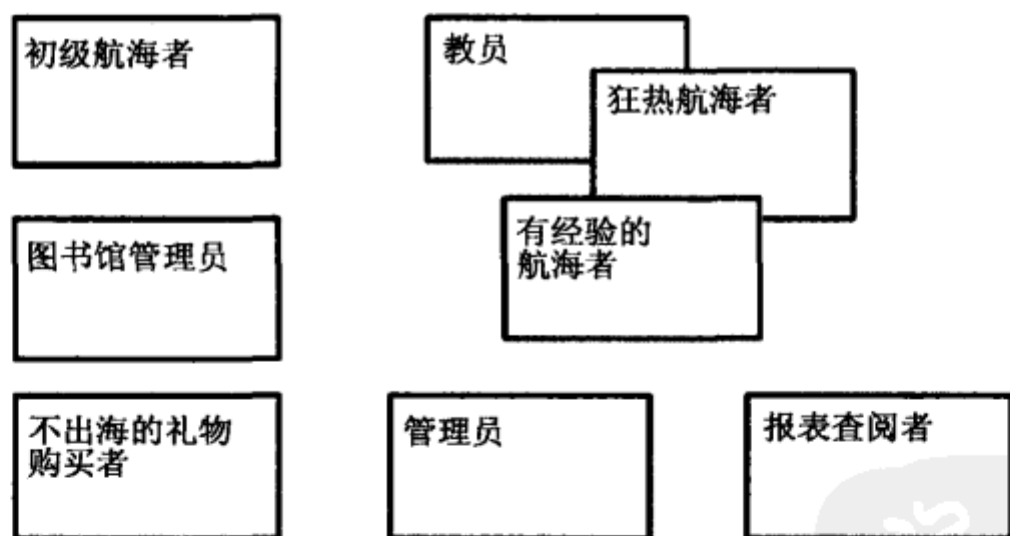


图 17.2 初步讨论整合后的角色

角色建模

接下来，团队考虑每一个角色，在角色卡片上添加一些详细描述。描述因不同的领域和软件类型而多种多样，需要考虑以下通用因素。

- 用户使用软件的频率。
- 用户在这个领域的专业能力程度。
- 用户对电脑和软件的熟练程度。
- 用户对团队正在开发的软件的熟练程度。
- 用户使用软件的目的。一些用户关心易用性，还有一些用户更在乎用户体验，等等。

大家针对每个角色卡片讨论这些问题。他们更新了用户角色卡片，如下所示。

- **初级航海者**：有网络购物的经验。在航海的前 3 个月进行过 6 次采购。有时指定某个书名；有时在选择合适的书籍时需要帮助。希望在选择合适的书籍上能比在实体书店里得到更多的帮助，从而找到程度合适的好书。
- **教练**：经常使用网站，一般一个星期一次。教员经常通过公司的电话销售部门下类似的订单(例如，20 本同样的书)。熟练使用网站，但是对使用电脑有些焦虑。希望能拿到最好的价格。对评论或其他“华而不实的东西”不感兴趣。
- **狂热航海者**：基本不熟悉电脑。经常购买许多公司产品目录中的产品，但是不怎么买书。从我们这里买很多设备。一般知道自己的需求。不希望在使用网站的时候让人觉得自己笨手笨脚。
- **有经验的航海者**：能熟练使用电脑。每季度会买一到两次，可能在夏天买的较多。了解航海，但通常仅限于本地区域。对其他航海者认为的好产品和航海的好去处十分感兴趣。
- **不出海的礼物购买者**：通常能熟练使用电脑(否则不会喜欢在网上买礼物)。不是航海者，最多知道一些航海术语。通常寻找指定的书，但也有可能寻找特定主题方面的书。
- **图书馆管理员**：能熟练使用电脑。知道自己要什么，喜欢用 ISBN 找书，而不是根据作者或书名。对礼物包装或递送跟踪之类的不太感兴趣。通常一年会购买几次，但是每个订单都比一般个人订单大。
- **管理员**：十分精通电脑。至少比较熟悉航海。每天访问系统后台是其工作内容之一。希望软件容易上手，但是之后需要一些高级用户的快捷方式。
- **报表查阅者**：比较精通电脑，尤其是商业程序(如电子表格和文档处理之类)。希望掌握一些具体数据，包括系统运行情况，用户买了什么或没买什么，用户怎么浏览或搜索网站。与速度相比，他更在乎报表的功能强大和深度这些商业价值。



添加虚构人物

有时花几分钟添加一个虚构人物是值得的。团队问 Lori 对于网站成功，这些角色中哪些用户的需求最需要得到满足。她说“狂热航海者”很重要，因为他们是长期客户。但是，即使他们频繁出海，他们也不会购买大量的书。另一方面，人数众多的“有经验的航海者”同样重要，他们会买大量的书。Lori 还补充说“教练”可能是最重要的角色。

“教练”每年可能要买上百本书。事实上，她想研究一些方法，为那些介绍学生到网站的“教练”提供一些金钱奖励。

有了这些信息，团队决定虚构两个人物。第一个虚构人物是 Teresa。Teresa 已经航海 4 年了。她是一个上市生物技术公司的 CEO，非常喜欢在网上购物。Teresa 主要在夏天出海，所以她只在春天或者夏天准备出海时使用网站。她非常繁忙，希望能通过我们的网站节省时间，找到她以前没看过的书。Teresa 嫁给了 Tom，Tom 自己从不出海，但是曾两次陪伴 Teresa 到地中海航行。

第二个虚拟人物是 Ron 船长。Ron 船长已经航海 40 年了，在圣地亚哥外经营着一个航海学校。他五年前从高校的教育岗位退休，从那时起就是一名航海教练。他是 10 年的忠实客户。他依然对自己办公室的电脑感到生疏，但是他对网上购物非常好奇，我们希望他能试试。

关于 Teresa 和 Ron 船长的补充说明

是否值得在这个系统中添加一些虚构人物还值得商榷。只有当团队认为有这样一个必须满足其要求的客户更容易让大家分析出用户故事时，才需要添加虚构人物。这几章里描述的南海岸航海用品系统功能比较少，因此不值得再设置更多的虚构人物。

尽管如此，作为一个十分有价值的工具，我还是加上了 Teresa 和 Ron 船长两个虚构人物，用以提供更完整的场景。



第 18 章 一些用户故事

为了生成初始的故事清单，团队决定召开一次故事编写工作坊。通过一两小时的工作坊，大家一起写出尽量多的故事。工作坊里，一种方式是不按照角色或者虚构人物的顺序写出故事。另一种方式是先从一个特定的用户角色或者虚构人物开始写出团队能想到的所有故事，然后考虑下一个角色或虚构人物，周而复始。两种方式的结果应该是一致的。在本例中，通过讨论，团队决定采用第二种方式。

Teresa 的故事

团队决定从第 17 章识别出的虚构人物 Teresa 入手，因为团队的客户成员 Lori 指出新的网站能否满足 Teresa 的需求非常关键。团队认识到 Teresa 十分在意速度和便捷性。作为一个高级用户(power user)，只要能帮助她更快地找到她所寻找的，她并不太介意一点点额外的复杂性。

故事卡 18.1

用户可以用作者、书名或 ISBN 搜索书籍。

开发人员对于这个故事有一些问题。例如，用户可以同时按照作者、书名及 ISBN 搜索，还是 Lori 想让他们一次搜索只能选择一个条件？但团队先把这类问题放在一边，以能够集中精力继续产生更多的初始故事。

接着，Lori 指出一个用户搜索到一本书后能看到该书的详细信息。针对她对这些信息的想法，她举了几个例子说明并且写下故事卡 18.2。

故事卡 18.2

用户可以查看书籍的具体信息。例如：页数、出版日期和内容简介。

可以预见，除了这三个信息之外，可能还有 Lori 想要的其他信息，但是开发人员可以在日后开始着手编码实现这个故事的时候再去询问她。

作为一个典型的电子商务网站，团队知道网站用户需要一个“购物车”并购买其中的书。作为客户，Lori 指出一个用户在结账前应该能够从“购物车”里删除任何书。如故事卡 18.3 和 18.4 所示。

故事卡 18.3

用户可以把书籍放进“购物车”，在结束购物时，可以购买其中的书。

故事卡 18.4

在完成订单前，用户可以从她的购物车中删除书籍。

为了能实际处理信用卡付账，系统需要知道信用卡和地址信息。由此便有了故事卡 18.5。

故事卡 18.5

购买书时，用户需要输入她的账单地址、送货地址及信用卡信息。

Lori 提醒开发人员，因为 Teresa 只有 4 年航海经验，所以她并不一定很了解自己需要什么书。为了帮助 Teresa 更容易找到想要的书，网站应该包括客户评级和发表评论的功能。由此，Lori 写下故事卡 18.6。

故事卡 18.6

用户可以对书籍进行评级和发表评论。

由于 Teresa 期望能尽可能便捷地下定单，所以团队认为系统需要保存送货地址和账单信息。有一些网站的客户，例如“不出海的礼物购买者”，可能因为不会经常订购，并不想创建能重复使用的用户信息。同样的，像 Ron 船长这样对使用新网站犹豫不决的人，在第一次使用网站时任何额外的步骤可能都会让他产生抵触。因此，Lori 决定，无论是否是注册用户，都能购买书籍。为此，她写下故事卡 18.7 和 18.8。

故事卡 18.7

用户可以创建一个网站账户用于保存送货地址和账单信息。

故事卡 18.8

用户可以编辑她的账户信息(信用卡、送货地址和账单地址等)。

团队觉得 Teresa 会有一些想要的书但现在暂时不下定单，她会想把它们放到愿望清单(wish list)中。她可以在日后下定单或者告诉她的丈夫 Tom，让他从清单中购买。因此，Lori 又写下故事卡 18.9 和 18.10。

故事卡 18.9

用户可以把书放入“愿望清单”，网站的其他访问者也可以看到该清单。

故事卡 18.10

用户可以把愿望清单(甚至是别人的愿望清单)中的项目放入自己的购物车。

我们需要确保故事卡 18.10 的开发者(不管是谁)明确知道用户可以从自己的或者他人的愿望清单中选择书。我们确保故事卡上有这样的注释(如卡片上括号所示)。

由于 Teresa 非常看重速度，所以 Lori 定义出一个与订购一本书花费时间有关的性能限制条件。她写下故事卡 18.11。

故事卡 18.11

老顾客必须能够在 90 秒内找到书和下订单。

(约束)

这个例子中，Lori 选择关注一个老顾客搜索书籍和完成购买所要花的时间长短。因为这个故事覆盖了网站用户体验的方方面面，所以是一个不错的性能需求。只是优化数据库查询和中间件的意义不大，因为令人迷惑的用户界面可能已经使用户要花 3 分钟才能到达搜索屏幕。这个故事比“搜索必须在 2 秒内完成”更好地反映了这个想法。当然，Lori 可以添加更多的性能限制条件，但通常选择几个像这个故事一样覆盖面较广的就足够了。



Ron 船长的故事

团队觉得目前写不出更多关于有经验的航海者 Teresa 的故事了，于是团队决定将视线转移到 Ron 船长。Ron 经营着一所航海学校。他的电脑操作经验比 Teresa 稍微生疏一些。Ron 船长访问网站时通常已经明确自己要找什么。这样一来，Lori 写出故事卡 18.12 和 18.13。

故事卡 18.12

用户可以查看自己的历史订单。

故事卡 18.13

查看历史订单时，用户可以方便地重新购买订单中的东西。

有了这些故事，Ron 船长能够查看他的以往购买信息并且从中再下订单。但是，Lori 指出 Ron 船长也希望能够购买他最近浏览过的书籍，即便是从来没有购买过的。为此，她写下故事卡 18.14。

故事卡 18.14

网站总是告诉购物者她最近三次查看的东西，并提供返回的链接。（即使在不同的会话中，这个功能也要能用。）

“初级航海者”的故事

接下来，团队考虑“初级航海者”角色。“初级航海者”的需要与 Teresa 和 Ron 船长的故事有很多重叠的地方。但 Lori 决定初级航海者能够看到一份我们的推荐清单会是有意义的。通过清单，“初级航海者”可以找到我们推荐的涵盖各种主题的书目。为此，她写下故事卡 18.15。

故事卡 18.15

用户可以看到我们针对各种主题推荐的书籍。

“不出海的礼物购买者”的故事

随着团队转到“不出海的礼物购买者”角色，大家开始讨论一个顾客能找到另一个人的愿望清单的重要性。他们甚至开始讨论不同的设计方案及用什么字段搜索，直到意识到设计讨论可以将来再进行。Lori 写下故事卡 18.16，而不是在会议上设计这个功能。

故事卡 18.16

用户，特别是“不出海的礼物购买者”，能够很容易查找到其他用户的愿望清单。

Lori 也知道系统需要支持礼物卡及包装。为此，她写出故事卡 18.17 及 18.18。

故事卡 18.17

用户可以选择对购买的东西进行礼物包装。

故事卡 18.18

用户可以选择附上礼物卡，并且可以在卡片上写上自己的信息。

“报表查阅者”的故事

Lori 说系统需要生成购买和流量规律模式等方面的报表。她还没有仔细考虑报表功能，所以开发人员写下一个简单的占位符故事，以提醒他们需要开发一些报表。他们会在日后决定报表的内容。目前，她写出故事卡 18.19。

故事卡 18.19

“报表查阅者”可以按照图书的类别、流量、最佳和最差销售情况来看每天的购买报表。

编写报表故事让 Lori 联想到它们的高度敏感性。当然，消费者访问的主网站上是不能看到这些报表的。但是，她指出只是公司内的几位人员有权访问这些报表。这可能表示只要你能看一份报表，就可以访问所有报表；或者表示部分用户只能访问部分报表。开发人员现在并不急于追问 Lori。她写下故事卡 18.20。



故事卡 18.20

用户必须经过适当的身份验证才能查看报表。

为了使这些报表有意义，Lori 明确指出网站必须使用现有的基于电话的系统的数据库。于是她写下显示这个限制条件的故事卡 18.21。

故事卡 18.21

在网站上提交的订单最终必须同电话订单放入同一个订单数据库。

(约束)

“管理员”的一些故事

至此，我们转移注意力到“管理员”用户角色。团队不假思索就列出故事卡 18.22 和 18.23。

故事卡 18.22

“管理员”可以在网站上增加新书。

故事卡 18.23

在评论放上网站之前，必须经过管理员批准。

添加新书的故事提醒团队管理员还需要能够删除书籍以及发现不正确的信息时能够编辑。因此，他们写出故事卡 18.24 和 18.25。

故事卡 18.24

管理员可以删除书。

故事卡 18.25

管理员可以编辑已有书的信息。

收尾

到此，Lori 想不出更多的新故事了。之前，每一个故事都是不费脑筋的，而现在她需要挖空心思寻找新的故事。因为项目会使用增量及迭代式开发过程，所以在开始时想出所有故事并不重要。但另一方面，Lori 期望能够了解大概需要花多久来开发这个系统，因此团队想在有限的时间内列出尽可能多的故事。若 Lori 在我们开始动工之后想到新的故事，她将有机会将它列入到这个发布中，前提是她需要替换出工作量大致相当的需求。

开发人员再次问 Lori 还有没有可能遗漏的其他故事。她写下故事卡 18.26。

故事卡 18.26

用户可以检查她最近的订单状态。如果订单还没有发货，她可以增加或删除书，更改送货方式、送货地址和信用卡信息。

而且，Lori 提醒开发人员虽然可扩展性要求不太高，但网站需要能够支持至少 50 个同时在线用户。于是，团队写下故事卡 18.27。

故事卡 18.27

系统必须能够支持 50 个并发用户。

(约束)



第 19 章 估算故事

故事编写作坊产生了 27 个故事，汇总在表 19.1 中。下一个目标是创建一个发布计划，给客户 Lori 展示开发人员期望完成什么，网站是否能在老板定下的 30 天期限内完成。因为很可能在 30 天里不能完成所有工作，开发人员要和 Lori 一起排列故事的优先级。

表 19.1 初始的故事集合

故事文本
用户可以用作者、书名或 ISBN 搜索书籍。
用户可以查看书籍的具体信息。例如：页数、出版日期和内容简介。
用户可以把书籍放进“购物车”，在结束购物时可以购买其中的书。
在完成订单前，用户可以从她的购物车中删除书籍。
购买书时，用户需要输入她的账单地址、送货地址和信用卡信息。
用户可以对书籍进行评分和发表评论。
用户可以创建一个网站账户用于保存送货和账单信息。
用户可以编辑她的账户信息(信用卡、送货地址和账单地址等)。
用户可以把书籍放入“愿望清单”，网站的其他访问者也可以看到该清单。
用户可以把愿望清单(甚至是别人的愿望清单)中的项目放入自己的购物车。
老顾客必须能够在 90 秒内找到书籍和下订单。
用户可以查看自己的历史订单。
查看历史订单时，用户可以方便地重新购买订单中的东西。
网站总是告诉购物者她最近三次查看的项目，并提供返回的链接。(即使在不同的会话中，这个功能也要能用。)
用户可以看到我们针对各种主题推荐的书籍。
用户，特别是“不出海的礼物购买者”，能够很容易查找到其他用户的愿望清单。
用户可以选择对购买的东西进行礼物包装。
用户可以选择附上礼物卡，并且可以在卡片上写上自己的信息。
“报表查阅者”可以按照图书的类别、流量、最佳和最差销售情况来看每天的购买报表。

续表

故事文本

用户必须经过适当的身份验证才能查看报表。

在网站上提交的订单最终必须与电话订单放入同一个订单数据库。

管理员可以在网站上增加新书。

在评论放上网站之前，必须经过管理员批准。

管理员可以删除书。

管理员可以编辑已有书的信息。

用户可以检查她最近的订单状态。如果订单还没有发货，她可以增加或移除书，更改送货方式、送货地址和信用卡信息。

系统必须能够支持 50 个并发用户。

为了创建发布计划，每个故事都需要一个估算。如同我们在第 8 章所学的那样，开发人员准备以故事点估算每个故事，故事点代表着理想日、复杂度或对团队有意义的其他一些度量值。

第一个故事

尽管没有必要从列表中的第一个故事开始(“用户可以用作者、书名或 ISBN 搜索书籍”),但这个案例中，第一个故事很适合我们开始做估算。当 Lori 编写这个故事时，开发人员不确定 Lori 的意思是允许用户同时用这些字段进行搜索，还是用户每次只能使用一个字段进行搜索。由于 Lori 的回答对估算有很大的影响，所以应该要先问问她。

Lori 当然说她希望两种都支持。她想要一个基本搜索模式，一个字段中的值既用于搜索作者，也会用于搜索书名。然后她也想要一个高级搜索界面，任何或所有这些字段都可用于组合搜索条件。即使故事同时包含两个搜索模式，它的大小也不算大，但是两种模式很容易分开，故而大家都同意撕掉这个故事卡，取而代之的是故事卡 19.1 和 19.2。

故事卡 19.1

用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。

故事卡 19.2

用户可以输入作者、书名和 ISBN，以它们的任意组合来搜索书。

为了估算故事，三位程序员(Rafe, Jay 和 Maria)和客户 Lori 来到一个房间。他们带来故事卡和一些空白卡片。程序员谈论到故事卡 19.1，通过询问 Lori 一些问题来澄清一些细节，然后每个程序员在索引卡片上写下他/她的估算。每个人都写好以后，各自拿出写好的卡片，让每个人都能看到。他们写下了以下内容：

Rafe: 1

Jay: $\frac{1}{2}$

Maria: 2

接着三位开发人员讨论他们的估算。Maria 解释她认为这个故事需要 2 个故事点的原因。她说他们需要选择一个搜索引擎，把它集成到系统中，接下来才能开始编写界面实现故事。Jay 说她已经对各种可能的搜索引擎非常熟悉，而且她对他们应该选择的方向相当自信，这就是她的估算很低的原因。

每个人都要求重写新的估算。完成后，他们再次展示他们的卡片。这回卡片上写着以下内容：

Rafe: 1

Jay: 1

Maria: 1

这相当容易。Jay 决定增加她的估算，而 Maria 确信他们可以做得比她原先设想的更快。这时他们对故事卡 19.1 的估算是 1 个故事点。他们开始写下估算，如表 19.2 所示。

表 19.2 开始写下估算

故 事	估算点数
用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。	1

注意，当程序员做出这些估算时，客户 Lori 在场，但她没有参与估算。因为 Lori 在项目中不是程序员，她不应该做估算。而且，她也不应该对估算表示任何的惊讶或者不赞同。假如她那么做，会妨碍大家的估算。当然，如果 Lori 听到一个似乎有点不靠谱的估算(要么太高，要么太低)，可能就需要她提供一些指导或澄清。例如，她可以提供大致如下的信息：“在你们描述的时候，我可以理解为何可能要 10 个故事点，但我觉得我想要的东西远比你们想的简单。我真正想要的是……”



高级搜索

接下来是故事卡 19.2 “高级搜索”。程序员再次在索引卡上写下他们的估算，同时把它们翻转过来，如下所示：

Rafe: 2

Jay: 1

Maria: 2

Rafe 说因为高级搜索有更多的搜索条件，故而所需时间长于基本搜索。Jay 同意 Rafe 的观点，但他觉得由于那时已经完成了基本搜索的编码工作，增加高级搜索的功能不会花太长时间。但是，Maria 指出故事之间是独立的，我们并不知道会先完成哪个故事。客户 Lori 说她不确定她想先完成哪个。她倾向于先完成基本搜索，但她不太确定，她需要知道每个故事的估算(即成本)。

在另外两轮估算后，大家都同意尽管高级搜索所需的工作比基本搜索稍多些，但并不会多太多，他们应该再次估算为 1 个故事点。

后面的几个故事估算起来很简单，没有必要分割。开发人员得出了如表 19.3 所示的估算。

表 19.3 建立估算清单

故 事	估算点数
用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。	1
用户可以输入作者、书名和 ISBN 的任意组合来搜索书。	1
用户可以查看书籍的具体信息。例如：页数、出版日期和内容简介。	1
用户可以把书籍放进“购物车”，在结束购物时可以购买其中的书。	1
在完成订单前，用户可以从她的购物车中删除书。	1/2
为了购买书，用户要输入她的账单地址、送货地址和信用卡信息。	2

评分和评论

下一个故事(“用户可以对书籍进行评分及发表评论”)稍微有点难。在写下估算并互相展示前,开发人员讨论该故事。评分部分似乎不难,但评论似乎更复杂。他们需要给用户提供一个界面,输入评论并预览。评论可以是纯文本,还是也可以是 HTML 类型?用户只能对他们从这里购买的书籍进行评论吗?

因为评论不只牵涉对书进行评分,我们决定分割该故事。由此得到故事卡 19.3 和 19.4。程序员对故事卡 19.3 的估算为 2 个故事点,故事卡 19.4 为 4 个故事点。

故事卡 19.3

用户可以对书进行 1(差)到 5(好)的评分。用户即使不从我们这里买书,也可以在此评分。

当他们正在考虑书的评分和评论时,他们也在考虑“在评论放上网站之前,必须经过管理员批准”。

故事卡 19.4

用户可以对图书写评论。在提交之前她可以预览评论。用户即使不从我们这里买书,也可以发表评论。

这可以很简单,或者可以牵涉更多,要求管理员给出驳回评论的原因,或者可能还要给评论者发送电子邮件。程序员觉得 Lori 不想要任何复杂的东西,他们的讨论结果是给这个故事 2 个故事点的估算。

账户

下一个故事(“用户可以创建一个网站账户用于保存送货信息和账单信息。”)似乎很简单,开发人员对它的估算是 2 个故事点。

接着,开发人员开始估算“用户可以编辑她的账户信息(信用卡、送货地址和账单地址等)”。这个故事不是很大,但很容易分割。分割像这样的故事通常是一个好主意,因为它可以让发布计划更灵活,而且它允许客户以更好的层次来排列工作优先级。例如,在我们的案例中, Lori 可能认为用户可以编辑他们信用卡这个功能很重要,但她可以等



几轮迭代再去实现用户可以修改地址的功能。分割好原始故事之后，产生了故事 19.5 和 19.6。这两个故事似乎都不难，所以程序员估算 19.5 为 $1/2$ 个故事点，19.6 为 1 个故事点。

故事卡 19.5

用户可以修改存储在她账户中的信用卡信息。

故事卡 19.6

用户可以修改存储在她账户中的发货地址和账单地址。

完成估算

同样的过程对其余每个故事重复进行。只有少数剩下的故事值得特别一提。首先是模糊的故事“用户，特别是“不出海的礼物购买者”，能够很容易查找到其他用户的愿望清单”。当被问及用户如何搜索愿望清单时，Lori 提供足够的细节，该故事可以重写为故事卡 19.7。

故事卡 19.7

用户，特别是不出海的礼物购买者，可以基于帆船主人的名字和所在州搜索愿望清单。

接着，大家都同意分割故事“用户可以检查她最近的订单状态。如果订单还没有发货，她可以增加或删除书，更改发货方式、送货地址和信用卡信息”。一个故事会覆盖检查最近的订单状态；另一个故事会覆盖更改还未发货的订单。如故事卡 19.8 和 19.9 所示。

故事卡 19.8

用户可以检查她最近订单的状态。

故事卡 19.9

如果订单还未发货，用户可以增加或移除书，更改发货方式、送货地址和信用卡信息。

最后，下面这三个故事是限制条件。

- 老顾客必须能在 90 秒内找到书和下订单。
- 在网站上提交的订单最终必须与电话订单放入同一个订单数据库。

- 系统必须能够支持 50 个并发用户。

作为限制条件，它们会影响其他故事，但它们本身不需要任何编码工作。

所有估算

表 19.4 展示了所有估算。

表 19.4 完成后的故事和估算清单

故 事	估算点数
用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。	1
用户可以输入作者、书名和 ISBN，以它们的任意组合来搜索书。	1
用户可以查看书籍的具体信息。例如：页数、出版日期和内容简介。	1
用户可以把书籍放进“购物车”，在结束购物时可以购买车里的书。	1
在完成订单前，用户可以从她的购物车中移除书。	$\frac{1}{2}$
为了购买书籍，用户要输入她的账单地址、送货地址和信用卡信息。	2
用户可以对书籍进行 1(差)到 5(好)的评分。用户即使不从我们这里买书，也可以在此评分。	2
用户可以对图书写评论。在提交之前她可以预览评论。用户即使不从我们这里买书，也可以发表评论。	4
在评论放上网站之前，必须经过管理员批准。	2
用户可以创建一个帐户用于保存送货信息和账单信息。	2
用户可以修改存储在她帐户中的信用卡信息。	$\frac{1}{2}$
用户可以修改存储在她帐户中的送货地址和账单地址。	1
用户可以把书放入“愿望清单”，网站的其他访问者可以看到该列表。	2
用户，特别是不出海的礼物购买者，可以基于帆船主人的名字和所在州搜索愿望清单。	1
用户可以检查她最近订单的状态。	$\frac{1}{2}$
如果订单还未发货，用户可以增加或移除书，更改发货方式、送货地址和信用卡信息。	1
用户可以把愿望清单中的项目(甚至是其他人的)放入到他/她的购物车中。	$\frac{1}{2}$
老顾客必须能够在 90 秒内找到书和下订单。	0
用户可以查看他的历史订单。	1
查看历史订单时，用户可以方便地重新购买订单中的东西。	$\frac{1}{2}$
网站总是告诉购物者她最近 3 次查看的项目，并提供返回的链接。(即使在不同的会话中，这个功能也要能工作。)	1
用户可以看到我们针对各种主题推荐的书。	4



续表

故 事	估算点数
用户可以选择对购买的东西进行礼物包装。	$\frac{1}{2}$
用户可以选择附上礼物卡，并且可以在卡片上写上自己的信息。	$\frac{1}{2}$
“报表查阅者”可以按照图书的类别、流量、最佳和最差销售情况来看每天的购买报表。	8
用户必须经过适当的身份验证才能查看报表。	1
在网站上提交的订单最终必须与电话订单放入同一个订单数据库。	0
管理员可以在网站上增加新书。	1
管理员可以删除书。	$\frac{1}{2}$
管理员可以编辑已有书的信息。	1
系统必须能够支持 50 个并发用户。	0





第 20 章 发布计划

创建发布计划需要以下步骤。

1. 确定迭代长度。
2. 估算速率。
3. 给故事安排优先级。
4. 将故事分配到一轮或多轮迭代中。

因为新网站功能需要在 4 周内交付，所以团队决定用两周长度的迭代。这样他们就有机会在截止日期前进行两轮迭代。他们将最高优先级的功能安排到第一轮迭代中，确保把它们完成。在第一轮迭代后，他们就可以评估速率，并决定在第二轮迭代中能做多少工作。

估算速率

Maria 和 Rafe 将是这个项目的程序员。Jay 曾帮助参与估算，但是因为有其他事情，他不能参与该网站的开发。因为这个项目与这些程序员之前做过的网站不同，他们不可能用前一个项目的速率来估算新项目的速率。所以他们需要一个有把握的猜测。

他们估算故事的时候，Maria 和 Rafe 没有经过深思熟虑就决定用一个理想日的工作量作为一个故事点。但现在他们发现一个理想日的工作量实际需要两三天才能完成。一轮迭代有 2 个星期(10 个工作日)和 2 个程序员，也就是一轮迭代有 20 人天。Maria 和 Rafe 估算他们每轮迭代能完成 7~10 个故事点。他们决定第一轮迭代估计得保守一些，估算速率为 8。

给故事安排优先级

作为客户，Lori 给故事安排优先级。决定故事优先级的主要因素是它所能交付的商业价值。然而，Lori 同样需要考虑故事的估算。有某些情况下，非常重要的故事会因为代价(估算)过高也会变得不那么必要。



开始排列优先级前，Lori 根据 4 周后的发布日期，按照故事的重要程度将故事卡按顺序排成 4 堆：必须要有的、应该有的、可以有的和不需要有的。如表 20.1 所示的是 Lori 的必须要有的故事。

表 20.1 在四周后第一次发布必须要有的故事

故 事	估 算
用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。	1
用户可以把书放进“购物车”，在结束购物时可以购买其中的书。	1
在完成订单前，用户可以从她的购物车中移除书。	½
为了购买书籍，用户要输入她的账单地址、送货地址和信用卡信息。	2
用户可以创建一个帐户用于保存送货信息和账单信息。	2
在网站上提交的订单最终必须与电话订单放入同一个订单数据库。	0
管理员可以在网站上增加新书。	1
管理员可以删除书。	½
管理员可以编辑已有书的信息。	1
系统必须能够支持 50 个并发用户。	0

必须要有的故事估算总和为 9。因为速率估算是每轮迭代 8 个故事点，有两轮迭代，所以还可以加一些应该有的故事。Lori 从应该有的故事堆中拿来如表 20.2 中所示的故事。她已经从必须要有的和应该有的故事中，找出了 15.5 个点，这已经接近程序员认为他们两轮迭代所能完成的 16 个点。

表 20.2 Lori 加入发布计划的应该有的故事

故 事	估 算
用户可以输入作者、书名和 ISBN，以它们的任意组合来搜索书。	1
用户可以修改存储在她账户中的信用卡信息。	½
用户可以修改存储在她账户中的送货地址和账单地址。	1
用户可以看到我们针对各种主题推荐的书。	4

最终的发布计划

最终的发布计划如表 20.3 所示，并且利用这个表与组织其他成员沟通。Maria 和 Rafe 将尽量完成第一轮迭代计划的工作。如果他们做得不错，他们将和 Lori 一起把一两个故

事挪到第一轮迭代。如果他们落后于计划，他们将和 Lori 一起从第一轮迭代挪一两个故事到第二轮迭代中。

表 20.3 最终的发布计划

迭代 1	迭代 2
用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。	管理员可以编辑已有书的信息。
用户可以把书籍放进“购物车”，在结束购物时可以购买其中的书。	用户可以输入作者、书名和 ISBN，以它们的任意组合来搜索书。
在完成订单前，用户可以从她的购物车中移除书籍。	用户可以修改存储在她账户中的信用卡信息。
为了购买书，用户要输入她的账单地址、送货地址和信用卡信息。	用户可以修改存储在她帐户中的送货地址和账单地址。
在网站上提交的订单最终必须与电话订单放入同一个订单数据库。	用户可以看到我们针对各种主题推荐的书籍。
用户可以创建一个账户用于保存送货信息和账单信息。	
管理员可以在网站上增加新书。	
管理员可以删除书。	
系统必须能够支持 50 个并发用户。	





第21章 验收测试

故事的验收测试用于决定故事是否完成，测试通过时客户可以接受它作为软件的一部分已经完成的事实。这意味着客户负责定义测试。尽管，通常客户需要得到项目中测试人员的帮助。由于该项目规模较小，没有专门的测试人员，Lori 可以争取获得 Maria 和 Rafe 的帮助。这样带来的另外一个好处是，除了可以列出验收测试清单，也会让 Lori 和程序员之间做进一步讨论。

搜索测试

Lori 列入第一个发布的搜索功能如故事卡 21.1 和 21.2 所示。故事 21.1 的测试如下。

- 用一个单词(如“navigation”)来搜索，这个单词是书名的一部分，而不是作者姓名的一部分。
- 用一个单词(如“john”)来搜索。这个单词应该是作者姓名的一部分，而不是书名的一部分。
- 用非书名非作者名的单词(如“wookie”)进行搜索。

故事卡 21.1

用户可以做基本的简单搜索，输入的单词或词组会同时在作者和书名中匹配。

故事 21.2 的测试如下。

- 用至少符合一本书的作者及书名进行搜索。
- 用不符合任何一本书籍的作者及书名搜索。
- 尝试使用 ISBN 搜索。

故事卡 21.2

用户可以输入作者、书名和 ISBN，以它们的任意组合来搜索书。



购物车测试

故事 21.3 和 21.4 涉及购物车的使用。

故事卡 21.3

用户可以把书放进“购物车”，在结束购物时可以购买其中的书。

故事卡 21.4

在完成订单前，用户可以从她的购物车中删除书。

Lori 和程序员讨论这些故事，认识到他们有一些悬而未决的问题：用户可以把缺货的书放入购物车吗？对于尚未印刷的书呢？此外，团队意识到故事卡 21.4 涉及了更改某商品数量为 0 的情况，但还没有明确的故事增加某商品的数量。他们可以编写一个单独的故事，但他们决定撕掉故事卡 21.4 并用故事卡 21.5 取而代之。

故事卡 21.5

用户可以调整购物车中某商品的数量。把数量设为 0 即可从购物车中删除该商品。

这次讨论的重要结果是该系统被简化了。通过决定不需要单独的故事从购物车中删除商品，团队既提高了系统可用性，也避免了今后的可能的工作。

故事 21.3 的测试如下。

- 把缺货的书放入购物车。验证系统是否会告诉用户书将在补货后发送。
- 把尚未出版的书籍放入购物车。验证系统是否会告诉用户书将在货到后发送。
- 把库存的书放入购物车。
- 两次放入同一本书。验证书的数量是否增加。

故事 21.5 的测试如下。

- 从 1 到 10 修改书的数量。
- 从 10 到 1 修改书的数量。
- 通过修改数量为 0 来删除书。

知乎
PDG

购买书

故事卡 21.6 涉及实际购买书的过程。在讨论该故事时，程序员与客户 Lori 澄清了少数几个方面。Lori 想让用户可以输入单独的送货地址和账单地址，或者指明两个地址是相同的。这个网站只接受 Visa 卡和万事达卡。

故事卡 21.6

为了购买书，用户要输入她的账单地址、送货地址和信用卡信息。

故事 21.6 的测试如下。

- 输入账单地址，并指明与送货地址是相同的。
- 输入单独的账单地址和送货地址。
- 以一个州名和其他州的邮编来测试，并验证系统能否捕捉到两者的不一致性。
- 验证发送书的地址是送货地址，而不是账单地址。
- 用有效的 Visa 卡测试。
- 用有效的万事达卡测试。
- 用有效的美国运通卡测试(失败)。
- 用过期的 Visa 卡测试。
- 用超过信用额度的万事达卡测试。
- 用缺少位数的 Visa 卡号测试。
- 用错位的 Visa 卡号测试。
- 用完全无效的 Visa 卡号测试。

用户账户

故事卡 21.7 涉及创建用户账户。此故事卡的测试如下。

- 不用创建账户，用户就可以下订单。
- 创建一个账户，然后重新访问它，查看信息是否已经保存。

故事卡 21.7

用户可以创建一个账户用于保存送货信息和账单信息。

故事卡 21.8 和故事卡 21.9 允许用户修改存储在他们账户里的信息。故事卡 21.8 的测试如下。

- 编辑信用卡的卡号，使其成为一个无效卡号。验证系统应该警告用户。
- 把过期日期修改成过去的日子。验证修改不会被保存。
- 把信用卡卡号修改成新的有效的卡号，确保修改被保存。
- 把过期日期修改成未来的日子，确保修改被保存。

故事卡 21.8

用户可以修改存储在她账户里的信用卡信息。

故事卡 21.9 的测试如下。

- 修改送货地址的不同部分，并验证修改被保存。
- 修改账单地址的不同部分，并验证修改被保存。

故事卡 21.9

用户可以修改存储在她账户里的送货地址和账单地址。

管理

故事卡 21.10 允许管理员在网站上增加新书。这个故事的测试如下。

- 测试管理员可以在网站上增加书。
- 测试非管理员不能增加书。
- 测试只有在必需数据都已经输入的情况下才能增加书。

故事卡 21.10

管理员可以在网站上增加新书。

故事卡 21.11 允许管理员删除书。这个故事的测试如下。

- 验证管理员可以删除书。
- 验证非管理员不能删除书。
- 删除一本书，然后验证已经提交的购买此书的订单仍会发货。

故事卡 21.11

管理员可以删除书。

故事卡 21.12 允许管理员修改书的信息。当程序员和 Lori 讨论故事卡 21.12 时，他们讨论如何处理那些订购了价格已变动但尚未发货的书订单。这变成了该故事的测试之一。

- 验证名字、作者、页数等图书信息可以被修改。
- 验证价格是可以修改的，但价格变化不会影响之前提交(但尚未入账和尚未发货的)的订单。

故事卡 21.12

管理员可以编辑已有书的信息。

测试限制条件

在 Lori 已经排入发布的故事中，有两个是限制条件，如故事卡 21.13 和故事卡 21.14 所示。

故事卡 21.13

在网站上提交的订单最终必须与电话订单放入同一个订单数据库。

故事卡 21.14

系统必须能够支持 50 个并发用户。

故事卡 21.13 唯一的测试是检查数据库，并验证从网站提交的订单会保存到数据库。

- 提交订单。打开电话订单数据库输入，并验证订单会保存到数据库。

Lori 拿着故事卡 21.14，翻转故事卡并写下以下内容。

- 以 50 个模拟用户，执行不同类型的查询和提交订单操作。确保没有界面会在 4 秒钟之后才显示出来，并且不会丢失订单。

最后一个故事

最后一个故事如故事卡 21.15 所示。

故事卡 21.15

用户可以看到我们针对各种主题推荐的书籍。

Lori 和开发人员讨论故事卡 21.15，并决定它将是一个简单的静态页面，包括不同主题的推荐列表。他们写下以下测试。

- 选择一个主题(例如，导航或巡航)并查看那个主题的推荐。确保它们很容易理解。
- 点击列表中的项，验证浏览器会跳转到相关书的信息页面。



第 V 部分 附 录

要读懂本书并从中获益，并不需要精通极限编程。但是，因为用户故事起源于极限编程，所以我们利用附录 A 做一个简单的介绍。附录 B 列出了各章末尾大多数问题的答案。



附录 A 极限编程概览

本附录将简要介绍极限编程(Extreme Programming, 后面简称 XP)的主要思想。假如你已经熟悉极限编程, 尽可以放心地略过本附录。如果不是, 请将本附录当作对极限编程的介绍, 然后再去阅读深入剖析极限编程的好书^①。

首先了解参与 XP 项目的人员(或角色)。接着看看 XP 的 12 个主要实践。最后, 我们思考 XP 团队的价值。

角色

XP 的客户角色负责编写故事、排列故事优先级以及编写和执行测试, 以验证故事按照预期进行开发。XP 客户可以是系统的用户, 但也可以不是。如果不是用户, XP 的客户往往是产品经理、项目经理或业务分析师。

在有些项目中, 客户角色可能实际上由客户团队充当, 由多个对项目感兴趣的个人组成。客户团队通常包括协助创建验收测试的测试人员。当项目有多个客户时, 他们之间达成共识很重要。许多方式可以帮助多个客户达成共识, 但最常见的是在客户团队中指定一个带头人。

XP 的程序员角色拥有广泛的技术技能。XP 项目往往不区分程序员、设计师、数据库管理员等角色。所有程序员以团队为单位一起工作, 并一起承担许多责任, 这些责任在非 XP 项目中可能会指派给特定的个人。几乎所有过程都期望程序员对他们的代码进行单元测试, XP 尤其重视这一点, 并期望程序员对他们编写的所有代码开发自动化单元测试。

① 重点推荐 《解析极限编程: 拥抱变化》(Extreme Programming Explained: Embrace change, Beck 2000)、《极限编程实施》(Extreme Programming Installed, Jefferies, Anderson 和 Hendrickson 2000)或《探索极限编程》(Extreme Programming Explored, Wake 2002)。



最后,许多XP团队都受益于使用XP教练和(可能的话)项目经理这两个角色。有时候,这些角色合为一体。教练负责监控团队应用XP实践的情况,在团队偏离时逐步调整进而将团队带回正确的轨道。项目经理更像是排头兵而不是经理,他负责防止团队的官僚作风,并千方百计地为团队移除障碍。

12 个实践

极限编程的特征是 Kent Beck 在最初的白皮书(Beck 2000)中描述的 12 个实践。如果选择在项目中尝试使用 XP,我们鼓励采用所有实践。XP 的 12 个实践是高度协作和相互依存的。每个实践支持并允许其他实践。例如,通过采用结对编程、简单设计、集体所有权、持续集成以及测试,实践重构变得更加容易。这 12 个实践并不是好想法的随机集合,所以要为 XP 团队选择适合的实践。随着经验的丰富,可以选择放弃或改变某个实践,但在熟悉标准的 XP 之前,确实应该尽量避免自己定制。

在本附录中,我们将关注以下 12 个 XP 实践:

- 短交付周期(small releases)
- 计划游戏(the planning game)
- 重构(refactoring)
- 测试(testing)
- 结对编程(pair programming)
- 持续一致的速度(sustainable pace)
- 团队代码所有权(team code ownership)
- 编码标准(coding standard)
- 简单设计(simple design)
- 隐喻(metaphor)
- 持续集成(continuous integration)
- 现场客户(on-site customer)

短交付周期

XP 项目在一系列迭代中不断进展,每轮迭代通常是 1~3 周时间。用户故事所描述的功能,在单轮迭代里是完全可交付的。团队不允许交付未完成的功能。类似地,团队不允许交付只达到一半质量标准的完整功能。在每一轮迭代结束时,团队负责交付可运

行的、已测试的代码，可以马上投入使用。

在项目开始时，团队选择一轮迭代长度，在项目进行期间使用该长度。迭代长度通常是 1 或 2 周，从不会超过 4 周。团队应选择尽可能短的迭代长度，但要确保仍能够交付可见的商业价值。如果拿不准如何取舍两个不同的迭代长度，就选择较短的长度。

迭代是固定的时间箱。团队不能在既定迭代的最后一天决定他们还需要两天。迭代在规定的日期结束。可以调整团队的工作量(但不是工作质量)以适应迭代。

计划游戏

“计划游戏”是 XP 中发布计划和迭代计划的名称，在做计划时，开发人员和客户一起对未来做出预测。开始计划前，客户已经在笔记卡上写好用户故事，开发人员已经估算好每个故事的成本或大小，并已将估算写在故事卡上。

开始计划时，开发人员估算他们能在项目选定的迭代长度中完成多少工作。然后，客户仔细查看所有故事卡，选择最高优先级的故事放入第一轮迭代。她可以选择的故事工作量，不能超过开发人员估算的一轮迭代可以完成的工作量。第一轮迭代填满工作之后，客户接着为第二轮和后面几轮迭代选择故事。

这样做完几轮迭代的计划后，由客户决定各轮迭代中是否已经有足够的故事，以形成一次发布。几乎可以肯定，发布计划不会准确反映哪些故事会被开发，以及它们以何种顺序开发。发布计划是关于开发如何进行的假设，在迭代开始、优先级改变，团队的进展速度更加明确，开发人员更加了解每个故事的实际预期成本后，可以更新发布计划。

每次迭代开始前，团队和客户计划迭代。包括选择可以在迭代中完成的最高优先级故事，接着识别完成故事所需的特定任务。

重构

重构(Fowler 1999; Wake 2003)指的是重组或重写代码以改善代码，但不改变其外部行为。随着时间推移，代码会变得丑陋。一个为单一目的设计的方法，可稍加改变以处理某种特殊情况。然后，由于它已经处理了那种特殊情况，所以可以再次修改以处理另一种特殊情况。以此类推，直到该方法变得非常脆弱，难以修改。

XP 主张不断关注重构。一旦有程序员更改代码，发现应该进行重构时，就必须重构它。不是鼓励她重构，而是她必须进行重构。这样可以避免代码中缓慢的但有时很难发



现且最终导致代码被废弃的腐朽。

重构是 XP 用于取代前期设计的方法之一。不是先期花时间去思考系统，而是先编码，因此要猜测其行为的某些方面，XP 系统不断重构，并确保很好地实现当前已知的需求。

测试

XP 令人兴奋的一个实践是对测试的重视。XP 项目中，开发人员编写自动化单元测试，客户编写验收测试。通常情况下，验收测试要么是客户自己编写，或者求助于开发人员。许多 XP 开发人员确实发现早期频繁测试的好处。此外，开发人员对测试的抵触减少了，因为 XP 的单元测试通常通过编写使用业务代码的测试代码来实现自动化，也就是说，即使是测试，他们也在编程。

在传统开发过程中，测试是在代码编写完以后编写的(如果确实会编写测试的话)。这是一个问题，因为一旦代码编写完、功能可以工作后，出于人类的天性，不太会去改进代码。因此，许多开发人员只会轻描淡写地测试一下自己的代码。(我知道这种情况：我过去也是其中一员。)XP 改变了这种情况，并把测试先行的实践称之为测试驱动开发(test-driven development, Beck 2003; Astels 2003)。

在测试驱动开发中，测试在编码之前编写。开发人员遵循“测试-编码-测试-编码”的短周期(分钟，而不是小时)。他们遵守一条规则，除非有相应的失败测试，否则不写业务代码。所以，他们先编写一个失败的测试。运行程序以验证测试会失败。只有这样做了，程序员才编写代码，使程序通过测试。

测试驱动开发可以保证代码保持良好的状态和可测试性。它也有助于产出易于维护的代码，因为从一开始，代码就处于有效的维护模式。

除了程序员的单元测试，客户测试是 XP 很重要的一部分。对每个故事，客户负责定义一系列测试，用来确定开发好的故事是否符合他们的期望和假设。在许多方面，这些客户编写的验收测试会取代瀑布过程的需求文档。

结对编程

XP 更具争议性的实践之一是结对编程。结对编程指的是两个开发人员共用一个键盘和一台显示器，但使用两个大脑编写代码。当一个程序员在敲击键盘时(脑子里正在思考下面几行代码)，另一个程序员注视正在开发的代码，并且思考得更广泛一些，例如这些

代码会在哪里导致什么样的问题。结对过程中，经常切换角色和搭档。

虽然结对编程听起来效率极其低下，Alistair Cockburn 和 Laurie Williams (2001) 已经研究过，并发现事实并非如此。他们发现，对于总体编程时间增加 15% 来说，结对编程可以带来以下好处：

- 更少的缺陷
- 解决相同的问题编写的代码更少
- 解决问题更快
- 理解每一块代码的人更多
- 开发工作的满意度增加了

结对编程对 XP 很重要，因为其他很多 XP 实践需要行为准则来保证。每次发现结构不良的代码，都需要大量的行为准则来做重构，或者总是在编写业务代码之前编写测试。没有结对的话，非常容易这样想“就这一次……”然后就跳过重构或测试。

持续一致的节奏

XP 鼓励团队以持续一致的节奏进行工作。XP 团队相信，相较于以无法持续的节奏进行工作，以一致轻快的步伐前进，能在一段时间内完成更多工作。这不是说 XP 团队每周精确地工作 40 个小时，然后回家。由团队来决定他们的可持续节奏，团队中不同成员很可能会有不同看法。

结对编程和测试驱动开发(test-driven development)是有效的，因为它们要求创建代码时两人的注意力非常集中。很少有人能够长时间保持这种强度的水平。团队通常会致力于每天结对 6 小时左右，剩下的时间花在其他事情上。

XP 教练负责监视团队是否过度劳累。如果教练感觉团队太劳累，她会帮助团队回归到可持续的节奏。

团队代码所有权

在非 XP 团队中，个别开发人员“拥有”或承担系统部分代码的全部责任，这是很普遍的。如此，系统每个部分将只有一位开发人员负责——至少直到某位开发人员挪到其他项目时，她的代码还是没有负责人。这种对于代码所有权的观点也导致团队做出这样的评论：“在 Eli 从假期回来前，我们不能修改计费模块的源代码”。此外，如果一个开发人员在 Eli 休假时改动了他的代码，Eli 回来后很可能对“他的代码”被修改



而感到不快。

XP 团队对代码所有权采用完全不同的方法：每人都拥有所有代码。在这样的团队所有权模型下，任何结对的开发人员都可以修改任何代码。事实上，由于重构的实践，结对者要修改其他人写的代码。

个人所有权用以确保一致的设计，并保持一个模块所有责任的平衡。在 XP 中，这个重担由测试驱动开发承担。一套健壮的单元测试可以确保修改不会引入意外的副作用。

编码标准

因为 XP 团队集体拥有他们的源代码，所以遵循同一个编码标准很重要。编码标准规定了主要的规则和约定，团队成员在编写代码时都要遵守这一标准：如何命名变量和方法？如何格式化代码行？等等。

小型严密的团队可以没有书面化的、正式的编码标准。他们可以通过团队的习惯建立并共享标准。除了一小部分开发人员，大多数团队会从书面的编码标准中获益，但仍要尽可能保持其简短和必要性。

简单设计

XP 团队追求使用最简单的设计去交付客户所需的功能。Kent Beck(2000)定义了四个约束，用以表明某种设计是最简单的。

1. 业务代码和测试代码充分表达程序员的意图。
2. 没有重复代码。
3. 系统使用最少数量的类。
4. 系统使用最少数量的方法。

隐喻

XP 团队通过寻找一个适用于整个系统的隐喻来支持对简单设计的追求。这个隐喻提供了一个他们如何思考系统的参照体系。例如，在一个项目中，我们的隐喻是系统就像一块黑板，系统的不同部分可以写在黑板上。用户完成工作后，她可以保存黑板上的内容，也可以将内容擦掉。这样考虑系统的方式极为简单，为我们提供了一种方便、简单的方式来思考系统的行为。

持续集成

我最近和一家公司的经理一起参与一个讨论，他们是最大的电子商务公司之一。他告诉我整合多个开发人员的工作对于大部分软件开发团队而言是最大的问题。他喜欢让他的团队每月集成一次他们的软件，这样他们可以避免因不经常整合而导致更大的问题。我问他如果他的团队每天都做集成会怎样。

XP 团队知道答案，他们每天至少做一次集成。我们很久以前就知道每天做构建(build)和冒烟测试(smoke test)的好处(Cusumano 和 Selby 1995)。XP 团队，或多或少都会对代码进行持续集成。例如，一位开发人员完成了一个小的变更，她把这个修改签入到源代码库，有一个进程注意到这个变更然后启动一个完整的构建过程。构建完成后，一套自动化测试自动运行。有任何测试失败，都会有电子邮件发送给开发人员，告诉他有关的失败信息。出现集成问题时，每次往往只需要少量的修改就能够解决。

现场客户

过去，客户编写好需求文档，把它扔给墙外编写代码的程序员，接着又把系统抛过墙给一些测试人员，这种情况是很普遍的。使用 XP，这道墙不复存在，客户将和开发团队坐在一起，并成为团队的一部分。客户编写故事和验收测试，并当场尽快回答团队的问题。

现场客户对于成功使用用户故事的方法是至关重要的，因为客户和开发人员之间必然会有许多讨论。如果客户不在现场，延迟将打乱 XP 团队所预测的进度。

极限编程的价值

除了 XP 的实践，XP 还提倡四大价值：沟通(communication)、简单(simplicity)、反馈(feedback)和勇气(courage)。XP 重视沟通，但并非所有沟通模式都有同样的效果。最理想的沟通模式是面对面的沟通，我们可以谈话、回应、做手势，也可以在白板上画图。最不希望的沟通模式是书面化文档。XP 强调通过结对编程等类似实践进行沟通。

简单是 XP 团队的价值，因为它关注为当下遇到的问题创建解决方案，而不会关注未来预期的问题。XP 团队只为当前迭代要开发的功能进行支持，而不会去架构一个能支持其他功能的系统。他们持续不断地关注于做最简单的、可行的东西。



XP 团队重视反馈，反馈越快越好。在结对编程的过程中，当一位开发人员对她的结对者指出潜在问题时，前者给出反馈，后者获取反馈。他们从经常执行的自动化测试中获得反馈。他们从持续(或至少每天)集成的过程中获得反馈。客户是团队的一部分，甚至与开发人员坐在一起，他们通过与团队的频繁交互以及他们编写的验收测试为团队提供反馈。

最后，XP 团队重视勇气。例如，他们有勇气重构他们自己的代码(因为他们有自动化测试作为坚实后盾)。他们有勇气在没有整体细致架构的情况下前进，因为他们会使用隐喻，并通过重构和测试驱动开发来维持简单的设计。

极限编程的原则

除了 XP 的价值和实践外，XP 具有五项基本原则：快速反馈(rapid feedback)、假设简单(assuming simplicity)、增量变化(incremental change)、拥抱变化(embrace change)以及高品质的产品(doing quality work)(Beck 2000)。自从引入 XP 后，针对一个团队如果只做到原来 12 个实践中的 11 个，那他们是否在做 XP，争论一直不断。如果一个团队追求简单设计，但他们会用最初几周时间来建模，请问他们是在做 XP 吗？

我认为答案是肯定的。如果他们遵循了 XP 的原则，这些团队就是在做 XP。这样的团队具有以下待质。

- 向客户提供快速反馈，并从反馈中学习。
- 喜欢简单，并总是试图在转移到更复杂的解决方案前，先做简单的解决方案。
- 通过小的、增量的变化提高软件质量。
- 拥抱变化，因为他们知道自己是真正地善于包容和适应。
- 坚决主张软件应该始终展示出最高的质量工艺。

毫无疑问，这是在做极限编程，即便他们缺少一两个实践。

小结

- XP 的客户角色负责编写故事和每个故事的验收测试，并与开发团队坐在一起。
- XP 项目中，程序员和测试人员之间的区别是模糊的。
- XP 项目包括一位教练，可能的情况下还包括一位单独的项目经理，负责指导团

队并移除障碍。

极限编程牵涉以下实践。

- 短交付周期(small releases)
- 计划游戏(the planning game)
- 重构(refactoring)
- 测试(testing)
- 结对编程(pair programming)
- 持续一致的速度(sustainable pace)
- 团队代码所有权(team code ownership)
- 编码标准(coding standard)
- 简单设计(simple design)
- 隐喻(metaphor)
- 持续集成(continuous integration)
- 现场客户(on-site customer)

极限编程具有以下价值。

- 沟通(communication)
- 简单(simplicity)
- 反馈(feedback)
- 勇气(courage)

极限编程具有以下关键原则。

- 快速反馈(rapid feedback)
- 假设简单(assuming simplicity)
- 增量变化(incremental change)
- 拥抱变化(embrace change)
- 高品质产品(quality work)



附录 B 参考答案

第 1 章 概览

1.1 用户故事包含哪三大部分？

答案：卡片、交流和确认。

1.2 客户团队由哪些人组成？

答案：客户团队包括那些保证软件达到未来用户需求的人。这可能包括测试人员、产品经理、真实用户和交互设计人员。

1.3 以下哪些不是好的用户故事？为什么？

- a. 用户可以在 Windows XP 和 Linux 上运行系统。
- b. 所有绘图和图表将用第三方类库完成。
- c. 用户可以最多撤销 50 步操作。
- d. 软件将在 6 月 30 日发布。
- e. 软件将用 Java 编写。
- f. 用户可以从下拉列表框里选择她的国籍。
- g. 系统将使用 Log4J 记录所有错误信息到一个文件。
- h. 如果用户 15 分钟内没有保存文档，系统将提示用户进行保存。
- i. 用户可以选择“导出到 XML”特性。
- j. 用户可以导出数据到 XML 文件。

答案：如表 B.1 所示。

表 B.1 问题 1.3 的答案

故 事		答 案
a.	用户可以在 Windows XP 和 Linux 上运行系统。	这是一个好故事。
b.	所有绘图和图表将用第三方类库完成。	这不是一个好故事。 用户不会关心图表是怎么实现的。
c.	用户可以最多撤销 50 步操作。	这是一个好故事。



续表

故 事		答 案
d.	软件将在 6 月 30 日发布。	这不是一个好故事。这是一个需要在发布计划中考虑的限制条件。
e.	软件将用 Java 编写。	这可能不是一个好故事，但是它依赖于产品。如果产品是一个面向 Java 程序员的类库，那些用户会比较关心使用的语言。
f.	用户可以从下拉列表框里选择她的国籍。	这是一个好故事，但它可能小了些。
g.	系统将使用 Log4J 记录所有错误信息到一个文件。	这不是一个好故事。它不应该指定使用 Log4J 实现日志功能。
h.	如果用户 15 分钟内没有保存文档，系统将提示用户进行保存。	这是一个好故事。
i.	用户可以选择“导出到 XML”特性。	这是一个好故事。
j.	用户可以导出数据到 XML 文件。	这是一个好故事。

1.4 需求对话相对于需求文档而言，有哪些优势？

答案：使用文档意味着需求是精确的，但这是文档无法保证的。用户故事用卡片作为对话的提醒，避免了需求非常精确的假象。把东西记录下来并不能保证客户能得到他们想要的，最好情况也只是客户得到了文档所写下的那些东西。频繁的沟通，特别是在开发过程中讨论功能，可以让开发人员和客户加强互相的理解。

1.5 为何在故事卡背面写测试描述？

答案：在卡片的背面写测试对于客户是沟通故事的期望和假设的非常好的方法。

第 2 章 编写故事

1.1 以下故事中，哪些故事是好的故事，哪些是不好的故事。如果不是好的故事，请说明理由。

- 用户能快速掌握系统。
- 用户可以修改简历上的地址。
- 用户可以增加、修改和删除多份简历。
- 系统可以计算 n 元二次型方程分布的鞍点近似值。
- 运行期错误都用同样的方法记录。

答案：见表 B.2。

表 B.2 问题 2.1 的答案

故 事		答 案
a.	用户能快速掌握系统。	这个故事应该修改。“快速”和“掌握”没有定义。
b.	用户可以修改简历上的地址。	这个故事可能太小了，但是取决于开发人员实现这个故事可能要花多长时间。
c.	用户可以增加、修改和删除多份简历。	这是一个复合故事，应该分成多个故事。
d.	系统可以计算 n 元二次型方程分布的鞍点近似值。	如果这个故事是客户写的，她可能知道这是什么意思。但是，如果开发人员不理解这个故事，客户应考虑重写这个故事(或者至少好好讨论这个故事)，这样开发人员才可以估算它。
e.	运行期错误都用同样的方法记录。	这个故事没问题。

1.2 将这句史诗故事分解为适当大小的故事：“用户可以设置、更改职位自动搜索工具。”

答案：这个史诗故事至少应分为 2 个故事，一个是设置工具，一个是更改工具。当然，还有许多不同的方式分解，这依赖于故事大概需要多长时间实现。下面是可能的分解方式：

- 用户可以设置职位自动搜索工具。
- 用户可以修改职位自动搜索工具的搜索参数。
- 用户可以修改职位自动搜索工具的执行次数。
- 用户可以修改职位自动搜索工具如何发送搜索结果报告。

第 3 章 用户角色建模

3.1 以易趣(ebay)为例。你能识别它有哪些用户角色吗？

答案：你的答案应当包括类似这样一些角色：一次性的卖家，小卖家，超级卖家，偶尔在网上购物的买家，超级买家，企业卖家，生产厂商，付费平台，收藏家，会员，软件开发人员，联盟机构，无线卖家，无线买家。

3.2 整合前一个问题想出来的角色，展示一下如何排列这些角色卡。然后解释一下回答。

答案：从我的答案中，我把卖家整合成一个通用的卖家角色和另外三个特定卖家：小卖家、超级卖家和企业卖家。类似地，我有一个通用的买家角色，细分

为偶尔在网上购物的买家、超级买家和收藏家。我还保留了付费平台、联盟机构和一个通用的无线用户。

3.3 为其中最重要的用户角色编写虚构人物描述。

答案：Brenda 是一个超级买家。一般她会一个星期至少访问一次网站，平均每个星期下 1 到 2 个订单。她一般买些电影和书籍，但是她还买园艺和厨房用品。她是一个房产经纪人，对我们的网站很满意，但是有点不喜欢学习大多数新软件。她通常在家通过拨号访问网站，偶尔也从公司里较快的网络访问网站。

第 4 章 搜集故事

4.1 如果团队只通过问卷调查来搜集需求，你会碰到哪些问题？

答案：问卷调查需要花很长时间，因而项目也会花更长的时间。需要一个人来搜集结果并分析，这意味着存在一些误解。因为问卷调查不提供真正的双向沟通，团队很难获得反馈，很难确定是否找准了方向。

4.2 将下面的问题重新整理成开放式背景无关的问题：你认为用户必须输入密码吗？系统应每 15 分钟自动保存用户的操作么？用户能看到另一个用户的数据库录入信息么？

答案：修改这些问题有很多版本。下面是一些例子：

- 描述系统如何保护敏感信息。
- 当用户使用系统时系统崩溃了，用户该怎么办？
- 告诉我用户如何方便地访问所保存的数据。

4.3 为什么最好问开放式的、与上下文无关的问题？

答案：与上下文无关的问题不会暗示答案（“你什么时候停止殴打你的妻子了？”），受访者不会觉得需要给一个“正确”的答案。开放式问题可以有详细的回答，而不是一个简单的是或不是。开放式上下文无关的问题是最好，因为它们不会影响到回答，通常回答会比是或不是的范围要广得多。

第 5 章 与用户代理合作

5.1 让用户的经理充当用户代理，会导致什么问题？

答案：即使用户的经理也是软件的用户，她的需求也会与其他用户不同。更糟的是，如果她曾经是用戶，她对系统的了解是过时的。

5.2 让领域专家充当用户代理，会导致什么问题？

答案：首先，领域专家可能不是系统的用户。如果她是，她使用系统的方式可

能会和其他用户不同。其次，最终你得到的系统是非常适合这些专家的，但是对那些相对缺少领域知识的用户来说不是那么易用。

第 6 章 用户故事验收测试

6.1 哪些人定义测试？哪些人提供帮助？

答案：客户定义测试。实际上，客户一般和程序员或测试人员一起创建这些测试，但是客户至少需要定义好测试，这些测试告诉我们怎样才能确定故事已被正确实现。

6.2 为什么要在写代码前定义测试？

答案：在写代码前定义测试是因为测试是与客户讨论新功能非常有效的方法。

第 7 章 优秀用户故事准则

7.1 假设用户故事“求职者可以搜索职位空缺”太大了，不适合放入一轮迭代。你怎么分割它？

答案：我们可以根据支持的搜索参数来分割这个故事，如地点，关键字，职位，薪水，等等。另外，还有不同的显示结果的方式。初始的故事可以是一个非常简单的匹配职位的列表。下一个故事可以在这个故事基础上加强结果的显示，可能是显示更详细的职位信息，允许用户选择一个排序顺序或显示哪些字段，或提供一个链接查看职位的更详细信息。

7.2 以下用户故事中，那些故事的大小适中，并且可以认为是封闭的故事？

- a. 用户可以保存她的偏好。
- b. 用户可以修改用于购买的默认信用卡。
- c. 用户可以登录系统。

答案：见表 B.3。

表 B.3 问题 7.2 的答案

	故 事	答 案
a.	用户可以保存她的偏好。	这个故事可能是封闭的，也可能不是，这依赖于系统。如果保存偏好是用户可能想做的一件事，可以认为它是闭合的。它可能小了些，但也还可以，同样取决于系统和团队怎么构建它。



续表

	故 事	答 案
b.	用户可以修改用于购买的默认信用卡。	这是一个封闭的大小适中的故事。
c.	用户可以登录系统。	这个故事不是封闭的而且可能太小了。

7.3 应该怎样修改故事“用户可以发布他们的简历”，使它更好一些？

答案：用户是否能发布多个简历在这里不是很清晰。毫无疑问讨论这个故事时会考虑到这个问题，但是把故事写成“求职者可以发布一个或多个简历”要好些。

7.4 应该怎样测试“软件要易于使用”这样一个约束？

答案：要测试这个约束，首先需要定义“易于使用”是什么意思。是不是说一个熟练用户可以用最少的按键来完成一般的工作？或者是不是意味着一个新用户可以很快的达到一个使用软件的指定的熟练级别？一般是后者的意思。如果是这样，可以像这样定义 1 个或多个测试：

- 一个新用户在第一次见到系统的 30 分钟内，可以搜索职位，在系统中注册，发布她的简历。

像这样的测试是不可能被自动在每晚构建中执行，通常我们可以观察新用户第一次使用系统来做易用性测试。

第 8 章 估算用户故事

8.1 在估算会议上三个程序员在估算一个故事。他们分别估算故事为 2，4 和 5 个故事点。他们应该用哪个估算？

答案：他们应该继续讨论这个故事，直到他们的估算更加接近。

8.2 三角测量估算有何用途？

答案：三角估算通过确保每个估算相对其他估算都是合理的来优化估算。如果一个 2 点的故事看起来是一个 1 点的故事的 2 倍，它也应当看起来是一个 4 点的故事的一半。

8.3 定义速率。

答案：速率是一个团队在一轮迭代中完成的故事点数。

8.4 团队 A 在上个 2 星期的迭代中完成了 43 个故事点。团队 B 在作另一个项目而且有两倍的开发人员。他们同样在上个 2 星期的迭代中也完成了 43 个故事点。为什么会这样？

答案：一个团队的故事点数和任何其他团队的故事点数没有可比性。从这个问题的信息中，我们不能认为团队 A 的效率是团队 B 的两倍。

第 9 章 发布计划

9.1 估算团队初始速率的三种方法是什么？

答案：可以用历史值，作一个猜测，或者试着做一轮迭代，用这轮迭代的速率。

9.2 假设迭代以 1 周为长度，团队里有 4 位开发人员，如果团队的速率是 4，项目总共有 27 个故事点，完成项目需要多少轮迭代？

答案：速率为 4，项目有 27 个故事点，团队需要 7 轮迭代来完成。

第 10 章 迭代计划

10.1 从这个故事中分解出任务：用户可以查看酒店的相关详细信息。

答案：当然有很多方法来分解任务，下面是其中一种。

- 设计这些网页的外观。
- 编写 HTML 显示酒店和房间照片。
- 编写 HTML 显示一个告知酒店位置的地图。
- 编写 HTML 显示一个酒店设施和服务的清单。
- 研究如何生成地图。
- 编写 SQL 从数据库获取信息。
- 等等。

第 11 章 测量并监控速率

11.1 一个故事估算为 1 个故事点，实际花了 2 天来完成。在迭代末计算速率时应该计入多少呢？

答案：计入 1 个点到速率中。

11.2 哪些信息可以从每日燃尽图中得到，却不能从迭代燃尽图中得到？

答案：每日燃尽图显示了团队在某轮迭代中的进度。你可以用这个信息判断计划的工作能否在迭代结束时都能完成。如果明显看出不可能完成所有的工作，团队和客户应该在迭代中进行沟通，看看应该推迟哪些工作。

11.3 你从图 11.7 中能得出什么结论？像这样的项目是会提前完成，延期完成，还是按时完成？

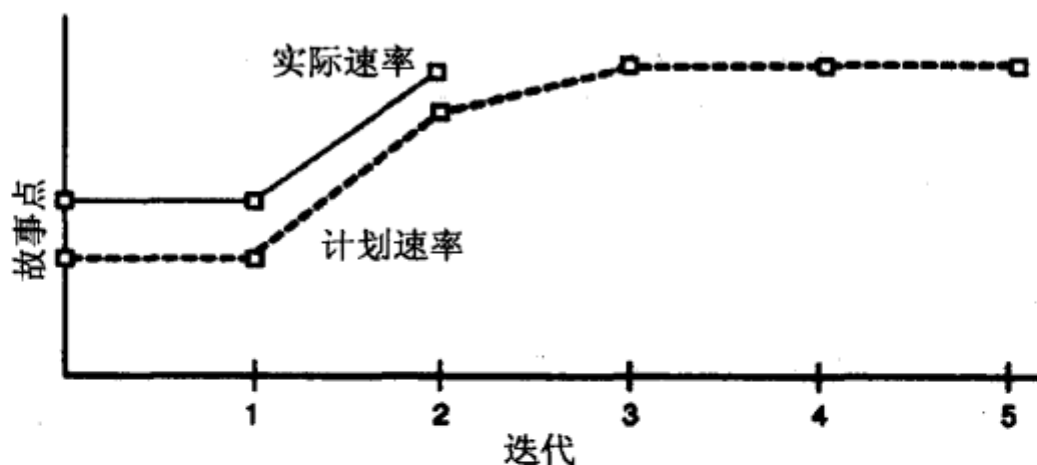


图 11.7 这个项目是会提前完成，延期完成，还是按时完成？

答案：这个团队从第一轮迭代开始比预期的要好。他们期望速率在第二和第三轮迭代提高速率，然后保持这个速率。在两轮迭代后，他们已经达到了他们三轮迭代后预期的速率。此时他们已经领先于计划，但是你不应在仅仅过了两轮迭代就下太多结论。

11.4 如表 11.3 所示的迭代，团队的速率是多少？

表 11.3 在迭代中完成的故事

故 事	故 事 点	状 态
故事 1	4	完成
故事 2	3	完成
故事 3	5	完成
故事 4	3	部分完成
故事 5	2	完成
故事 6	4	还没开始
故事 7	2	完成
速率	23	

答案：16。部分完成的故事不能计入速率中。

11.5 在哪些情况下迭代燃尽图会有向上的趋势？

答案：如果加新任务的速度比完成任务的速度要快，或者团队觉得有大量的任务被低估了，迭代燃尽图就会有向上的趋势。

11.6 填写表中空缺的值，完成表 11.4。

表 11.4. 填写空缺的值

	迭代 1	迭代 2	迭代 3
迭代开始时故事点	100		
在迭代中完成的	35	40	36
改变的估算	5	-5	0
新加故事的故事点	6	3	
迭代结束时故事点	76		0

答案：完成的表如表 B.4 所示。

表 B.4 问题 11.6 的答案

	迭代 1	迭代 2	迭代 3
迭代开始时故事点	100	76	34
在迭代中完成的	35	40	36
改变的估算	5	-5	0
新加故事的故事点	6	3	2
迭代结束时故事点	76	34	0

第 12 章 故事不是什么

12.1 用户故事和用例的主要区别是什么？

答案：通常情况下一个用户故事比用例规模小。用户故事不包括像用例那么多细节。用户故事只在开发用户故事的当前迭代有用；用例常常用作项目的永久性的工件。

12.2 用户故事和 IEEE 830 需求声明的主要区别是什么？

答案：IEEE 830 需求声明关心解决方案的特征，用户故事关心用户的目标。IEEE 830 需求声明鼓励团队在之前完成所有的需求声明，而不是像用户故事那样使用迭代的方式。写需求声明需要花很大精力来确保文字表达了正确的意思；用户故事鼓励通过口头交流澄清细节。

12.3 用户故事与交互式设计场景的主要区别是什么？

答案：交互式设计场景比用户故事要具体的多，经常很具体地描述虚拟人物和系统使用的背景。而且，一个场景通常描述的范围比一个用户故事要大。

- 12.4 对于一个重要的项目，为什么不能在项目启动时编写所有需求？

答案：在项目开始时尝试编写所有的需求忽略了重要的反馈循环。当系统的用户开始看到并与系统交互时，新的需求就产生了。

- 12.5 与琢磨待开发软件的特性列表相比，考虑用户的目标有哪些优势？

答案：特征列表给读者对产品的整体理解与用户故事和交流不一样。而且，如果我们的工作都是通过一个产品特性列表驱动的，当我们做到最好时就是交付了一个拥有列表上特性的产品。这和交付的产品达到所有用户目标是不一样的。

第 13 章 用户故事的优势

- 13.1 列举使用用户故事描述需求的 4 大优势。

答案：用户故事强调口头交流，大家都能理解，大小适合做计划，支持迭代开发，鼓励延迟细节，支持随机应变的设计，鼓励参与性设计，传播隐性知识。

- 13.2 列举使用用户故事的两个不足之处。

答案：在大型项目中组织成百上千的故事是困难的；故事需要额外的文档增强可追溯性；通过面对面的交流加强传播隐性知识的同时，在大型项目对话不能完全替代文档。

- 13.3 参与性设计与经验性设计之间有哪些主要区别？

答案：参与性设计中，系统的用户成为团队的一部分一起设计系统的行为。经验性设计中，软件的设计师学习或观察用户，然后做出所有的设计决定。

- 13.4 “所有多页的报表应该被编号” 这句需求语句有什么问题？

答案：“应该被编号”的意义不清楚。这是指明程序员要应该编写代码实现这个功能，但不是必需的？还是说如果页面有空间的时候页面应该被编号？

第 14 章 用户故事不良症兆一览

- 14.1 如果团队总是发现很难做下一轮迭代的计划，你应该怎么办？

答案：你应考虑是否太多故事是独立的，或太小，或太大，也可能是其他的原因。

- 14.2 如果团队总是觉得小卡片太小写不下用户故事，你应该怎么办？

答案：他们应该用更小的卡片来确保不在故事描述中加入细节。

- 14.3 客户在哪些情况下会觉得很难为用户故事排优先级？

答案：故事可能大小不对(可能太大或太小)，或者故事没有清晰的表述对用户或客户的价值。

14.4 在什么情况下，你会觉得划分的故事太多？

答案：你不得不依赖你的直觉。故事经常合理地被分解，因为他们开始都写成史诗故事，或者因为他们太大而不能放入一轮迭代中。如果你发现你经常因为其他的原因分解故事，你可能做得太多了。

第 15 章 Scrum 与用户故事

15.1 描述递增过程与迭代过程的区别。

答案：迭代过程是通过持续改进来取得进展。递增过程是团队按功能点(in pieces)开发和发布软件。

15.2 产品 Backlog 和 Sprint Backlog 有何关联？

答案：在 Sprint 开始的时候 Scrum 团队从产品 Backlog 中把足够的项目放到 Sprint Backlog 上，下一轮迭代开发实现。

15.3 什么是潜在可交付的产品功能增量？

答案：在每个 Sprint 最后，Scrum 团队负责产出潜在可交运的产品功能增量。即软件已经被完成代码，测试并且可以交付给用户。

15.4 谁负责排优先级？谁负责替团队选择下一个 Sprint 的任务？

答案：产品负责人负责派优先级，但是团队选择下一个 Sprint 的任务。当然他们应该选择最高优先级的任务。

15.6 在每日 Scrum 简会中，每个成员都要回答哪些问题？

答案：你昨天做了什么？你今天打算做什么？有什么困难？

第 16 章 其他话题

16.1 如何处理让系统支持 1000 个并发用户的扩充性需求？

答案：应该将这个需求作为限制条件写下来，然后为其添加适当的测试。根据不同的系统，可以先在一轮迭代中从 100 个并发用户的测试，然后在若干轮迭代中逐步增加到 1000 个用户。

16.2 你喜欢在笔记卡还是在软件系统里编写故事？请解释一下你的答案。

答案：笔记卡技术含量低，简单，这使得它们适合许多项目。卡片同时提供有限的空间，帮助我们精简故事。因为卡片可以很容易在桌上或墙上移来移去，很适合做计划。当然，如果团队不在一起或有严格的可追溯性需求，使



用软件可能更合适。

16.3 迭代过程对应用程序的用户界面有何影响？

答案：系统迭代改进可能让用户感到学习系统很困难。当菜单系统改变或功能出现在不同的地方，用户必须重新学习系统。

16.4 给出一些例子，说明在系统前期考虑用户界面比典型敏捷项目的做法具有更多好处。

答案：有许多的例子，下面是一些参考：

- 一个商业产品在一个成熟行业中通过易用性来竞争
- 软件目标用户是初学者
- 软件不经常用，只是在某些特定阶段大量使用(比如申报所得税的时候)
- 软件为视力不好的用户或行动不便的用户设计

16.5 一旦开发完故事，你是建议保留还是销毁故事？请解释一下。

答案：略。





参 考

书 和 文 章

- Adolph, Steve, Paul Bramble, et al. *Patterns for Effective Use Cases*. Reading, Mass.: Addison-Wesley, 2002.
- Antón, Annie I., and Colin Potts. "The Use of Goals to Surface Requirements for Evolving Systems," in *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, April 1998: 157–166.
- Astels, Dave. *Test Driven Development: A practical guide*. Upper Saddle River, N.J.: Prentice Hall, 2003.
- Beck, Kent. *Extreme Programming Explained: Embrace change*. Boston: Addison-Wesley, 2000.
- Kent Beck. *Test Driven Development*. Reading, Mass.: Addison-Wesley, 2003.
- Beck, Kent, and Martin Fowler. *Planning Extreme Programming*. Reading, Mass.: Addison-Wesley, 2000.
- Beedle, Mike, et al. "SCRUM: A Pattern Language for Hyperproductive Software Development." In Neil Harrison et al. (Eds.), *Pattern Languages of Program Design 4*. Addison-Wesley: 1999, pp. 637–651.
- Boehm, Barry. "A Spiral Model of Development and Enhancement." *IEEE Computer* 28, no. 5 (May 1988): 61–72.
- Boehm, Barry. *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.
- Bower, G. H., J. B. Black, and T. J. Turner. "Scripts in Memory for Text." *Cognitive Psychology* 11 (1979): 177–220.
- Carroll, John M. "Making Use a Design Representation." *Communications of the ACM* 37, no. 12 (December 1994): 29–35.
- . *Making Use: Scenario-based design in human-computer interaction*. Cambridge, Mass.: The MIT Press, 2000.
- . "Making use is more than a matter of task analysis." *Interacting with Computers* 14, no. 5 (2002): 619–627.
- Carroll, John M., Mary Beth Rosson, George Chin Jr., and Jürgen Koenemann. "Requirements Development in Scenario-Based Design." *IEEE Transactions on Software Engineering* 24, no. 12 (December 1998): 1156–1170.
- Cirillo, Francesco. "XP: Delivering the Competitive Edge in the Post-Internet Era." At www.communications.xplabs.com/paper2001-3.html. XP Labs, 2001.
- Cockburn, Alistair. *Writing Effective Use Cases*. Upper Saddle River, N.J.: Addison-Wesley, 2001.

- Cockburn, Alistair, and Laurie L. Williams. "The Costs and Benefits of Pair Programming." In Giancarlo Succi and Michele Marchesi (Eds.), *Extreme Programming Examined*. Upper Saddle River, N.J.: Addison-Wesley, 2001.
- Cohn, Mike. "The Upside of Downsizing." *Software Test and Quality Engineering* 5, no. 1 (January 2003): 18–21.
- Constantine, Larry. "Cutting Corners." *Software Development* (February 2000).
- . "Process Agility and Software Usability: Toward lightweight and usage-centered design." *Information Age* (August–September 2002).
- Constantine, Larry L., and Lucy A.D. Lockwood. *Software for Use: A practical guide to the models and methods of usage-centered design*. Reading, Mass.: Addison-Wesley, 1999.
- . "Usage-Centered Engineering for Web Applications." *IEEE Software* 19, no. 2 (March/April 2002): 42–50.
- Cooper, Alan. *The Inmates Are Running the Asylum*. Indianapolis: SAMS, 1999.
- Cusumano, Michael A., and Richard W. Selby. *Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets, and manages people*. New York: The Free Press, 1995.
- Davies, Rachel. "The Power of Stories." XP 2001. Sardinia, 2001.
- Djajadiningrat, J.P., W. W. Gaver and J. W. Frens. "Interaction Relabelling and Extreme Characters: Methods for exploring aesthetic interactions." *Symposium on Designing Interactive Systems 2000*, 2000: 66–71.
- Fowler, Martin. "The Almighty Thud." *Distributed Computing* (November 1997).
- Fowler, Martin, et al. *Refactoring: Improving the design of existing code*. Reading, Mass.: Addison-Wesley, 1999.
- Gilb, Tom. *Principles of Software Engineering Management*. Reading, Mass.: Addison-Wesley, 1988.
- Guindon, Raymonde. "Designing the Design Process: Exploiting opportunistic thoughts." *Human-Computer Interaction* 5, 1990.
- Grudin, Jonathan, and John Pruitt. "Personas, Participatory Design and Product Development: An Infrastructure for Engagement." In Thomas Binder, Judith Gregory, and Ina Wagner (Eds.), *Participation and Design: Inquiring into the politics, contexts and practices of collaborative design work, Proceedings of the Participatory Design Conference 2002*: 2002: 144–161.
- IEEE Computer Society. *IEEE Recommended Practice for Software Requirements Specifications*. New York, 1998.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Upper Saddle River, N.J.: Addison-Wesley, 1992.
- Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.
- Jeffries, Ron. "Essential XP: Card, Conversation, and Confirmation." *XP Magazine* (August 30, 2001).

- Jeffries, Ron, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Boston: Addison-Wesley, 2000.
- Kensing, Finn, and Andreas Munk-Madsen. "PD: Structure in the Toolbox." *Communications of the ACM* 36, no. 6 (June 1993): 78–85.
- Kovitz, Ben L. *Practical Software Requirements: A manual of content and style*. Greenwich, Conn.: Manning, 1999.
- Kuhn, Sarah, and Michael J. Muller. "Introduction to the Special Section on Participatory Design." *Communications of the ACM* 36, no. 6 (June 1993): 24–28.
- Lauesen, Soren. *Software Requirements: Styles and techniques*. London: Addison-Wesley, 2002.
- Lundh, Erik, and Martin Sandberg. "Time Constrained Requirements Engineering with Extreme Programming: An experience report." In Armin Eberlein and Julio Cesar Sampaio do Prado Leite (Eds.), *Proceedings of the International Workshop on Time Constrained Requirements Engineering*, 2002.
- Newkirk, James, and Robert C. Martin. *Extreme Programming in Practice*. Upper Saddle River, N.J.: Addison-Wesley, 2001.
- Parnas, David L., and Paul C. Clements. "A Rational Design Process: How and why to fake it." *IEEE Transactions on Software Engineering* 12, no. 2 (February 1986): 251–7.
- Patton, Jeff. "Hitting the Target: Adding interaction design to agile software development." Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2002). New York: ACM Press, 2002.
- Poppendieck, Tom. *The Agile Customer's Toolkit*. In Larry L. Constantine (Ed.), *Proceedings of forUSE 2003*. Rowley, Mass.: Ampersand Press: 2003.
- Potts, Colin, Kenji Takahashi, and Annie I. Antón. "Inquiry-Based Requirements Analysis." *IEEE Software* 11, no. 2 (March/April 1994): 21–32.
- Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*. Reading, Mass.: Addison-Wesley, 1999.
- Schuler, Douglas, and Aki Namioka (Eds.). *Participatory Design: Principles and practices*. Hillsdale, N.J.: Erlbaum, 1993.
- Schwaber, Ken, and Mike Beedle. *Agile Software Development with Scrum*. Upper Saddle River, N.J.: Prentice Hall, 2002.
- Stapleton, Jennifer. *DSDM: Business Focused Development*. Reading, Mass.: Addison-Wesley, 2003.
- Swartout, William, and Robert Balzer. "On the Inevitable Intertwining of Specification and Implementation." *Communications of the ACM* 25, no. 7 (July 1982): 438–440.
- Wagner, Larry. "Extreme Requirements Engineering." *Cutter IT Journal* 14, no. 12 (December 2001).
- Wake, William C. *Extreme Programming Explored*. Reading, Mass: Addison-Wesley, 2002.



- . "INVEST in Good Stories, and SMART Tasks." At www.xp123.com, 2003a. Wake, William C..
Refactoring Workbook. Reading, Mass.: Addison-Wesley, 2003b.
- Weidenhaupt, Klaus, Klaus Pohl, Matthias Jarke, and Peter Haumer. "Scenarios in System Development: Current practice." *IEEE Software* 15, no. 2 (March/April 1998): 34–45.
- Wiegers, Karl E. *Software Requirements*. Redmond, Wash.: Microsoft Press, 1999.
- Williams, Marian G., and Vivienne Begg. "Translation between software designers and users." *Communications of the ACM* 36, no. 6 (June 1993): 102–3.

网 站

www.agilealliance.com
www.controlchaos.com
www.foruse.com
www.mountangoatsoftware.com
www.userstories.com
www.xprogramming.com
www.xp123.com





关于著译者和审校者

关于作者

Mike Cohn 是敏捷联盟的发起成员之一，并担任其文章项目的总监。他 1984 年开始编程，1988 年开始管理软件项目，客户包括富达投资、维亚康姆、宝洁、NBC 和花旗银行。Mike 写本书时是 Fast401k 的软件工程副总裁。这家行业领先公司提供基于互联网的 401(k) 档案保存和管理解决方案。Fast401k 向金融服务行业客户提供自主品牌的 e401k 软件产品，作为外包服务供应商，利用专有技术实现规模经济效应。在本书之前，Mike 著有或合写了 4 本编程方面的书籍。(译者注：Mike 也是《敏捷估计及估算》及 *Succeeding with Agile* 两本重要敏捷著作的作者，并与其他两位敏捷泰斗 Ken Schwaber 和 Esther Derby 一起创办了 Scrum 联盟。)

关于译者和审校者

Jackson

张博超(Jackson Zhang), Irdeto BSS 软件开发工程师, CSM, CSP, 敏捷爱好者。Jackson 从事软件开发 5 年，现主要做 .NET 开发。关注敏捷开发，积极实践各种敏捷方法。曾作为团队 ScrumMaster，帮助团队实践 Scrum，在团队中推行敏捷思想。他在团队中作为 Leader Engineer，引导团队开发和实践 TDD 与结对编程等敏捷方法。个人博客：<http://www.neodream.info/blog>。

Stone

石永超(Stone Shi), Irdeto BSS 软件开发工程师, CSM, 敏捷爱好者。从业 5 年，目前主要做 .NET 开发。曾在 Infosys 参与 CMMI 体系的软件开发方法，认识到了传统软件过程的一些问题。后进入 Irdeto，开始关注并实践敏捷方法近 3 年时间，并从中受益。作为团队 Leader Engineer，帮助团队一起实践高效、简单、高质量的软件开发方法。个人博客：<http://blog.csdn.net/zerostone>。



Daniel

滕振宇 (Daniel Teng), Irdeto BSS 高级软件经理, CSP, InfoQ 中文站特约编辑。个人博客: www.cnblogs.com/tengzy。Daniel 有超过 6 年的 Scrum 和 XP 经验, 创建了 Irdeto BSS 上海研发中心, 并把 Scrum 和 XP 成功引入了团队。目前该团队主要负责大型付费媒体计费及客户关系管理系统的开发和维护。Daniel 一直致力于将敏捷理论及方法介绍到国内, 帮助国内的软件团队有效并有趣地工作, 真正为客户创造价值。Daniel 是敏捷中国 2009 的讲师, 并受邀在 CSDN 举办的软件开发 2.0 大会、杭州 Scrum 论坛、Scrum 中文网、Scrum 沙龙等会议发表关于敏捷的演讲。

Bill

李国彪 (Bill Li), 敏捷及 Scrum 培训师与推广者, 教练及组织转型顾问, Uperform(优普丰)顾问机构的 Managing Principal 和创办者, 致力于在中国推广有效的敏捷与 Scrum 思想与实践。Bill 有超过 16 年的海内外跨职能 IT 行业经验。在创立 UPerform 之前, 他是某知名项目管理咨询组织的区域总经理。早于 1996 年, 他通过 Unisys 公司为中国移动领导和管理国内首个商业中文短信平台的实施。他近几年翻译或合作翻译了 Ken Schwaber 的两本著作《Scrum 敏捷项目管理》和《Scrum 敏捷项目管理实战》, 以及另外两本传统项目管理方面的著作。Bill 是加拿大约克大学 Schulich 商学院的 MBA, 也是 CSPO、CSM 及 CSP。其公司网站: www.UPerform.CN。



[G e n e r a l I n f o r m a t i o n]

书名=用户故事与敏捷方法

作者=(美)科恩著

页数=220

出版社=北京市：清华大学出版社

出版日期=2010.04

SS号=12636857

DX号=000006876195

URL=<http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000006876195&d=D1CCA24E66007EC66B27D6FF63F38B26>

封面
书名
版权
前言
目录

第 部分 起步

第 1 章 概览

什么是用户故事？

细节在哪里？

“必须多长时间完成？”

客户团队

使用故事的过程是怎么样的？

规划发布和迭代

什么是验收测试？

为什么要变？

小结

问题

第 2 章 编写故事

独立的

可讨论的

对用户或客户有价值的

可估计的

小的

分割故事

合并故事

可测试的

小结

开发人员职责

客户团队职责

问题

第 3 章 用户角色建模

用户角色

角色建模的步骤

通过头脑风暴，列出初始的用户角色集合

整理最初的角色集合

整合角色

提炼角色

两个额外的技术

虚构人物

极端人物

如果有现场用户该如何？

小结

开发人员职责

客户职责

问题

第 4 章 搜集故事

引出和捕捉是不合用的

够用就行，不是吗？

方法

用户访谈

问卷调查

观察

故事编写工作坊

小结

开发人员职责

客户职责

问题

第 5 章 与用户代理合作

用户的经理

开发经理

销售人员

领域专家

市场营销团队

以前的用户

客户

培训师和技术支持

业务分析师或系统分析师

与用户代理合作时，做些什么？

能接触到用户但访问受限时

实在不能接触到用户时

可以自己来吗？

设立客户团队

小结

开发人员职责

客户团队职责

问题

第 6 章 用户故事验收测试

在写代码之前写测试

客户定义测试

测试是过程的一部分

多少测试才算多？

集成测试框架

测试类型

小结

开发人员职责

客户职责

问题

第 7 章 优秀用户故事准则

从目标故事开始

切蛋糕

编写封闭的故事

卡片约束

根据实现时间来确定故事规模

不要过早涉及用户界面

有些需求并不是故事
在故事里包括用户角色
只为一个用户编写
以主动语态编写
由客户编写
向故事卡编号说“不”
不要忘记意图
小结
问题

第 部分 估算和计划

第 8 章 估算用户故事 故事点

以团队估算

估算

三角测量
使用故事点
如果用结对编程呢？
一些提醒
小结
开发人员职责
客户职责
问题

第 9 章 发布计划

我们想在什么时候发布
希望在发布中包含哪些功能？
排列故事优先级
混合优先级
高风险故事
根据架构需要安排优先级
选择迭代长度
从故事点到预计工期
初始速率
猜测速率
创建发布计划
小结
开发人员职责
客户职责
问题

第 10 章 迭代计划

迭代计划概览
讨论故事
分解任务
准则
承担职责
估算并确认
小结
开发人员职责

客户职责
问题

第 1 1 章 测量并监控速率

测量速率
计划速率和实际速率
迭代燃尽图
迭代中的燃尽图
小结
开发人员职责
客户职责
问题

第 部分 经常讨论的话题

第 1 2 章 故事不是什么

用户故事不是 I E E E 8 3 0
用户故事不是用例
用户故事不是场景
小结
问题

第 1 3 章 用户故事的优势

口头沟通
用户故事容易理解
用户故事的大小适合做计划
用户故事适合于迭代开发
用户故事鼓励延迟细节
用户故事支持随机应变的开发
用户故事鼓励参与性设计
用户故事传播隐性知识
用户故事的不足
小结
开发人员职责
客户职责
问题

第 1 4 章 用户故事不良症兆一览

故事太小
故事互相依赖
镀金
细节太多
过早考虑用户界面细节
想得太远
故事划分太过频繁
客户很难为故事安排优先级
客户不愿意写用户故事，也不愿意为故事安排优先级
小结
开发人员职责
客户职责
问题

第 1 5 章 S c r u m 与用户故事

- S c r u m是迭代和递增的
- S c r u m基础
- S c r u m团队
- 产品B a c k l o g
- S p r i n t计划会议
- S p r i n t评审会议
- 每日S c r u m简会
- 在S c r u m中使用用户故事
- S c r u m和产品B a c k l o g
- 在S p r i n t计划会议中使用用户故事
- 在S p r i n t评审会议中使用用户故事
- 在每日S c r u m简会中使用用户故事
- 一个案例
- 小结
- 问题

第 1 6 章 其他话题

- 处理非功能性需求
- 纸质还是软件？
- 用户故事和用户界面
- 保留故事
- 缺陷的用户故事
- 小结
- 开发人员职责
- 客户职责
- 问题

第 部分 一个完整的实例

第 1 7 章 用户角色

- 项目
- 定义客户
- 定义一些角色雏形
- 整合与提炼
- 角色建模
- 添加虚构人物

第 1 8 章 一些用户故事

- T e r e s a的故事
- R o n船长的故事
- “初级航海者”的故事
- “不出海的礼物购买者”的故事
- “报表查阅者”的故事
- “管理员”的一些故事
- 收尾

第 1 9 章 估算故事

- 第一个故事
- 高级搜索
- 评分和评论
- 账户
- 完成估算

	所有估算
第 2 0 章	发布计划
	估算速率
	给故事安排优先级
	最终的发布计划
第 2 1 章	验收测试
	搜索测试
	购物车测试
	购买书
	用户账户
	管理
	测试限制条件
	最后一个故事
第 部分	附录
附录 A	极限编程概览
附录 B	参考答案