

需要整本电子书，联系我QQ: [2667271557](#);
此处是样章，取的完整版的前面几页，和最后
面几页；完整版是带书签的，样章没带书签；
另外需要其他书，也可以找我。

C#

敏捷开发实践

【英】Gary McLean Hall 著 许顺强 译

拥抱敏捷，编写自适应代码
轻松应对恼人的需求变更

Adaptive Code via C#

Agile coding with design patterns and SOLID principles



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS



图灵程序设计丛书

C# 敏捷开发实践

【英】Gary McLean Hall 著 许顺强 译



Adaptive Code via C#

Agile coding with design patterns and SOLID principles

人民邮电出版社

北 京

图书在版编目 (C I P) 数据

C#敏捷开发实践 / (英) 加里·麦克莱恩·霍尔
(Gary McLean Hall) 著 ; 许顺强译. — 北京 : 人民邮
电出版社, 2016. 7
(图灵程序设计丛书)
ISBN 978-7-115-42789-2

I. ①C… II. ①加… ②许… III. ①C语言—程序设
计 IV. ①TP312

中国版本图书馆CIP数据核字 (2016) 第139459号

内 容 提 要

本书共分为敏捷基础、编写 SOLID 代码和自适应实例三大部分, 将理论与实践相结合, 介绍了当前使用 Microsoft .NET Framework 进行 C# 编程的最佳实践, 详尽探讨了 C# 开发人员如何应用 Scrum 等敏捷方案实现高质量、自适应的代码, 并给出大量代码示例, 是 .NET 中高级程序员进阶的实用指南。

本书的读者对象为有一定经验的 .NET 开发人员。

-
- ◆ 著 [英] Gary McLean Hall
译 许顺强
责任编辑 朱 巍
执行编辑 杨 琳 赵瑞琳
责任印制 彭志环
- ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷
- ◆ 开本: 800×1000 1/16
印张: 22
字数: 533千字 2016年7月第1版
印数: 1-3 000册 2016年7月北京第1次印刷
- 著作权合同登记号 图字: 01-2015-2389号
-

定价: 69.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广字第 8052 号

谨以此书献给 Amelia Rose。

译者序

翻译这本书算是圆了我学生时期的一个心愿：在一本纸质书的封面上印上自己的名字。作为一名 IT 技术人员，总是感觉自己的时间不够用，因为技术永无止境，让我从不敢懈怠。以前阅读过许多“大侠”翻译的不少书籍，但对于译者的辛酸和困难都没有太大的感触。直到自己开始实际动手翻译时，才发现整个过程中心情都是忐忑不安的，担心自己的翻译不能清楚传达原书作者的真正意图，害怕自己的译文会让读者觉得乏味，于是，总是会对不满意的翻译片段心生焦虑，也尽了自己最大努力对译文的选词和顺序进行反复的推敲和斟酌。当然，由于自身能力和精力有限，相信大家肯定会找出翻译中需要改进的地方，在此先行感谢大家的热心指正。

原书内容的精彩我就不多赘述。本书主要针对 C#程序员，基于敏捷方法论，介绍使用 Microsoft .NET Framework 进行 C# 编程的当前最佳实践，其中包括了从敏捷项目过程到代码编写的理论和实践的详细讲解。对于那些需要实操指导的读者来说，几乎全部内容都可以直接应用在实际敏捷项目的管理和编码活动当中。如果你是一名初学者，可以在本书中学习使用 C# 进行敏捷开发的常见模式和实践，明辨其优劣，让自己走在正确的方向上，为后续的能力提升打下良好的基础。如果你是一名中级开发人员，可以在本书中学习业界的最佳实践，了解各种实践组合，对 SOLID 原则获得深入的理解，并完全认识到其在实际代码开发中带来的益处。如果你是一名高级开发人员，毫无疑问，你将获益最多。本书提供了大量设计模式、SOLID 原则、单元测试、重构等理论的示例，将理论与实践关联起来，让你可以直接拿来应用于工作之中。

当然，尽信书不如无书，相信有读者会对书中所讲并不完全赞同，但表达意见的前提是首先要理解书中讲解的本意。技术皆有优劣，作为 IT 技术人员，切忌用个人主观情绪来表达对技术观点的不满。我有时会听到周围有人说：“我就是看着不爽！”他们习惯了一刀切，也不明白一句古话：“三人行，必有我师焉。”原书作者 Gary 已经在全书技术理论和实例讲解过程中很好地穿插了优缺点和应用场景的讨论。认真读完一本技术书的收获应该是：明悉技术的概念和原理，了解其优缺点，并且知道其适用场合；如果还能在实践中针对缺点提出改进，那就再好不过了。

许顺强

2016 年 3 月 21 日

前言

本书英文书名中首先提到的一个关键术语是自适应代码（Adaptive Code），这一关键术语很好地诠释了应用本书中介绍的基本原则能够达成的效果：无需大量返工，代码即可自动适应后续新的需求和无法预见的场景。本书旨在将使用 Microsoft .NET Framework 进行 C# 编程的当前最佳实践囊括于一卷之中。尽管其他书中也会涵盖本书中的某些内容，但这些书要么偏重于讲解理论，要么就不是特定于 .NET Framework 开发的。

编写代码不能急于求成。与阻碍变化的代码库相比，如果你的代码具有自适应能力，你能够更加快速、轻松地对其进行更改，并且不会引发多少错误。相信大家都知道，对于需求，不变的主题就是变化。因此管理需求变更是软件项目成败的一个关键因素。面对需求变更，开发人员可以有很多种应对方法，这些方法可以归类到下面要讲述的两种方法论。这两种方法论的主旨几乎是截然相反的，特别是在变更处理的连续性上。

第一种是瀑布方法论。这种方法论要求开发人员必须遵循严格的流程。应用这种方法论的项目中，大到要遵循的开发流程，小到要实现的设计都很不灵活，甚至几乎与 50 年前用穿孔卡片进行编程开发一样死板。所有瀑布方法论都在竭尽全力确保软件很难被自由更改。软件开发被划分为分析、设计、实现和测试几个显著不同的阶段，而且整个过程是单向的。一旦进入实现阶段，用户就很难去变更需求，或者说至少要付出昂贵的代价才能变更需求。当然，代码也无需为需求变更作任何准备，因为整个瀑布流程几乎不提供任何其他选项。

第二种是敏捷方法论。它不仅仅是另一种选择，而且是对瀑布方法论的一种彻底推翻。敏捷流程的主旨就是拥抱变化，它被看作是客户和开发者之间的一个必要的联系纽带。如果客户想要对自己付费的产品进行某些改变，就应该把时间和资金的代价与需求的变更关联起来，而不是直接把变更加入流程中正在进行的阶段。软件工程的基础是源代码，相对于物理工程，它具有更好的可塑性。建造一座房子的过程就是使用水泥把砖逐块粘合在一起，所以改变房子设计的代价就很自然地与房子的建造完成度直接相关了。假设工程尚未开始，只有设计蓝图，那么变更设计的代价相对来说就会很低。如果已经装好了窗户，布好了电线、管线，此时再想把楼上的浴室改到楼下的厨房旁，那代价就会非常高了。软件产品的源代码具有良好的可塑性，因此移动特性和修改用户界面的导航看起来不应该有很高的代价，但不幸的是，事实并不总是如此。单单时间成本就经常不允许在软件产品中进行这样自由的变更。在我看来，这主要就是因为代码缺乏对需求变更的自适应能力。

本书将通过一些实际的例子，为大家演示和讲解敏捷流程以及如何编写自适应代码。

本书面向的读者

本书的意图是要把理论与实践关联起来。如果你是经验丰富的高级开发人员，想要找一些设计模式、SOLID 原则、单元测试、重构等理论的示例，那么这本书就是为你而作。

如果你是具有一定经验和能力的中级开发人员，想要学习业界的最佳实践，了解它们是如何配合使用的，或者你对现在的业界最佳实践组合有疑问，都可以从这本书中获益，因为现实的项目开发中很难找到简单且容易理解的实例或者理论。开发人员对大多数 SOLID 原则已经有所了解，但是对于其中比较复杂的开放与封闭原则（第 6 章）和 Liskov 替换原则（第 7 章）的理解还不够充分。即使是经验丰富的开发人员，有时也无法完全认识到依赖注入（第 9 章）给代码开发带来的好处。与此类似，接口（第 3 章）能给代码带来的适配灵活性也经常被忽视。

如果你是刚刚入门的初级开发人员，读完这本书，也会有所收获。你可以学习到常见的模式和实践，并且知道哪些方面是好的，哪些方面从长期来看是不好的。我见到的软件开发实习生所写的代码有很多共同点，到处都可以看到随从反模式（第 2 章）和服务定位器反模式（第 9 章）的代码。通常，实习生已经具备了很多方面的软件开发技能，要把他变成一个重量级的开发人员，只需要在正确的方向上推他们一把即可。本书也提供了多种可选的实践方案，并对其优缺点进行了详细解释。

阅读本书的前提条件

要阅读这本书，你应该具备一些在语法上与 C# 类似的编程语言（比如 Java 或 C++）的实战经验，也应该精通条件分支、循环和表达式等核心过程编程概念。此外，还应该有使用类进行面向对象开发的经验，并且对接口的概念有所了解。

本书不适合哪些人

如果你刚刚开始学习编程开发，那么本书并不适合你，因为书中涉及一些高级开发话题，需要你对基本的编程开发概念有深入的理解。

本书结构

本书共分为三个部分，每一部分都以上个部分为基础。尽管如此，也可以从任何一个部分开始阅读本书。每个章节都详细讲解了一个完整的主题，并在适当的地方包括了指向其他章节的交叉索引。

第一部分 敏捷基础

这个部分会讲解如何以自适应方式开发软件的基础概念，其中包括业界有名的敏捷流程 Scrum，该流程要求代码具有自适应变更的能力。这一部分的所有章节都围绕接口、设计模式、

重构和单元测试进行详细的讲解。

❑ 第 1 章 Scrum 介绍

这一章是本书的开篇，首先介绍一种业界知名的敏捷项目管理方法论 Scrum，然后详细介绍 Scrum 项目中相关的工件、角色、度量标准和阶段的概念，最后为大家展示如何在敏捷环境下组织资源和代码。

❑ 第 2 章 依赖和分层

这一章将引领你一起探索依赖和架构分层。代码要做到自适应，前提是解决方案的结构允许这样做。首先讲解三种不同类型的依赖：第一方、第三方和框架；然后讲解如何从反模式（应该避免的）到模式（应该使用的）来管理和组织所有的依赖关系；最后会介绍一些高级主题供你进一步阅读，比如面向切面编程和非对称分层等。

❑ 第 3 章 接口和设计模式

在现代.NET 应用的开发中，接口几乎无处不在，但是它们也经常被滥用、误解和错用。这一章将首先通过一些常见和实用的设计模式来展示接口的多种用途；然后阐明除了使用接口进行简单的抽象外，还能够以多种不同的方式使用接口来解决同一个问题。如果能够流利地使用开发者武器库中的混合类型、鸭子类型和流接口，将更能体会到接口的强大用途。

❑ 第 4 章 单元测试和重构

单元测试和重构正在成为开发人员必备的两个实战技能，只有同时应用这两个技能，才能编写出自适应代码。没有完备的单元测试，重构动作肯定会造成很多错误；没有重构，代码则会变得臃肿、僵化且难以理解。这一章会从一个十分简单的单元测试示例开始讲解，然后扩展讲解更高级、实用的模式和实践，比如流断言、测试驱动开发和模拟。本章还提供了真实的重构示例来讲解如何改善源代码的可读性和可维护性。

第二部分 编写 SOLID 代码

这一部分以第一部分为基础，每一章都会专门讲解 SOLID 中的一个原则。这些章节不会单单讲解为什么要使用这些原则，还有详细的实战示例来讲解如何在编码实现中使用这些原则。每一章都会提供一个实际项目中的例子来展示 SOLID 原则的作用。

❑ 第 5 章 单一职责原则

这一章会为开发人员展示如何使用修饰器和适配器模式来实践单一职责原则。应用这个原则后，类的数目会增加，但是每个类的规模会变小。相对于那些功能繁多的大型类，小型类可集中解决一个大型问题中的一小部分。聚合使用这些小型类会比使用单个大型类更强大、更灵活。

❑ 第 6 章 开放与封闭原则

这一章要讲解的是开放与封闭原则，它的概念非常简单，但是它对代码有着举足轻重的影响。它负责确保那些遵守所有 SOLID 原则的代码不会被改动，而只是添加新代码。这

一章还会讨论跟开放与封闭原则相关的可预测变化的概念，并且会探讨它如何能够识别后续自适应扩展的切入点。

❑ 第 7 章 Liskov 替换原则

这一章会展示在代码中应用 Liskov 替换原则所带来的好处，特别要提到的是，Liskov 替换原则有助于确保开放与封闭原则的应用，并避免代码改动所带来的副作用。这一章还会介绍代码契约，它包括前置条件、后置条件和数据不变式三个要素，可以通过代码契约工具来确保代码满足它们。此外，这一章还描述了一些子类型原则，比如协变、逆变和不变性原则，并介绍了违背这些原则会带来的不良影响。

❑ 第 8 章 接口分离原则

在实际的代码中，接口和类的问题类似，它们的规模通常太过庞大。接口分离原则是一个经常被忽视却简单有效的实践原则。这一章会展示将接口规模限制到足够小，以及配合使用这些小型接口能够带来的好处。此外还会探究可能会促使接口分离的各种不同的原因，比如客户端需求和架构需求。

❑ 第 9 章 依赖注入原则

这一章的核心是依赖注入，它能够将本书中讲解到的所有特性结合为一个整体。依赖注入真的非常重要，没有它，很多其他原则都无法起作用。这一章会详细介绍依赖注入并比较实现依赖注入的不同方法。这一章还会讨论如何使用控制反转容器来管理对象的生命周期，以避免服务定位器等反模式；此外，还会讨论如何识别组合根和解析根。

第三部分 自适应实例

这一部分以一个示例应用为主线，将本书的剩余内容组织到一起。尽管这些章节中有很多代码，但我也提供了很丰富的注释和讲解。因为本书的讲解背景是敏捷环境，所以下面这些章节会按照 Scrum 的冲刺进行组织，并且所有工作都是由积压工作项和客户变更请求而来。

❑ 第 10 章 自适应实例简介

这个部分要实际开发的示例应用是一个基于 ASP.NET MVC 5 开发的在线聊天应用。这一章会先详细描述这个应用，并给出一个简要的设计作为后续架构的指导，最后还给出了产品积压工作上所有特性的解释。

❑ 第 11 章 自适应实例冲刺 1

这一章介绍如何使用测试驱动开发方法来开发示例应用的首要特性，包括查看和创建聊天室和消息。

❑ 第 12 章 自适应实例冲刺 2

这一章讲解客户对应用提出需求变更，以及整个开发团队如何通过自适应代码来适应这些必然会发生的变更。

附录 自适应工具

附录简要介绍如何使用 Git 源代码控制从 GitHub 上下载代码, 以及如何使用 Microsoft Visual Studio 2013 编译下载的代码。请注意, 附录不是完整的 Git 使用说明, 你可以在网上找到很多非常详尽的资料, 比如 Git 的官方教程: <http://git-scm.com/docs/gittutorial>。

通过快速 Web 搜索可以找到其他来源。

附录还会简要介绍其他的一些开发工具, 比如持续集成和开发环境。

本书约定

本书中有若干反复出现的约定用法, 你可以在微软出版社的出版物中找到标准的解释, 我也在这里先给出一些简要的解释。

代码清单

书中代码清单会在适当的地方出现, 相关的代码会在同样的背景块中。比如下面的代码清单 I-1。

代码清单 I-1 这是一个代码清单, 在书中会经常出现

```
public void MyService : IService
{
}
}
```

只要看到代码清单, 你都应该关注其中特定的一部分代码。比如, 当对上一份示例代码进行改动时, 与改动相关的代码就会加粗显示。

阅读辅助和补充信息

阅读辅助主要提供一些与主题相关的边栏, 比如注意或者警告, 而补充信息则是提供一些更进一步扩展主题的信息。下面是一些示例。



注意 这是阅读辅助, 其中包括与主要内容相关的小信息, 不过具有额外的重要性。

这是补充信息

尽管已经尽量缩短篇幅, 但是补充信息通常包含与主要话题不太相关的较长讨论。

图片

有时候,无论文字解释有多么形象,还是不足以表达确切的含义。这时候,就必须提供图片了。书中所有使用 Microsoft Visio 2013 创建的图表都是黑白色的,目的是为用户提供清晰的说明。同样,截图也是在高对比度主题下获取的。

系统要求

为了使用书中提供的代码示例,需要以下硬件和软件。

- ❑ 安装了以下任意一个操作系统: Windows XP SP3(Starter Edition 除外)、Windows Vista SP2(Starter Edition 除外)、Windows 7、Windows Server 2003 SP2、Windows Server 2003 R2、Windows Server 2008 SP2 以及 Windows Server 2008 R2。
- ❑ 安装了 Visual Studio 2013 的任意版本(如果你使用的是 Express Edition 系列产品,可能需要下载几个安装包)。
- ❑ 安装了 Microsoft SQL Server 2008 Express Edition 或者更高版本(2008 或 R2 版本),以及 SQL Server Management Studio 2008 Express 或更高版本(Visual Studio 安装包中已经包括, Express Edition 则需要单独下载)。
- ❑ 处理器频率不低于 1.6 GHz(推荐 2 GHz)。
- ❑ 内存大小不低于 1 GB(32 位操作系统)或 2 GB(64 位操作系统)。如果运行虚拟机系统或者使用 SQL Server Express 版本,则另外需要 512 MB 内存,更高的 SQL Server 版本需要更多内存。
- ❑ 硬盘可用空间不低于 3.5 GB。
- ❑ 硬盘转速不低于每秒 5400 转。
- ❑ 显卡必须支持 DirectX 9,并且支持 1024×768 或者更高分辨率的显示。
- ❑ DVD 光驱(如果需要从 DVD 安装 Visual Studio)。
- ❑ 能够连接互联网以便下载软件或者代码示例。

基于不同的 Windows 配置,你可能需要本地管理员权限才能安装或配置 Visual Studio 2013 和 SQL Server 2008 等产品。

下载示例代码

我会尽力保证书中的代码片段都是一个可独立运行的应用或者单元测试中的一部分。我使用 MSTest 写了很多简单的单元测试,因此无需使用其他额外的测试器,另外我也使用 NUnit 写了一些更复杂的单元测试。我使用 Visual Studio 2013 Ultimate 编写了书中所有的代码。尽管一些代码是用 Visual Studio 2013 Ultimate 预览版编写的,但是它们都能够使用正式版成功编译和测试。我尽量不使用 Visual Studio 2013 Express 版本之外的特性,但是有些主题的代码必须使用,因此你可能需要安装一个付费版本来运行部分代码。

代码可以从 GitHub 下载，网址是：http://aka.ms/AdaptiveCode_CodeSamples。

附录包含了 Git 的使用方法简介。

我很乐意看到你的留言，下面是我的 WordPress 博客网址：<http://garymcleanhall.wordpress.com>。

致谢

本书的作者署名只有我自己，这肯定是不准确的。因为如果没有大家给我提供各方面的帮助，我根本无法完成这本书。

感谢我的妻子 Victoria，她是这本书成为可能的关键。这不是空话，这就是事实。

感谢我的女儿 Amelia，她每天的表现都十分完美。

感谢我的母亲 Pam，她帮助我校对，感谢她对我毫无保留的鼓励。

感谢我的父亲 Les，感谢他的所有付出。

感谢我的兄弟 Darryn，感谢他给我源源不断的指导。

感谢来自 Online Training Solution 公司的 Kathy Krause，她的努力让这本书读起来舒服流畅。

感谢 Devon Musgrave 的耐心帮助。

勘误、更新和相关支持

我们已经尽了最大的努力来确保书中内容以及相关材料是正确的。你可以从下面的网址得到本书的更新，其中包括完整的勘误表：<http://aka.ms/Adaptive/errata>。

如果你发现了勘误表之外的新错误或者不当之处，也请在上面的网址提交。

如果你需要额外的支持，可以发送电子邮件给微软出版社支持中心：mspinput@microsoft.com。

此外，有关微软软件或者硬件产品的支持不能通过上面的网址获得，请访问以下网址：<http://support.microsoft.com>。

微软出版社的免费电子书

微软出版社有很多免费的电子书，深入讲述了很多技术主题。这些电子书以 PDF、EPUB 和 Mobi（Kindle 电子书阅读器支持的格式）等格式供读者下载：<http://aka.ms/mspressfree>。

常去看看，你会发现很多新的电子书。

期待你的反馈

在微软出版社看来，读者的满意度始终是第一位的，读者的反馈是最有价值的资产。请在下面的网址将你对这本书的看法提交给我们：<http://aka.ms/tellpress>。

我们知道大家都很忙，所以尽量只设计了少量的简单问题。大家的回答会被直接发送给微软出版社的编辑（不需要个人信息）。谢谢大家的热心反馈。

保持联系

让我们保持联系，下面是我们的 Twitter 网址：<http://twitter.com/MicrosoftPress>。

电子书

扫描如下二维码，即可购买本书电子版。



目 录

第一部分 敏捷基础

第 1 章 Scrum 介绍	3
1.1 Scrum 与瀑布	4
1.2 角色和职责	6
1.2.1 产品负责人	7
1.2.2 Scrum 主管	7
1.2.3 开发团队	8
1.2.4 “猪”和“鸡”	8
1.3 工件	9
1.3.1 Scrum 面板	9
1.3.2 图表和度量标准	20
1.3.3 积压工作	24
1.4 冲刺	25
1.4.1 发布计划会议	26
1.4.2 冲刺计划会议	26
1.4.3 每日站立会议	28
1.4.4 冲刺演示会议	29
1.4.5 冲刺回顾会议	30
1.4.6 Scrum 日历	31
1.5 Scrum 和敏捷的问题	32
1.6 总结	36
第 2 章 依赖和分层	37
2.1 依赖的定义	38
2.1.1 一个简单的例子	38
2.1.2 使用有向图对依赖建模	44
2.2 依赖管理	48
2.2.1 实现与接口	48
2.2.2 new 代码味道	49

2.2.3 对象构造的替代方法	52
2.2.4 随从反模式	54
2.2.5 阶梯模式	56
2.2.6 依赖解析	57
2.2.7 使用 NuGet 管理依赖	67
2.3 分层	70
2.3.1 常见的模式	71
2.3.2 纵切关注点	76
2.3.3 非对称分层	77
2.4 总结	79
第 3 章 接口和设计模式	80
3.1 接口是什么	80
3.1.1 语法	80
3.1.2 显式实现	83
3.1.3 多态	87
3.2 自适应设计模式	88
3.2.1 空对象模式	88
3.2.2 适配器模式	94
3.2.3 策略模式	96
3.3 更多形式	98
3.3.1 鸭子类型	98
3.3.2 混合类型	102
3.3.3 流接口	106
3.4 总结	108
第 4 章 单元测试和重构	109
4.1 单元测试	109
4.1.1 布置、动作和断言	110
4.1.2 测试驱动开发	113
4.1.3 更复杂的测试	118

4.2 重构	131	第 7 章 Liskov 替换原则	189
4.2.1 更改已有代码	131	7.1 Liskov 替换原则介绍	189
4.2.2 一个新的账户类型	139	7.1.1 正式定义	189
4.3 总结	143	7.1.2 Liskov 替换原则的规则	190
第二部分 编写 SOLID 代码		7.2 契约	190
第 5 章 单一职责原则	147	7.2.1 前置条件	192
5.1 问题描述	147	7.2.2 后置条件	193
5.1.1 重构清晰度	150	7.2.3 数据不变式	194
5.1.2 重构抽象	153	7.2.4 Liskov 契约规则	195
5.2 单一职责原则和修饰器模式	160	7.2.5 代码契约	201
5.2.1 复合模式	162	7.3 协变和逆变	208
5.2.2 谓词修饰器	165	7.3.1 定义	208
5.2.3 分支修饰器	168	7.3.2 Liskov 类型系统规则	213
5.2.4 延迟修饰器	169	7.4 总结	216
5.2.5 日志记录修饰器	170	第 8 章 接口分离原则	217
5.2.6 性能修饰器	172	8.1 一个分离接口的示例	217
5.2.7 异步修饰器	175	8.1.1 一个简单的 CRUD 接口	217
5.2.8 修饰属性和事件	177	8.1.2 缓存	223
5.3 用策略模式替代 switch 语句	178	8.1.3 多重接口修饰	226
5.4 总结	180	8.2 客户端构建	228
第 6 章 开放与封闭原则	181	8.2.1 多实现、多实例	229
6.1 开放与封闭原则介绍	181	8.2.2 单实现、单实例	231
6.1.1 Meyer 的定义	181	8.2.3 超级接口反模式	232
6.1.2 Martin 的定义	181	8.3 接口分离	233
6.1.3 缺陷修复	182	8.3.1 客户端需要	233
6.1.4 客户端感知	182	8.3.2 架构需要	239
6.2 扩展点	183	8.3.3 单方法接口	243
6.2.1 没有扩展点的代码	183	8.4 总结	244
6.2.2 虚方法	184	第 9 章 依赖注入原则	245
6.2.3 抽象方法	184	9.1 简单的开始	245
6.2.4 接口继承	185	9.1.1 任务列表应用	248
6.2.5 “为继承设计或禁止继承”	186	9.1.2 对象图的构建	250
6.3 防止变异	186	9.1.3 控制反转	254
6.3.1 可预见的变化	187	9.2 比较复杂的注入	267
6.3.2 一个稳定的接口	187	9.2.1 服务定位器反模式	267
6.3.3 足够的自适应能力	187	9.2.2 非法注入	270
6.4 总结	188	9.2.3 组合根	272
		9.2.4 约定优于配置	277

9.3 总结	280	11.7 回顾会议	311
第三部分 自适应实例		11.7.1 什么做得比较好	312
第 10 章 自适应实例简介		11.7.2 什么做得不太好	312
10.1 Trey Research 公司		11.7.3 什么需要改变	313
10.1.1 团队		11.7.4 什么需要保持	314
10.1.2 产品		11.7.5 遇到了什么意料之外的 事情	314
10.2 最初的产品积压工作		11.8 总结	315
10.2.1 从描述中挖掘故事		第 12 章 自适应实例冲刺 2	316
10.2.2 故事点估算		12.1 计划会议	316
10.3 总结		12.2 “我想发送正确格式化的标记”	317
第 11 章 自适应实例冲刺 1		12.3 “我想过滤消息内容以确保它是适合 发表的”	321
11.1 计划会议		12.4 “我想同时服务数百个用户”	323
11.2 “我想创建多个房间以对会话进行 分类”		12.5 演示会议	325
11.2.1 控制器		12.6 回顾会议	326
11.2.2 房间存储库		12.6.1 什么做得比较好	326
11.3 “我想查看代表会话的房间的 列表”		12.6.2 什么做得不太好	327
11.4 “我想查看发送到一个房间内的 消息”		12.6.3 什么需要改变	327
11.5 “我想给房间内的其他成员发送 纯文本消息”		12.6.4 什么需要保持	327
11.6 演示会议		12.6.5 遇到了什么意料之外的 事情	327
		12.7 总结	328
		附录 自适应工具	329

Part 1

第一部分

敏捷基础

本 部 分 内 容

- 第 1 章 Scrum 介绍
- 第 2 章 依赖和分层
- 第 3 章 接口和设计模式
- 第 4 章 单元测试和重构

这一部分主要介绍敏捷原则和实践的基础知识。

编写代码是软件开发的核心工作，而编写好用的代码有很多不同的方式。即使抛开平台、语言和框架的影响，对于一个开发人员，最简单的一个功能的实现也会有多种选择。

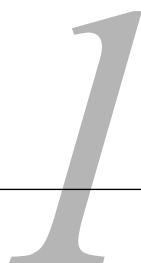
在软件开发产业，开发成功的软件产品一直以来都是焦点。但是近几年，开发人员开始重视那些能够被重用并能提高代码质量的实现模式和实践，因为大家逐渐意识到软件产品的质量是无法与代码的质量割裂开来的。随着时间的推移，质量差的代码会逐渐降低产品的质量，至少一定会延迟可工作软件的完整交付。

为了开发高质量的软件产品，开发人员必须努力确保编写的代码是可维护的、可读的，并且是经过测试的。在此基础上，对开发人员提出了一个新的要求：编写的代码也应该具备一定的自适应变更的能力。

这一部分的四个章节主要介绍现代的软件开发流程和实践。这些流程和实践有一个统一的类型名称，那就是敏捷，以表达它们具有快速响应变更和改变方向的能力。敏捷流程（Agile Process）给软件开发团队推荐了很多的方法，用于快速得到反馈、响应并调整工作焦点。敏捷实践（Agile Practice）还推荐了很多方法来帮助开发团队编写出自适应代码。

第 1 章

Scrum介绍



完成本章学习之后，你将学到以下技能。

- ❑ 给项目的主要干系人分配角色。
- ❑ 识别Scrum需要生成的各种文档和其他工件。
- ❑ 监测Scrum项目进度。
- ❑ 诊断Scrum项目的问题并提出补救措施。
- ❑ 高效主持Scrum会议以达成最大的会议成果。
- ❑ 对比Scrum和其他敏捷或严苛方法之间的优劣。

Scrum是一个具体的项目管理方法论。更准确地说，它是敏捷方法的一种。Scrum的核心概念是以迭代的方式为软件产品增加价值。整个Scrum流程是可重复的，也可以迭代多次，一直持续到整个产品完成或者流程被终止。这些迭代被称为冲刺（sprint）。经过若干个冲刺得到的软件是随时可以发布的。整个软件产品的所有工作项都会在产品积压工作（product backlog）上按照优先级排列，在每个冲刺开始时，开发团队会把在这个新的冲刺中承诺要完成的工作项添加到冲刺积压工作（sprint backlog）上。Scrum中工作项的单位是故事（story）。产品积压工作实际上就是一个排好序的候选故事队列，每个冲刺则由要在这一个冲刺中承诺要开发完成的故事组成。图1-1展示了Scrum流程的框架。

Scrum过程中，开发团队内部或外部相关角色人员会产出一些文档工件，此外他们还会一起参加一些会议。只用一章的篇幅当然无法从项目的角度完整展示整个Scrum的细节，但是本章会尽量提供足够多的细节，为你进一步深入学习Scrum打下基础，并为后续的每日实践提供一个方向性的指导。

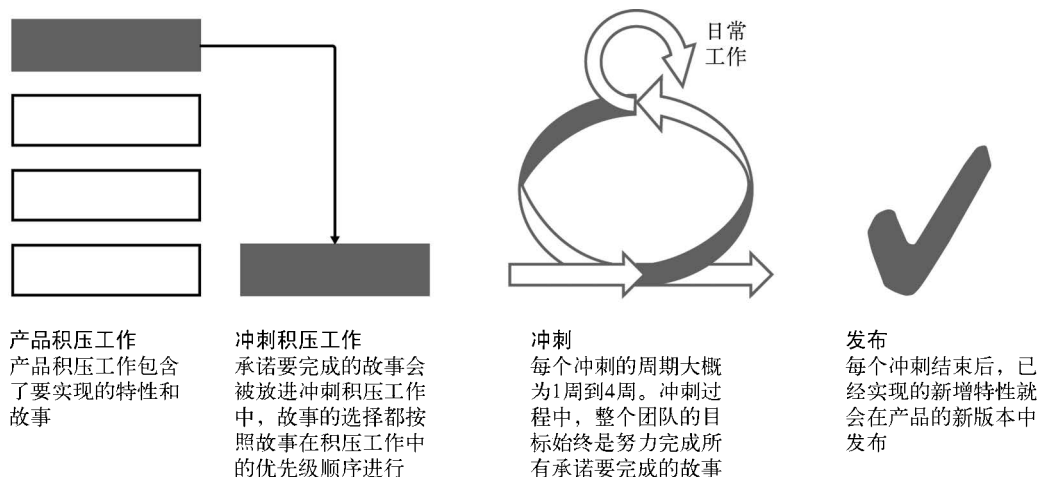


图1-1 Scrum的工作方式看起来就像一条为软件产品逐步添加小特性的生产线

Scrum是一种敏捷方法

敏捷方法论 (Agile) 包括一组轻量级软件开发方法，它们都允许及时响应客户的需求，即使项目已经在进行过程中了。敏捷是在很多严苛的结构化项目实践失败的教训中应运而生的。敏捷宣言列出了敏捷和严苛方法论之间详细的对比项。大家可以在以下网址查看完整的敏捷宣言：<http://www.agilemanifesto.org>。

敏捷宣言最初只是由十七位开发人员联名发起的。但是从那一刻起，敏捷方法的影响力就在日益扩展，现在已经成为了敏捷大环境下各个软件开发角色公认的基本约定。Scrum是目前敏捷方法论中应用最广泛、最常见的一个流程实现。

1.1 Scrum 与瀑布

根据我多年的软件产品开发经验，敏捷方法比瀑布方法工作的效果要好，而且我也乐于推广各种敏捷流程。瀑布方法论的根本问题在于它过于严苛和僵化。图1-2展示了一个瀑布工程涉及到的流程阶段。

从图1-2可以看到，每个阶段的输出都是下一个阶段的输入，而且每个阶段必须在进入下一个阶段前完成。这就要求在一个阶段完成时没有遗留任何错误、问题、难点或者误解，因为整个过程只是单向的。

瀑布流程还要求在任一阶段完成后不允许再发生任何更改，而这与大量的经验和统计数据不符。变化本身就是生活固有的一部分，软件工程也不例外。瀑布方法支持的这种应对变更的僵硬

态度是高代价的、不值得的，而且是肯定可以避免的。瀑布方法论认定可以花费更多时间在需求和设计阶段来识别出所有潜在的变更，这样在后续阶段就不会发生变更了。这个观点很不靠谱，因为变更总会发生，不论你愿意与否。

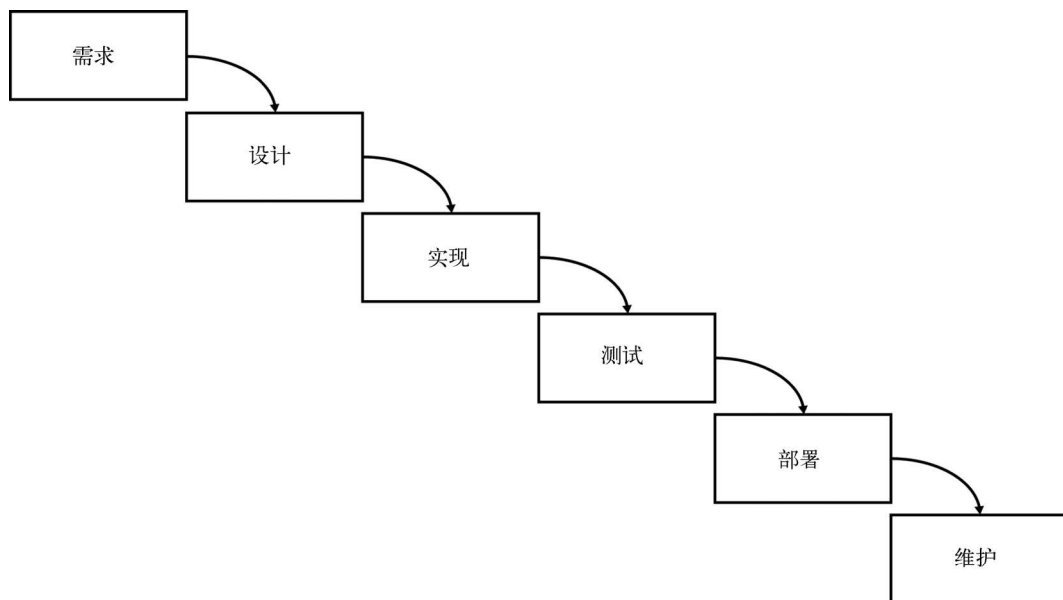


图1-2 瀑布开发流程

为了应对变更，敏捷流程引入了另外一种方法，这个方法主动拥抱变化并且允许每个人都能快速响应任何变更。尽管敏捷（包括Scrum）提供了流程级别上的变更响应机制，但是在现代软件开发的信条里，要在代码级别响应变更是最困难的，也是最重要的。本书的宗旨就是竭力为你展示如何编写出能够灵活地自适应变更的代码。

另外，瀑布方法论是以文档为核心的，会产出大量的文档，而这些文档并不能直接改善软件产品。敏捷则恰恰相反，它认为能工作的软件就是这个软件产品最重要的文档。毕竟真正的软件行为是由软件源代码定义的，而不是由与代码相关的文档决定的。此外，瀑布方法论的文档和代码是完全分开的，所以很容易就会出现文档没有与代码完全同步的情况。

Scrum设置了一些度量标准，用于反映项目进度和整体健康状况，这些标准与产品的说明性文档是不同的。总体而言，敏捷倾向于有适量的文档以避免规避责任的现象，但是它也不强制要求必须撰写这些文档。如果这些文档不是撰写一次就再也不会被查阅的话，那么有些代码肯定能从这些支持文档中获益。出于这个原因，易用的在线文档（比如Wiki）就成为了Scrum团队很常用的工具。

本章的剩余部分将会更深入地讨论Scrum中最重要的方面，其中讨论的不完全是Scrum，也有与Scrum相关的常见变体。Scrum流程的目标不仅仅是通过迭代的方式改进软件产品，而且也

包括改进开发流程。在Scrum团队特有的状况和上下文中，Scrum流程鼓励团队持续微调来保证整个流程对所有成员来说是合适的。

本章在讨论过Scrum的基本构成后，还会指出它的一些缺陷。这一章是本书的基础，由于Scrum流程承诺拥抱变更，后续章节就会在此基础上详细讲解如何编写出能够自适应变更的代码。一个声称自己可以优雅地处理变更但很难在代码层次实现的流程是没有意义的。

Scrum的不同形态

任何时候，当一个开发团队声明他们遵循了Scrum方法时，通常是说他们自己遵循了Scrum的某种**变体**（variant）。纯正的Scrum并不包含很多常见的实践活动，它们来源于诸如极限编程（eXtreme Programming, XP）等其他的一些敏捷方法。Scrum有三个分支，它们的实现都逐渐偏离了纯粹的Scrum理论。

增强的Scrum

Scrum并不包含一些常见的实践活动，比如测试先行以及结对编程。但是对于很多团队而言，这些实践活动是对Scrum流程本身很好的补充，因此它们被认为是有助作用的实践。当团队从其他敏捷方法（比如极限编程或者看板）引入一些实践活动时候，实际操作的流程应该被称为增强的Scrum。这意味着，Scrum和一些优秀实践活动的组合起到增强而非削弱标准Scrum流程的作用。

弱化的Scrum

一些开发团队声称他们正在实践Scrum，但是却忽略了一些关键点。他们在产品积压工作上按照优先级顺序排列了工作项，在Sprint冲刺中选择了要完成的工作项，而且有追溯会议以及每日站立会议。但是，他们并没有按照故事点估算故事，而是采用真正的时间估算。这种Scrum流程的变体被称为弱化的Scrum。这种情况下，团队尽管在很多方面应用Scrum定义的实践活动，但是实际上却忽视了几个关键的活动。

根本不是Scrum

如果一个开发团队的实践严重偏离了Scrum方法，他们实际上就不是在使用Scrum。这一定会在开发过程中引起问题，特别是当团队成员希望能应用一种敏捷方法而实际的流程却一点都不像Scrum流程时。我发现每日站立会议是最容易被开发团队采纳的Scrum实践活动，但是让团队成员在心中对变更进行相对估算并且积极拥抱变更却很难。当很多Scrum流程的实践活动被团队放弃后，实际运行的流程就根本不是Scrum了。

1.2 角色和职责

Scrum仅仅是个流程，我要反复强调的是，只有团队成员都遵循流程它才会起作用。不同角色的人有着不同的职责，不同的职责则需要采取不同的行动。

1.2.1 产品负责人

产品负责人（Product Owner，PO）的角色在Scrum中非常重要，因为产品负责人是Scrum团队和客户之间唯一的联系。产品负责人要对最终产品负责，他们负有以下相应的责任。

- ❑ 决定要构建哪些特性。
- ❑ 根据业务价值设定特性的优先级。
- ❑ 接受或者拒绝“已完成”工作。

作为项目成功的关键干系人，产品负责人必须时刻准备着为团队服务，给他们清晰呈现项目的愿景。所有开发团队成员都应该清楚地知道项目的长期目标，而且都能够及时清楚地了解到变更的详细信息。在制定短期冲刺计划时，产品负责人应首先安排好要开发什么以及什么时候开始。产品负责人按照软件的发布计划来选择特性，并且在产品积压工作中对这些特性设定优先级。

尽管产品负责人是个关键角色，但是这并不意味着产品负责人在整个Scrum流程中的影响不受约束。产品负责人不能决定开发团队一个冲刺内需要完成的工作量，因为这是由开发团队自身的开发速度决定的。同样，产品负责人不能决定如何实现工作项，因为开发团队会从技术层面上决定一个故事的详细实现方案。当然，在冲刺的过程中，产品负责人不能改动冲刺目标，改变验收标准，或者增删故事。经过冲刺计划会议，目标和承诺的故事已经选定，此时处于运行状态的冲刺就是不可更改的了。任何改动都必须等到下一个冲刺，除非明确中止当前冲刺或者整个项目，然后重新开始。冲刺的这种不可破坏性保证了开发团队能够在整个冲刺进行过程中心无旁骛地朝着既定的目标努力。

整个冲刺期间，无论故事在进行中还是已经完成，产品负责人都要经常去试用进行中的产品，看看特性的状态如何，或者给正在进行的任务提提意见。产品负责人多花点时间与开发团队保持紧密的沟通是很重要的，这样才可以及时应对那些突然出现的意外和混乱。到冲刺结束时，产品负责人不能简单地接受那些声明“已经完成”但却偏离了初始目标的故事，而是应该通过制定好的验收标准来检验一个故事是否已经完成以及是否可以展示。

1.2.2 Scrum 主管

Scrum主管（Scrum Master，SM）负责在冲刺进行过程中为团队隔离所有外部影响，并且处理团队成员在每日站立会议上提到的各种影响开发的障碍。这样才可以保证在冲刺期间，整个开发团队能够高效地朝着当前的既定目标努力。

产品负责人要对做出怎样的产品负责，Scrum主管则要对如何完成产品的流程负责。因此，Scrum主管的职责就是确保整个团队按照既定的流程来完成既定的产品目标。Scrum主管能够主导对流程的改进（比如把冲刺周期从四周缩短到两周），但是他们的权限也是有限的。比如Scrum主管只可以指导开发团队按照Scrum流程开发，而不可以越俎代庖地指定开发团队如何实现一个故事。

作为流程负责人，Scrum主管要负责组织每日站立会议，他们要确保所有开发团队成员都出席会议，并且记录会议纪要以防遗漏某些行动项。不过，这并不代表着团队成员要在会议上给

Scrum主管做工作汇报，每个成员参加站立会议的真正意图是为了让所有与会者都能大概了解到自己的工作项进度和状况。

1.2.3 开发团队

理想情况下，一个敏捷团队由一些全科专家（generalizing specialist）组成。也就是说，团队的每个成员应该能够熟练使用多个领域的技术，而且精通或特别擅长其中的某个领域。比如说，在一个由四名开发人员组成的敏捷团队中，每个开发人员都能胜任所有与ASP.NET、MVC、Windows Workflow以及Windows Communication Foundation（WCF）相关的工作项；但是其中两个开发人员特别擅长Windows Forms，另外两人则喜欢使用Windows Presentation Foundation（WPF）和Microsoft SQL Server。

团队中开发人员的技能重合能够防止团队中出现筒仓现象。筒仓现象的表现是，团队中的各种专家（比如网页开发专家、数据库专家或WPF专家等）各自掌握了产品开发必需的一种知识。Scrum中，代码是由开发团队集体持有的，而这种筒仓现象会妨碍全体开发人员的参与度。因此在组建团队的时候，应该尽可能避免出现这种筒仓现象。此外，筒仓也不利于业务的开展，因为业务的某个领域会过度依赖单个“专家”开发人员的能力，而且这些“专家”开发人员顶着“他们是唯一能完成这些工作项的人”的大帽子，压力也会非常大。

软件测试人员负责保证所开发软件的质量。在开始实现一个故事前，测试人员可能需要规划自动化测试以保证故事的实现符合各种预先定义的验收标准。他们要么与开发人员一起制定计划，要么单独完成。当开发人员完成一个故事时，就会提交故事的实现以供测试，然后测试分析人员会核实软件产品是否按照要求的那样正常工作。

1.2.4 “猪”和“鸡”

Scrum流程中的所有角色都可以划分为两类：“猪”和“鸡”。这种形象的比喻来源于下面的故事。有一天，鸡对它的好朋友猪说：“猪哎，我有个很棒的想法，我们应该开个饭店！”猪也觉得主意不错，很兴奋地问鸡：“那我们给它起个啥名呢？”鸡说：“要不就叫‘火腿和鸡蛋’吧！”猪想了一下就发火了，说道：“门都没有，你只是下下蛋，我却要割肉！”

这个寓言故事很好笑，用在这里只是用来强调一个项目中的不同角色有着不同的参与度。“猪”需要全身心投入到项目中，并且要对它们的产出负责；而“鸡”只是有所贡献，不会深入参与到项目中。产品负责人、Scrum主管和开发团队扮演的都是“猪”的角色，因为他们都需要全身心致力于产品的交付。通常，客户不需要深入产品开发中，因此他们扮演的是“鸡”的角色。类似地，因为执行管理层的支持工作是针对项目本身的，而不是产品，所以也应该是“鸡”而不是“猪”。

1.3 工件

在所有软件项目的生命周期内都会创建、评审和分解细化很多的文档、图表和度量标准。从这个角度看，一个Scrum项目也会有同样的工件。然而，Scrum文档与其他类型项目管理的文档有着不同的目的和类型。所有敏捷流程和严苛流程之间的一个关键区别，就在于文档的重要性上。比如，结构化系统分析和设计方法（Structured Systems Analysis and Design Methodology, SSADM）就特别强调需要撰写很多的文档。这也被笑称为“大设计优先”（Big Design Up Front, BDUF）理论，这种理论相信，如果能够花足够多的时间和人力资源来撰写文档，所有的恐惧、不确定性和怀疑就可以从项目中清除。敏捷流程则致力于减少文档的数量，只保留那些对于项目成功至关重要的文档。此外，敏捷流程更看重代码而非文档，因为代码本身作为最具有权威性的文档，是随时可以部署、运行和使用的。敏捷流程还倾向于所有干系人都直接沟通，而不是写一些很少有人看的文档。总而言之，文档对于敏捷项目依然是重要的，但是它不能取代可工作的软件和沟通本身。

1.3.1 Scrum 面板

Scrum项目日常工作的中心区域是Scrum面板。应该为面板保留几面墙的空间，太小的面板将没有空间展示重要的细节。也许专门为此在办公室建足够大的墙面代价会比较大，你也可以采用其他的一些性价比较高的办法。比如可以重新利用那些经常被人忽视的、比较大的白板作为Scrum面板，再加上磁铁和金属填充格，就更像一个Scrum面板了。如果办公室是租来的，或者因某种原因不能损坏墙壁，还可以使用“神奇的”白板，即白纸，因为它方便简洁且无需擦除内容。一定要尝试在办公室找一块合适的地方放置Scrum面板，无论选择了什么样的面板，也无论如何放置它，如果在使用了几个冲刺后，感觉不合适，可以随时改变它。对于Scrum流程而言，Scrum面板实物是必备的，其他的工具（比如数字化面板等）永远无法给你那种站在实际面板前的体验和感觉。尽管数字化面板也有它们的用途，但我仍然相信它们只是辅助工具。图1-3展示了一个典型的Scrum面板。

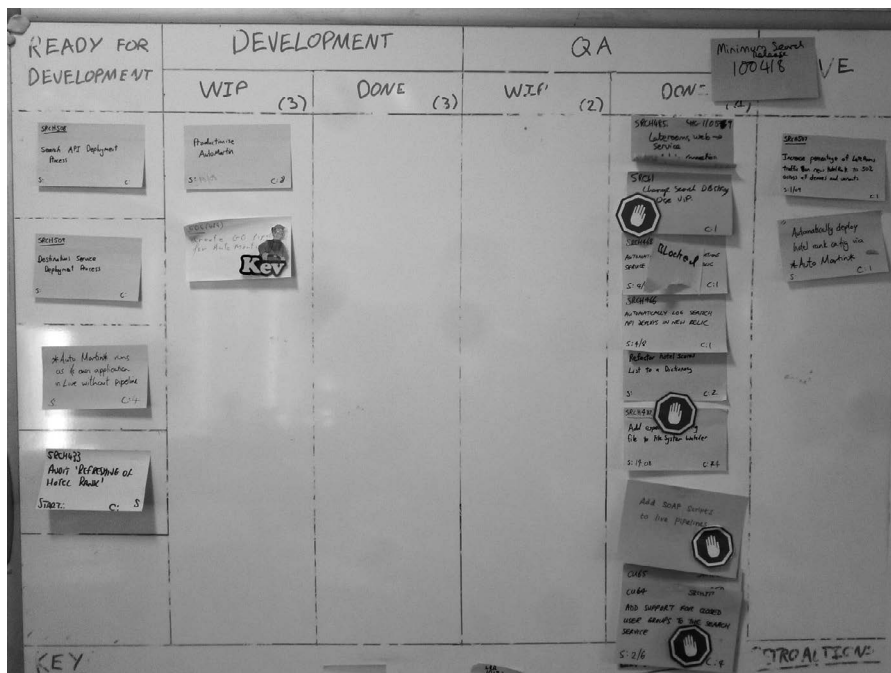


图1-3 一个Scrum面板，展示了当前的开发状态

Scrum面板是项目信息的聚集地，包括了很多细节，并能展现出正在进行的工作项的重要性。下面几节会全面详细地介绍Scrum面板的使用。

1. 卡片

Scrum面板上的主要物品就是卡片。卡片会用来展示软件产品进度的不同元素，从软件的发布到最细小的独立任务。为了清晰起见，不同类型的卡片应该有不同的颜色。因为空间限制，Scrum面板通常只用来展示与当前冲刺相关的故事、任务、缺陷以及技术债务（technical debt）。



提示 单独使用颜色来区分可能无法满足每位团队成员的需求。比如，对于那些无法分辨颜色的团队成员而言，可以辅助使用不同的形状来作卡片类型的区分。

● 组成的层次结构

图1-4展示了Scrum面板上不同类型卡片的层次关系。请注意，这里的前提是产品是由很多个任务组成的。即使是最复杂的软件也可以分解成为有限的若干个独立任务，要完成整个软件，就必须先完成这些任务。

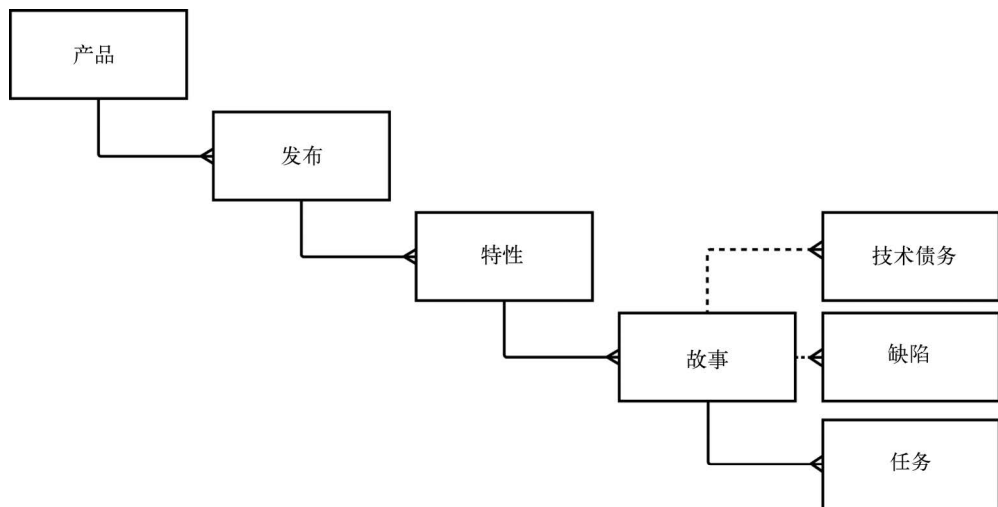


图1-4 Scrum面板上的不同类型的卡片展示了组成产品的不同元素以及它们之间的层次关系

● 产品

Scrum食物链的顶端是要构建的软件产品。软件产品的例子很多：集成开发环境（IDE）、网络应用程序、会计软件、社交媒体应用程序等。你要开发的是软件，要交付的是产品。

通常每个开发团队每次只开发一个产品，但是有时候单个团队也可能会同时为交付多个产品负责。

● 发布

每个要开发的产品都会有多个发布。一个发布就是一个特定版本的软件，用户可以购买软件或者使用该软件提供的服务。有的发布是为了解决若干缺陷，有的发布则会为关键用户提供有价值的新特性，还有的发布只是提供测试版本来让用户尝鲜和反馈。

网络应用程序通常只会在所有发布前部署一次，版本变更也不会很明显。实际上，Google Chrome网络浏览器就是个很有意思的范例。尽管它是一个桌面应用，但是它的每次发布都很小，对用户而言几乎没有感觉，这与其他浏览器的高调发布方式截然相反。像Internet Explorer 8、9和10分别都有自己的广告投放，而Chrome则没有选择这样的发布模式，Google只是简单地为浏览器本身做推广，推广的时候也不会提及版本。而这种迭代式的发布方式也变得越来越流行。Scrum在每个冲刺后能够很自然地通过专注于交付可工作的软件来支持这种发布模式。

最小可行发布

第一个发布可以是一个**最小可行发布**（Minimum Viable Release, MVR），它包括了能够满足用户最基本需求的一组特性。比如，对于会计软件，最基本的特性集合包括：创建新客户，客户账户上的存取交易以及账户金额汇总。最小可行发布的核心目的就是引导项目尽早

做到自筹资金并继续发展。尽管最小可行发布的结果不太可能达到自筹资金这个目标，但是至少通过这种发布可以提前获得一些回报来抵消部分开发成本。此外，即使最小可行发布的目标客户范围受限，它的部署也能够获得宝贵的用户反馈，或许还会影响到后续该软件发展的方向。这种及时调整方向的能力也是Scrum（以及其他敏捷方法）自身与生俱来的，因为这些敏捷方法都认为软件不断进化的前提就是软件必须要变化。

无论发布的目的和方式（或者频率）如何，理想情况下，单个产品的生命周期内都会包含多个发布。

● 特性

每个发布由上一个发布的软件没有的若干个新特性组成。任何软件的版本1.0和版本2.0的最显著的区别就是那些团队成员认为能够吸引新老用户购买软件或者升级包的新特性。

最小可售特性（Minimum Marketable Feature, MMF）这个术语可以用来界定一个发布需要的特性集合。下面列出了一些示例特性，这些特性非常通用，可以应用于许多不同的项目，同时也非常特定，它们就是真实世界的特性。

- ☐ 以可移植的XML格式导出数据
- ☐ 需要在0.5秒内响应网页请求
- ☐ 保存数据以备后续使用
- ☐ 复制和粘贴文本
- ☐ 与同事在网络上共享文件

特性如果能给客户带来价值，那么它就是可以出售的。在尽量提炼出最小功能集合的同时依然能保持其具有可交付的价值，此时得到的特性的粒度也是最小的。

史诗/特性、最小可售特性与主题

当谈论Scrum的时候，可能更经常使用的术语是**史诗**（epic）而不是**特性**（feature），我个人不太经常使用前者。史诗和特性通常被看作“大型的故事”，也就是说，这些故事比最小可售特性大很多，因此无法在一个单独的冲刺后交付。

特性也与Scrum的另外一个术语**主题**（theme）类似，主题是指一组有着共同目标的故事。

对每个发布而言，可以将特性划分为三类：必需的、可选的和想要的。这三个分类是互斥的，用于反映每个特性的优先级。通常，开发团队必须首先完成所有必需的特性，然后才是可选的特性，此外，只有在时间允许的情况下才可以触及想要的特性。你也许已经猜到，这些分类和特性本身总是可变的。当团队想要切换工作焦点的时候（当然，这会附带引起计划和经费的改变），可以在任意时间中止它们、对其重新排序、交换其顺序以及废弃它们。Scrum中的一切都是不确定的，而本书就是致力于帮助你学会应对这些不确定性。

- 用户故事

用户故事可能是绝大多数开发人员最熟悉的Scrum工件，但实际上它根本不是由Scrum定义和提出的。用户故事最先是极限编程方法定义的工件，因为它在软件开发领域变得很常用，因此Scrum方法后来也引入了它。用户故事需要通过下面的模板来声明。

“作为 [某个用户角色]，我想要 [做某种行为]，以便于 [给这个用户角色带来某种价值]。”

上面模板中，方括号中的内容会因具体用户故事而有所不同。下面举个具体的例子来作进一步的解释。

“作为一个注册但未经验证的用户，我想要重置我的密码，以便于当我忘记密码的时候还能够再次登录系统。”

这个用户故事描述示例包含了很多值得注意的地方。首先，这个故事并不包含足够的信息来实现必需的行为。特别要注意的是，用户故事是从用户的角度出发编写的。尽管这一点看起来很明显，但实际上很多人都忽略了这一点，很多故事都是错误地从开发人员的角度编写的。这就是为什么用户故事模板中的第一部分“作为 [某个用户角色]”至为重要的原因。类似地，第三部分“以便于 [给这个用户角色带来某种价值]”也同等重要，因为如果没有它，那么这个故事存在的根本原因就会被忽略掉。通常这部分也表达了用户故事与父特性之间的联系；上面这个用户故事示例可能属于诸如“被忘记的凭据是可恢复的”这样的特性。也很可能与“用户忘记了登录名”以及“用户忘记了登录名和密码”这两个用户故事归属于同一主题。

既然并不能单单从用户故事开始开发工作，那么它的价值何在？用户故事代表了开发团队与客户之间的对话。当需要实现某个用户故事时，开发人员会带着故事与客户讨论具体的需求，并且会产生出在整个用户故事的生命周期内都必需遵守的若干个验收标准，仅可以将完全符合了所有验收标准的用户故事标注为已完成。

当需求收集完之后，开发人员就需要开始准备一些设计来满足这些需求。这个阶段可能需要使用Balsamiq、Microsoft Visio或者其他一些工具来制作用户界面的模型。通常使用统一建模语言（Unified Modeling Language, UML）图表来从技术角度上详细表达如何改动现有代码以满足这些新的需求。

当设计确定后，团队就可以开始将用户故事分解为小的任务，然后通过完成这些任务来实现整个故事。当开发人员认为故事的实现已经满足要求了，那么他们就可以交付故事实现以进行验收测试。最后的质量保证（quality assurance, QA）阶段会再次根据验收标准来对可工作的软件进行测试验收，并决定批准或者驳回该故事的实现。如果用户故事的实现得到了批准，那么才真正完成了这个用户故事。

让我们花点时间来回顾一下前面讲述的要点。以一个用户故事为依据，开发人员在分析阶段收集需求，然后准备好设计方案，接着做具体实现，最后按照验收标准做测试。这听起来就像是一个瀑布开发方法的过程。这的确是用户故事的要点，为每个用户故事执行一遍小规模但完整的软件开发生命周期。这种方式有助于防止出现无用功，因为在没有分解和实现用户故事之前，

开发人员不需要做什么，但是他们一直知道用户故事依旧符合软件产品的价值主题。

用户故事是Scrum过程中最主要的工作重心；Scrum流程是通过使用用户故事的故事点数来激励团队成员的。在冲刺计划会议上，团队一起给每个故事估算故事点数，当团队完成了某个用户故事时，就认为团队已经挣取了它的故事点数，并可以从整个冲刺的总点数中扣除该故事的所有点数了。后面的章节会进一步详细讲解故事点数的概念和应用。

● 任务

任务是比用户故事还要小的工作项。可以把一个用户故事分解成多个容易管理的任务，然后分配给多个开发人员并行开发。我倾向于在准备实现故事时再开始做任务分解，但是也有很多人会在冲刺的计划会议时就完成了所有承诺要完成的故事的任务分解。

尽管用户故事是在功能角度上的完整竖切，但是分解得到的任务依然可以分层，这样就可以充分利用团队成员的技术特长。比如说，要给一个数据表增加一个新的数据项，这很可能需要改动用户界面、业务逻辑以及数据访问层实现。团队可以将这个故事分解成三个任务，它们分别针对分解得到的三层需求并指派给三个各有特长的开发人员：WPF高手、C#能手以及数据库大咖。当然，如果团队成员全都是全能技术能手，任何人都可以在任何时间胜任任何任务，那么你就太幸运了。这种分层分解任务的方式可以让每个团队成员拥有很广的选择任务范围，也有助于他们理解整个项目并有更高的满意度。

竖 切

在我小的时候，爸爸每年圣诞节都会做蛋奶层叠蛋糕。这是一种传统的多层英国甜点。最下面是水果块层，然后是松糕层、果酱层和牛奶沙司层，最上面是奶油层。我哥哥喜欢用勺子一直从上朝下吃，而我则喜欢换着吃不同的层。

好的软件设计就像层叠蛋糕一样分层。最下面是数据访问层，然后是对象关系映射层、域模型层、服务层和控制层，最上面是用户界面层。与吃层叠蛋糕一样，开发这种分层软件也有两种方式：横切和竖切。

如果选择横切方式，每层都需要作为一个整体来实现。但是这样不能保证每层的实现都协调同步。用户界面也许已经可以允许用户去交互完成某些特性，但是下面的功能层还没有实现。实际的结果就是用户必须要等待所有层都配合完成这个功能后才可以使用这个应用。这会导致无法及时得到用户的反馈，并且会增加做无用功或者做错的可能性。

在敏捷流程里，应该选择竖切方式。每个用户故事应该由每层上的功能组合而成，并且与最上层的用户界面联系起来。这样才可以完整演示某个功能来获取用户的及时反馈。这种方式也可以避免从程序员角度撰写用户故事，比如“我想能查询数据库以知道那些本月没有缴费的用户清单”这种描述听起来像个任务；而正确的故事描述应该是“生成未付费账户的报告清单”。

请注意，只有用户故事才持有故事点数，用户故事分解得到的任务并不能继承和持有故事点数。可以说五个故事点数的用户故事被分解为三个独立的任务，但是不可说两个一点的任务和一个三点的任务组成了五个点数的用户故事。这是因为完成单独的任务并不会获得故事集点数的

激励。只有当测试人员和产品负责人在冲刺结束前确认整个用户故事已经完成了才可以获取该故事的所有用户点数。如果没有完成或只是部分完成，整个用户故事的故事点数依然不能从冲刺故事点数中扣除。理想情况下，用户故事会在一个冲刺内完成，如果没有完成，应该在下一个冲刺继续。如果一个故事因耗时太长而无法在一个单独的冲刺内完成，那说明这个用户故事的粒度太大，应该分解为多个更小、更容易管理的故事。

- 技术债务

技术债务是个很有意思的概念，但是它也很容易被误解。它是个隐喻，用于描述在一个用户故事生命周期内在架构设计和实现上所做的折中和妥协。本章后面专门有一节讲解技术债务。

- 缺陷

如果某些完成的用户故事没有符合某些验收标准，就需要创建缺陷卡片了。这就要求有自动化的验收测试：针对某个用户故事撰写的一组测试就形成了一个回归测试套件，可以用于保证后续的工作不会破坏已通过验收的用户故事的实现。

与技术债务一样，缺陷卡片也不会持有故事点数，因此修复缺陷和技术债务带来的问题并不会获得故事点数激励。虽然做到零缺陷和零债务很不现实，但是开发人员应该尽最大努力去避免引入缺陷和技术债务。

所有软件都有缺陷，这是软件开发无法逃避的事实，缺乏计划或不勤勉并不是人会犯错误的根本原因。缺陷通常会被划分为三大类：灾难缺陷(Apocalyptic defect)、行为错误(Behavioral error)和外观上的问题(Cosmetic issue)；按照首字母也被称为A、B、C三类。

灾难缺陷会直接导致应用程序的彻底崩溃或者导致用户的操作无法继续。未捕获异常是一个典型的例子，因为此时应用必须先强制结束然后再次启动，或者是必须重新加载一次网络场景下的网页。这些缺陷应该具有最高优先级，并且必须在软件发布前修复好。

行为错误不像灾难缺陷那样严重但是也会令用户不满。这种类型的错误还有可能比直接让应用程序崩溃更有破坏性。试想一下，如果错误的货币转换逻辑把账户资金数额的小数部分给弄没了，无论算法的错误是对用户还是业务有害，有人终究是要为这样的行为错误承担资金损失的。当然，不是所有的逻辑错误都会这么严重，但是这个例子有助于让我们明白，行为错误可以是中优先级，也可以是高优先级。

界面问题通常是和用户界面相关的问题，比如图片未对齐，窗口无法全屏，或者网页上某个图片无法加载显示等。这些问题不影响软件的使用，只是影响它的外观。尽管这种问题通常只是低优先级的，但是它们也很重要，因为界面的表现也是用户对软件的期望之一。如果用户界面上的按钮无法使用，图片无法加载显示，用户也自然无法相信软件的内部行为会正常工作。相反地，一个花里胡哨的用户界面也会让用户感到这个软件并不是针对公众发布的。信誉不好的项目通常需要重新设计用户界面(甚至可能改变产品品牌)来改善市场对该产品的看法以及重设用户对该产品的期望。

让卡片意图更明确

有很多种办法可以自定义以及个性化Scrum面板上的卡片。

颜色主题

在卡片上应用任何颜色主题都可以，不过按照我的经验，其中一些比较合理。特性和用户故事可以使用索引卡（index card），任务、缺陷和技术债务可以使用即时贴（sticky note），因为可以很方便地把它贴在相关故事附近。下面是我推荐的使用方式。

- 特性：绿色的索引卡
- 用户故事：白色的索引卡
- 任务：黄色的即时贴
- 缺陷：红色或粉色的即时贴
- 技术债务：紫色或蓝色的即时贴

注意，作为使用卡片最多的用户故事和任务，它们应该使用最常见可用的索引卡和即时贴。此外为了避免索引卡不够用，请尽量使用最常见且可用的颜色。

谁来创建卡片？

这个问题的答案很简单直接，那就是“任何人”。当然，在实际应用中也会有些条件。虽然任何人都可以创建一个卡片，但是该卡片的有效性、优先级、严重程度以及其他一些状态值并不能只由创建者单独决定。所有的特性和用户故事卡片必须由产品负责人最终核实，而任务、缺陷和技术债务卡片则只应该由开发团队来核实。

头像

类似于网上论坛、博客和Twitter上的用户头像，团队的每个成员也要有个自己的小画像。让团队成员自己选择自己的头像，这也是Scrum流程中比较有趣的一个过程。当然，需要引导团队成员不要选择有冒犯性的头像，只要保证最终每个人的头像是有所区别的就可以。

在迭代过程中，这些头像会被移动好多次，通常是每天移动一次。因为Scrum面板上已经贴满了故事索引卡和任务即时贴，因此这些头像不应该大于两寸照片。使用薄板覆压图像，可以防止其出现折角和破损，还可以用小片胶带把头像固定在一个地方。

2. 泳道

在Scrum面板上，通过多个垂直线划分出了多条泳道。每条泳道可以包含多个用户故事卡以表示相关故事在其开发生命周期内的进度。典型的排列是从左到右的四条泳道：积压工作、开发、验收和完成。

积压工作泳道中的故事是开发团队已经承诺在当前冲刺要进行开发的，所以它们迟早会被移动到开发泳道中，除非它们在开发前被中止了。这条泳道内的故事卡可以按照优先级排列，这样最上面的卡片总是包含下一个要实现的故事。

从积压工作泳道中取出故事后，首先要先和产品负责人沟通以确定该故事的范围和需求，然后分解为多个独立的任务，最后把该故事和分解它得到的多个任务一起放进开发泳道中。此时，所有参与开发该故事的团队成员的小头像也应该贴在故事卡周围了。不同的泳道会有不同的限制。比如，你也许要求最多同步开发三个故事，这样才可以强迫团队优先完成已经开始的故事，而不是再从积压工作泳道中取出新的故事。请切记：没有完全完成的故事无法争取该故事的任何点数。

当一个故事经过分析、设计和实现后，它会被标记为“已经完成开发”的状态，此时就可以把它移动到验证泳道中了。理想情况下，验证环境应该和产品环境尽可能地一致，以避免后期部署时因为环境的些许不同而发生错误。测试分析人员使用验收标准来评估故事。本质上讲，他们要做的是尽量破坏故事以证明代码无法按照预期正常工作。通常，测试人员通过给某些操作输入边界和错误数据，来验证操作代码的验证逻辑是否工作正常。测试人员甚至会尝试找出安全漏洞，以确保可疑终端用户无法获取额外的高级别权限。测试验证通过之后，所有和该故事相关的故事点数就可以从当前冲刺的总故事点数上扣除了，同时冲刺燃尽图（展示冲刺进度的图表）也可以进行更新了。这些工件将在后面详细讲解。

- 水平泳道

Scrum面板还可以通过水平泳道来做进一步的划分。团队可以使用水平泳道按照特性来给故事分组，这样团队所有人一眼就能看得出当前正在攻关哪里，也能知道需要缓解哪些瓶颈。

在面板顶部有个特殊的泳道称为快速泳道，可以把所有优先级非常高的任务放在这条水平泳道。通过指定多名团队成员集中在快速泳道上的工作项，来尽快完成这些高优先级的任务；但同时，这也会对其他未完成的任务产生不好的影响。集体攻坚可以确保团队停下手上的工作，以集中力量协同解决一个最高优先级的问题或者任务。这个办法对这种出现最高优先级问题的场景很有用，但是也应该慎用。快速泳道中最常见的工作项就是在已经发布的产品中发现的灾难缺陷。

3. 技术债务

技术债务这个概念值得做进一步的解释。在实现一个故事的过程中，很有可能需要在“最优代码”和“足够好的代码”间做出一些折中以确保不错过最后期限。当然这并不意味着为了赶工期，就可以心甘情愿地容忍（不是积极地鼓励）糟糕的设计，但是当前先简单实现，后面再做改进也是很有实际应用价值的。

- 技术债务的好坏

在项目的生命周期内，债务可能会逐渐累积。之所以使用术语债务（debt）是因为这是一个很好的隐喻，用于描述应该如何看待出现的问题。有些类型的资金债务没有一点错，比如，你准备买辆车但无法一次付清车款，正好有机会选择分十二个月无息贷款购买，虽然此时你有负债了，但从长远看，这个选择负债的决定也是好的。这辆车帮助你赚回了每月要支付的分期款，因为你现在可以开车按时上班了。

当然，有些负债并不是好事。比如你在不清楚如何还款的情况下用信用卡买了一些奢侈品，最终肯定会陷入一个总在以最低限度支付利息的恶性循环中。直到事后，你才会发觉这是一个多么糟糕的债务决定。问题的关键点在于，首先要仔细分辨候选项，然后再决定是要背负债务还是要提前全款付清。

软件开发也有同样的折中。你可以选择暂时先实现一个次优的方案来确保不错过最后期限，或者选择花费更长时间来改善设计但是会错过最后期限。选择没有绝对的对错，只有分析具体情况才能判断引入技术债务是对还是错。

- 技术债务象限图

Martin Fowler是一位非常杰出的敏捷先行者。他提出了技术债务象限图的定义，用于对完成

故事可能需要做出的折中和妥协进行归类。象限图的x和y轴将一个平面划分成四个象限，x轴代表问题“是否在为正当的原因积累技术负债？”，y轴代表问题“是否有其他的替代方案可以避免技术负债？”。

如果你对第一个问题回答“是”，那么你在引入一个有着长期利益的技术债务，因为你能指出增加债务的正当理由而且你也很清楚要注意什么。如果你回答“否”，那么这个债务是有负面后果的，你最好现在就处理掉这个债务，而不应该允许这种债务继续增加。

对于第二个问题，肯定的回答代表你已经考虑了债务的替代方案且决定当前先引入这个技术负债。否定的回答则表明了你并没有为引入的技术债务认真考虑过其他的替代方案。

问题的答案组成了图1-5展示四个可能的场景。

- ❑ 不计后果的，有意的：这种类型的负债是最糟糕的。相当于在说“我没有时间设计”，这代表了一种很不健康的工作环境。出现这样的决定时，就应该给每个成员提出警告：现在的团队不能适应当前状况，而且正在走向注定失败的结局。
- ❑ 不计后果的，无意的：这种类型的债务大多数是因为缺乏经验才引入的。这是由于开发人员不够了解现代软件工程的最佳实践导致的。很像上一个场景，代码很可能是乱糟糟的，但是开发人员不知道还有更好的可供选择的替代方案。解决方案就是再学习，只要开发人员愿意学习，他们就可以停止引入这种类型的技术债务。
- ❑ 谨慎的，无意的：这种类型的债务发生在当你遵循了最佳实践时，却发现仍有更好的方法。“虽然现在没做，但是已经知道以后该怎么做了。”这和前一种场景类似，不同的是，这种情况发生时所有的开发人员的意见是一致的，那就是当时他们都不知道还有更好的方案可供选择。
- ❑ 谨慎的，有意的：这是最令人满意的债务类型。你已经认真考虑到了所有的候选项，也很明白自己正在做什么，也知道为什么要引入这个债务。引入这种类型的债务后，通常会伴随一个这样的决定：“现在先发布，然后再处理那些已经考虑清楚的问题”。

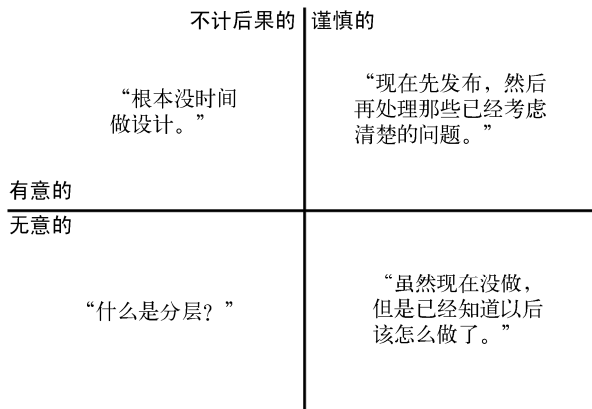


图1-5 正如Martin Fowler所描述的那样，技术债务象限图能让开发人员明白技术债务可以归为四类

- 债务偿还

技术债务和缺陷一样，不能直接持有故事点数，但是即使缺乏直接的激励，作为债务依然是必须要偿还的。最好的方式是给故事再附加一个技术债务卡，并且重构代码以实现新的设计和行为。再从面板中取候选的故事时，就需要检查是否有任何要改动的代码带有技术负债，如果有，就将该技术负债和该故事放在一起处理。

4. 数字Scrum面板

如果不把数字Scrum面板一直挂在墙上，就一定会隐藏很多重要的项目信息。通过开放信息并让整个公司都看到它们，你可以邀请人们对流程提出问题。流程越透明越好，特别是当你在公司第一次引入Scrum实践时。它鼓励获取重要干系人的反馈，你应该很好地引导这些干系人参与到流程中来。

话虽老套，但是人们的确很害怕改变。害怕和焦虑是人本身对未知的自然反应。通过告知人们你在做什么和特定图表表达的意义（以及为什么墙上会有很多的索引卡），你能将协作沟通的积极态度推广给了大家。此外，给一个外行解释这些信息对你也是有好处的，因为你的解释过程或许会让你自身对整个流程有了更清楚的理解。

所有的工具中，最好的总是那些易接触和没有约束的工具。这些工具会被频繁自由地使用，而当一个工具开始变得有些不便使用时，它也逐渐会被抛弃，那些原本被经常使用却未能与时俱进的工具也会迅速地被人遗忘。

5. 完成的定义

所有项目都需要完成的定义（Definition of Done, DoD）。为了验收判断是否完成，所有用户故事都必须符合清晰的完成的定义。下面是开发人员经常提及的，你有听到过多少次呢？

“做完了，不过我还要再做一下测试……”

“做完了，但是我刚刚又发现了一个问题，可能需要修复……”

“做完了，只是我觉得设计还不完美，我计划改一下接口……”

我过去也用过这些托辞。如果真的做完了，那就说“做完了”，不需要再附加任何说明、条件或解释。上面这几个例子实际上代表着开发人员还需要多一点时间，因为他们原先的估算不够好或者遇到某个未预料到的问题。所有人必须认同并且至始至终遵守完成的定义。如果一个故事不符合标准，它不能算是真正完成了。只有符合完成的定义的用户故事才可以标记为完成状态。

完成的定义都包含什么？这完全取决于你、你的团队，以及你想要的质量保证流程的严格程度。无论如何，你可以参考下面这个常用的完成的定义。

为了将某个用户故事标记为完成状态，必须确保以下事项。

- ☐ 对所有代码的成功和失败路径都做了完整的单元测试，并且通过了所有测试。
- ☐ 所有代码都无误地提交到了持续集成系统中，编译和构建是成功的，并且通过了所有测试。
- ☐ 通过了验收标准和产品负责人的验收。
- ☐ 不在同一个用户故事下的开发者相互实施了代码评审。
- ☐ 只撰写了适量的用于沟通的文档。


❑ 拒绝了那些不计后果的技术债务。

在上面几条的基础上，你可以任意删除、修改或者增加定义条款，但是请务必确保定义是严谨的。如果一个用户故事无法满足所有验收标准，你要么确保这个故事仍然能够满足所有标准，要么把要求比较高的标准从完成的定义中完全删除。比如，如果你觉得交叉代码评审太死板或不必要，那就不要把它包含在你的完成的定义中。

1.3.2 图表和度量标准

在Scrum项目中，有多个图表可以用来监测项目进度。它们能够表现项目的健康状态和进展历史，还可以预测后期可能取得的成果。这些图表要在Scrum面板上有着醒目的位置，尺寸大小要保证较远的时候也能看得清。这种公共的展示方式在向所有团队成员暗示，这些图表不是什么秘密，也不是用来监测项目管理的消耗情况的；相反地，这些图表很直接地展示了项目进度的监测方式。这种公共的展示方式能够告诉所有参与项目的人员，创建这些图表不是为了监控和提高效率，而是用来体现和诊断整个项目中存在的问题。

此外，请尽量避免评测团队中个人层面的任何事情，比如单个开发人员争取的故事点数。因为这会错误地引导团队个人把自己的进度看得比整个团队和项目的进度更重要。一旦有了这些个人考核标准，开发人员会迅速开始为这些个人目标行动起来，为了不在考核结束时被评为不合格，他们会尽量去独占大型的用户故事以争取这些故事的所有点数。所以，请警惕你对这种个人目标设置的奖惩机制。

 **警告** 请谨慎选择你要监测的事项，因为你的选择会产生“观察者效应”。比如说，对于某些标准而言，在开始监测之前必须先调整测量的对象。再比如，要测量汽车的胎压，就要先给轮胎放点气以改变胎压。人性也存在同样的情况，当团队成员知道他们要被考核时，他们将会竭尽全力去改善他们的数据表现来达到绩效要求。这并不是说团队成员全都是利己主义者，而是说当团队成员认识到故事点数是他们的一个考核标准时，每个人都很可能争相选择那些高性价比的用户故事（也就是说，实际工作量一样但故事点数更多）。这里需要使用三角测量法（1.4.5节中会有讲解）来调整估算工作量和实际工作量的差距。

1. 故事点数

故事点数的意图是激励开发团队为每个冲刺的发布增加商业价值。整个开发团队会在冲刺计划会议（本章后面专门有一节讲解冲刺计划会议）期间讨论并为承诺要完成的用户故事分配故事点数。一个用户故事点数用来衡量实现该用户故事定义行为所需的工作量，其中包括了整个软件生命周期的所有阶段的工作量：需求分析、方案设计、含单元测试的代码实现、测试验收，还有部署实验。尽管每个故事都应当足够小以确保能够在一个冲刺中完成，但是实际故事的规模变化还是很大的。

实现最小的“一个点数的故事”只需要最小的工作量。有个很有意思但也很重要的事实是，

故事点数值只对得出该估算点数值的开发团队才有意义。因为技能和经验值的不同，一个团队的一个点数的故事对另外一个团队来说很有可能变为三个点数的故事。经过多个冲刺后，不同团队对于同一个故事的工作量估算慢慢才会变得接近了。

有一点需要强调的是，故事点数不是用来表达工作量的绝对时长（天数、小时数或者其他时间单位的测量值）。故事点数只是根据以往实际时长范围对工作量进行粗略的估算，如图1-6所示。表中的垂直线表示估算的时间范围，附在垂直线上的水平短粗线表示相应点数的故事实际耗费的时长。

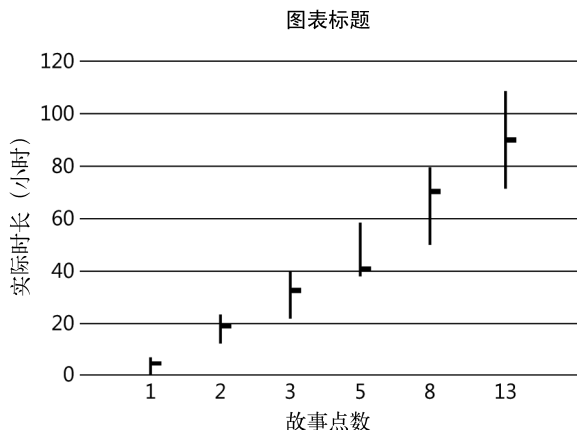


图1-6 最小/最大/平均值表，展示了估算工作量与实际工作量之间的关系

从图1-6中还可以看出：大的用户故事的实际工作量时长范围更大，也应该更难准确预计实际工作量的时长。

2. 速度

经过多个冲刺后，可以开始计算已经完成的用户故事的平均开发速度了。假设一个团队已经完成了三个冲刺，分别完成了8、12和11个故事点数。也就是说，三个冲刺总共完成了31个故事点数，平均每个冲刺完成10个故事点数。“每个冲刺10个故事点数”就是该团队的开发速度，可以按照以下两种方法使用速度值。

第一种方法，团队的速度可以作为团队下一个冲刺承诺要完成的故事点数的上限。如果团队每个冲刺平均能完成10个故事点数，那么在单个冲刺中承诺高于10个故事点数并不只代表太过乐观，还意味着要准备接受更高的失败概率。设置一个可完成的目标，然后完成，甚至超额完成，这总比设置一个不现实的目标然后失败要好。如果团队承诺了10个故事点数，最终实际完成了11，那么新的团队速度就是11： $(12+11+11)/3$ 。这就是Scrum流程的反馈机制。

第二种方法，团队可以使用速度分析交付存在的问题。如果团队速度在某个冲刺中明显变慢，这就表明冲刺中有糟糕的事情发生了，需要尽快解决。有可能是故事规模太大并且作出了错误的估算，导致偏差太大，这种大型故事应该需要多个冲刺才可以真正完成；也有可能是因为太多的

关键团队成员同时休假或者生病，从而导致进度变慢；另外一种可能的情况是，团队花费了太多的时间去重构现有的代码，而没有专注在新特性的实现上。无论什么原因，25%以内的速度降幅还不是很严重，但是它已经暗示很可能会出现問題，一旦問題出现，团队应该及时处理。如果速度每周都有下降，而且减速幅度越来越大，这一定表明已经出现了严重的问题，很有可能就是部分代码无法快速适应变化导致的；而本书的主旨就是为了帮助你解决这个问题。

3. 冲刺燃尽图

每个冲刺开始的时候，都要在Scrum面板上创建一个二维平面图，其中，y轴表示总的故事点数，x轴表示工作天数。如图1-7所示，笔直的对角线也称为最吻合线（line of best fit），它展示了最理想的冲刺进度中故事点数随着时间扣除的过程。

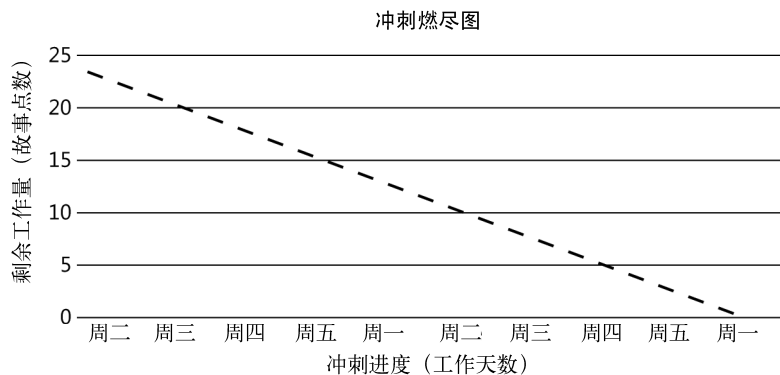


图1-7 一个刚开始的冲刺燃尽图。其中的对角线展示了最吻合目标的冲刺过程（在这个例子中，目标是争取23个故事点数）

在每日站立会议上，会从总的故事点数中扣除所有从已经完成的故事上争取的故事点数。图1-8展示了为达到冲刺目标，实际冲刺过程就是一条与最吻合线有偏差的曲线。

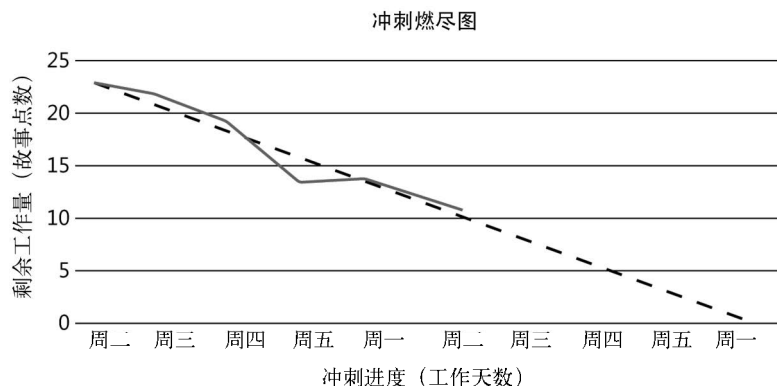


图1-8 一个进行到一半的冲刺燃尽图。在这个例子中，团队正在努力让实际冲刺过程贴合“完美路径”，尽管第一个周五和周一期间并没有任何进展

用不同的颜色绘制实际进度曲线和最吻合线，可以让两者有视觉上的差异。在冲刺过程中，如果实际进度曲线位于最吻合线的上部，这表明有问题出现了，团队能够完成的工作量比计划的要少。相反，如果实际进度曲线位于最吻合线的下方，这表示当前项目进度比计划要快。在整个冲刺过程中，有时会出现实际进度曲线在最吻合线上下微微波动的现象，这是正常的。但是如果实际进度曲线相对于最吻合线的波动过大，那就需要认真查找原因了。

当冲刺需要在固定时间段内完成固定的工作量时，使用燃尽图对项目是很有帮助的。燃尽图中，曲线是无法位于x轴下方的，因为当y等于0时，代表团队已经完成了所有的工作。

4. 特性燃耗图

在一个冲刺中，冲刺燃尽图用于从故事层面追踪实际的进度，而特性燃耗图则是用来追踪特性完成的进度。在每个冲刺结束时，可能会完整实现一个新的特性。特性燃耗图的最大好处就是能防止冒充交付特性，因为图中的曲线根本就没有变化。随着时间的推移，特性燃耗图中的曲线会呈现线性增长，最理想的情况是一直没有出现过平行线。图1-9展示了一个很好的特性燃耗图。

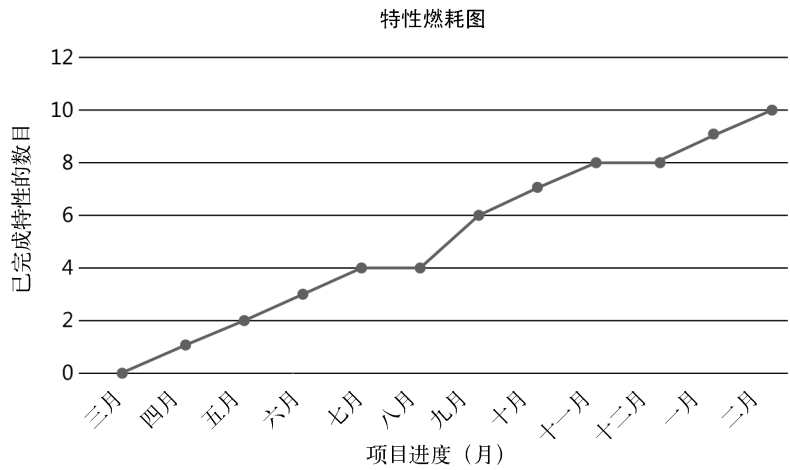


图1-9 这个特性燃耗图展示了一个健康的项目在一年中持续进展的过程

图1-9中曲线坡度可能有些平缓，但它表明了团队的开发节奏很好，正在以一种可预期的速度持续交付特性。虽然与完美的直线有点偏离，但是这很正常，没什么可担心的。

相反，图1-10中的特性燃耗图已经表明出现了开发问题。在开始阶段，团队的状态很好，快速地交付了很多特性，但是后期整整八个月只交付了两个特性，整个团队完全陷入了泥沼。

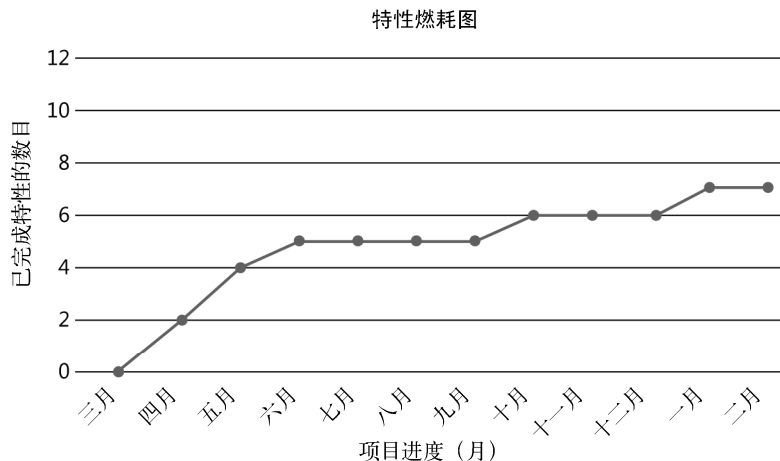


图1-10 这个特性燃耗图展示了一个停滞的项目在一年中缓慢进展的过程

这个问题很清楚：代码无法适应需求的变化。曲线前段的点可能表明团队没有做单元测试，没有做分层架构或者没有采用其他的最佳实践。通过忽视了这些其实很必要的行动，团队设法早早完成了更多的特性。但是，代码逐渐变得愈发臃肿和混乱，开发进度会显著变慢，很长时间才能完整交付一个特性。这种陷于停滞的进度很可能是由于大量累积的缺陷和糟糕的技术债务引起的。如果这个项目继续这种糟糕的状况，最终更好的选择可能是重新开始。长期来看，努力让项目重回正轨的重构工作将会是非常值得的。当然，如果能早早发现问题，团队和项目就能够重新恢复。尽管如此，还是建议从项目一开始就正确地应用各种敏捷开发的优秀实践，而不是等进行中的项目（brownfield project）变得一团糟的时候才引入这些实践。



注意 brownfield是指项目已经处于进行中的状态。与它相反的是greenfield项目，代表新的还未启动的项目。这两个术语都是从建筑产业借鉴而来的。

1.3.3 积压工作

积压工作是一个列表，其中包含了一些需要处理的待定工作项。这些工作项在合适的时间会从积压工作中取出，然后处理直至完成。每个工作项都有自己的优先级和工作量估算值，整个积压工作列表是按照优先级高低和工作量大小进行排序的。

Scrum过程中维护了两个积压工作：产品积压工作和冲刺积压工作，它们分别有着自己特有的用途。

1. 产品积压工作

在产品生命周期内，产品积压工作上总是包含一些待实现的特性。因为产品积压工作中的特

性还没有被取出并放入到冲刺当中，所以开发团队实际上并不会处理产品积压工作上的特性。尽管如此，开发团队（或者团队的代表）仍需要花时间去评估这些特性的工作量。花时间评估特性工作量也有助于对这些产品积压工作上的特性进行排序，以此保证产品积压工作上所有的特性总是排好序的。

每个特性业务价值的高低决定了各自的优先级别的高低。特性的业务价值是由产品积压工作的所有者（即产品负责人）来决定的。产品负责人会向开发团队描述具体业务并为特性定义明确的业务价值。为了挖掘和定义具体特性内在的业务价值，产品负责人必须有丰富的业务知识和工作经验。当产品积压工作上的两个特性的业务价值一样重要时，特性所需工作量的大小会决定二者的优先级。假设有两个高业务价值的特性，相比较其中一个所需工作量要大一些，那么应该首先实现工作量小的特性。因为工作量相对较小的特性，风险会小一些，实现周期也会短一些，相应的投资回报率（Return On Investment, ROI）也会高一些。

当业务需要发布产品的新版本时，有两种方式可以根据产品积压工作作为这个发布选择最有价值的特性。第一种是业务需求已经确定了最终的发布日期，需要根据估算的工作量和可用时间从产品积压工作中选择可能完成的特性。第二种是业务需求已经确定了最终的发布特性，只需要根据从产品积压工作中选择的目标特性的估算给出预计的发布日期。

除了特性外，产品积压工作还可以包含那些必须要修复但仍未进入任何冲刺的缺陷。与特性一样，也可以给缺陷指定业务价值。此外，由于缺陷的根本原因还不清楚，因此在确定缺陷修复工作量之前，可能需要时间对它们进行估算。

产品积压工作也应该像其他敏捷文档一样保持开放。所有人都应该能够看到产品积压工作文档，可以在产品开发期间贡献想法，提出建议或指出问题。同样，产品积压工作需要时刻展示权威的、真实的项目状态。很多时候，错误的决定都源自不准确的信息，根据过期的产品积压工作制定的关键发布计划执行起来肯定会是一团糟。

2. 冲刺积压工作

冲刺积压工作包含了所有在即将开始的冲刺中承诺要完成的用户故事。冲刺开始的时候，团队会根据当前开发速度和待开发故事的工作量估算为冲刺选择足量的工作。选定故事后，团队可以开始将所选故事分解为任务，并使用实际时间单位小时来度量这些任务的工作量。最后，团队中每个开发人员再为自己选取足量的任务。

开发团队全权负责冲刺积压工作及其工作量估算。其他人没有权力给冲刺积压工作增加工作项，也不可以为开发团队指定工作量估算。尽管详细的冲刺积压工作是由开发团队单独定义，但是它必须要基于有优先级排序的产品积压工作。

1.4 冲刺

Scrum项目的迭代被称为冲刺。一个冲刺周期最短一周，最长四周，最常见的是两周。如果冲刺周期太短，开发团队没有足够时间来完成既定目标，太长又容易让团队失去工作焦点。

冲刺通常有数字索引，从冲刺0开始。冲刺0的目的是让团队准备开发环境以及在后续真正的

冲刺开始前召开一些主要的计划会议。冲刺0很可能没有故事点数，但是冲刺0中完成的准备工作会对过渡到后续Scrum冲刺有很多好处。

大家很容易就会想按照工作周来安排冲刺，也就是说从周一开始，到周五结束。这种安排的问题是冲刺回顾（后面章节会简短讲解）会需要大量的会议时间，但是没有人喜欢在周五的下午无精打采地坐在会议室开半天的会。因为通常周五很多人都打算早早离开办公室，而且周末马上到了，大家精力都不够集中，工作效率很可能会下降。同样地，也没有人喜欢每周上班的第一天就开半天的会。因此，最好的安排是在周中（周二、周三或者周四）启动冲刺以避免上面提及的问题。

下面按照出现顺序讲解冲刺中出现的所有会议。

1.4.1 发布计划会议

软件的发布计划必须在冲刺开始前就准备好。为此，用户和产品负责人需要在发布计划会议上决定发布日期以及该发布所包含特性的优先级排序和工作量估算。

1. 特性工作量估算

无论特性的规模多大，都可能会有大量的工作要做。因此，任何尝试准确预计所需工作量的效果都有可能大打折扣。鉴于这个原因，特性工作量的估算可以使用常见的T恤码号来计量。

☐ 特大码（XL）

☐ 大码（L）

☐ 中码（M）

☐ 小码（S）

☐ 特小码（XS）

2. 特性优先级

特性的优先级排序也很重要，因为有时很难预计一个实际的发布最终能包含多少个特性。在一个特定的发布中，需要给每个特性指定下面三种优先级中的一种。

☐ 必需的（Required）

☐ 首选的（Preferred）

☐ 想要的（Desired）

所有必需的特性构成了最小可行发布。首选的特性是在冲刺时间有盈余的情况下才需要处理。想要的特性优先级是最低的，它们都只是一些锦上添花的特性（当然，用户肯定会想如果能在发布中包含这些特性那也挺好的）。

此外，业务干系人要给所有特性编号，以便开发团队能够按照明确的优先级顺序实现每个特性。

1.4.2 冲刺计划会议

冲刺计划会议的输出应该是所有承诺要在该冲刺中完成的用户故事的估算。与Scrum的其他

流程一样，用户故事估算流程也有很多种变体。本节主要讨论计划扑克和亲和估算，前者是很常见的一种估算讨论的方式，后者是一种快速估算故事相对大小的方法。当用户故事数量比较多的时候，亲和估算会更合适一些。在一个单独的冲刺内，当要估算的用户故事数目不多时，可以使用计划扑克，而当故事数量太大或者时间太短时，就可以使用亲和估算。

1. 计划扑克

计划扑克讨论环节需要整个开发团队参与，包括业务分析人员、开发人员以及测试分析人员，此外还包括Scrum主管以及产品负责人。开始讨论时，首先对产品积压工作上每个用户故事作一些详细的介绍，然后要求每个人用故事点数来给故事的大小投票。

为了避免故事点数都比较接近的情况，最好事先对可选点数进行限制。比如，常见的一组故事点数是经过修改的斐波那契数列：1、2、3、5、8、13、20、40和100。无论选择何种点数序列，总的故事点数的个数必须是有限的，而且点数之间的间隔应该逐渐变大。0作为最小的点数值，可以加到序列中来表达“无需任何工作”，因为有些故事的工作量几乎是可以忽略不计的。序列中最大的点数值用来表达该故事规模太大，无法在一个冲刺内完成，在真正进入开发前需要作进一步的竖切。

投票时可以使用真正的扑克牌，不过，利用闲置的索引卡甚至直接写上数字的白纸也不错。投票时，所有人必须同时展示出自己卡片上的故事点数，这样可以避免在确定故事点数时相互干扰。当然投票结果不是每次都可以达成一致的，很多时候团队成员间会出现较大的分歧。这种现象非常正常，因为总会有个别人出现估算偏差的，他们需要在达成一致的目标下重新考量自己的估算。比如，当平均点数是8而有人却投了1时，会议主持者应该（礼貌地）要求该投票人解释他认为所投故事工作量这么小的原因。同样地，超过平均值过多的投票者也需要给大家解释他们认为工作量很大的原因。整个讨论都是为了得出一个整个团队都认可的、实现该用户故事所需的故事点数。

当投票者讲述了理由之后，可能需要重新投票，因为其他人也许发现自己的估算过大或过小了，而刚才偏离平均值的人才是正确的。最终，所有人会达成一致并得到合适的故事点数。当已经估算的用户故事点数之和达到团队当前的速度时，就可以停止讨论和评估剩余的故事了，因为每个冲刺能够选择的最大工作量应该不大于团队当前的开发速度。

避免出现帕金森定律所揭示的现象

帕金森定律指出：

“大多数情况下，人们会应付工作直至耗尽所有可用的时间。”

当你不使用实际时间单位来估算用户故事的时候，出现帕金森定律所说现象的几率也就变小了。团队从始至终都应该聚焦在尽快完成故事的行动上，也就是说尽快满足完成的定义。

2. 亲和估算

亲和估算的提出是为了解决计划扑克的局限性，因为如果故事数目很大时，使用计划扑克方

法达成估算需要耗费大量的时间。使用亲和估算方法，团队无需逐个讨论每个故事，只需要从产品积压工作中提取两个优先级最高的故事并对比它们的工作量大小，然后将小故事的卡片放在桌子的左侧，大故事的卡片放在桌子的右侧。

然后，团队成员再从产品积压工作中取出优先级最高的一个故事，并根据该故事的大小把它放置在已有的小或大故事的周边。如果放在小故事左侧，代表更小；放在大故事右侧，代表更大。如果放在某个故事（不论大小）上面，代表与该故事大小基本一致。如果放置于两个故事之间，则代表着大小在二者之间。如此反复，直至觉得当前冲刺的故事工作量已经达到饱和。

这种方法按照相对大小对故事进行分组，团队可以按照计划扑克方法中提及的经过修改的斐波那契数列从左至右地给所有故事组分配故事点数。如果要估算的故事很多或没有时间逐个估算，此时亲和估算就是一个很好的办法，因为它能根据故事的相对大小很快得到所有故事的点数估算。

1.4.3 每日站立会议

尽管有几个Scrum会议会持续好几个小时，但是平时几乎感觉不到Scrum流程在运作，只有在每日Scrum会议上才能看到，这个会议也被称之为“每日站立会议”。

召开这个会议的时候，所有团队成员都要围着Scrum面板站立，每个人都要面对着其余团队成员发言。每日站立会议的持续时间最长不应超过十五分钟。为了保持会议的高效，每个人都应该回答下面三个问题。

- ❑ 昨天做了什么？
- ❑ 今天计划做什么？
- ❑ 遇到了什么障碍？

每日站立会议的重点是给所有团队成员公开昨天的实际进度以及今天的预计进度。在讲述昨天完成了什么的时候，你可以自由更新Scrum面板，把卡片从一个泳道挪到另外一个泳道，或把小头像从一个卡片挪到另外一个卡片上，同时概要地讲述昨天的工作情况和感受。如果当前手头上没有任务了，你需要在会议上告知Scrum主管并申请新的工作项。障碍是指所有可能影响你达成今天目标的事情。因为你需要在明天的站立会议上声明你今天做的事情，所以记录下任何有可能影响你完成计划的事情非常重要。障碍有可能与工作内容直接相关，比如“如果网络还像昨天那样断开，我将无法继续今天的工作”，或者只是与工作无关的个人事宜，比如“我下午两点预约去看牙医，所以我很可能无法完成计划”。不论障碍是什么，Scrum主管都应该记录下来以了解所有人手头上的故事进度和状态。

表情日历（niko-niko calendar）

表情日历有时也称为“情绪面板”（日语中，niko-niko的意思是“笑脸”），它作为一个晴雨表能很好地反应团队成员在冲刺期间对进度的感受。表情日历画在Scrum面板上，行名是冲刺的工作日，列名是成员名字，如图1-11所示。

	周一	周二	周三	周四	周五	周一	周二	周三	周四	周五
John	😊	😊	😊	😊	😊					
Bob	😐	😐	😊	😐	😞					
Alice	😞	😐	😞	😞	😐					
Mark	😞	😞	😞	😞	😞					

图1-11 表情日历可以清楚地展现出了成员冲刺状态的好坏

在每日站立会议上,每个人都要在自己昨天的日历格子上放置一个贴纸。这种贴纸有绿、黄、红三种颜色,分别代表昨天的工作状态:好、一般、糟糕。表情日历能清楚地展示出,有些成员因为进度一直不好需要帮助,只是他们可能不想主动提出来。

表情日历只是另外一个有助于促进反馈效率的测量方法。如果所有团队成员持续地对他们手头的工作提不起精神,此时可能需要提振一下团队的士气了。如果只有团队中某个成员一直感觉很糟糕,而其他成员状态都还不错,这说明这个成员进度落后太多或者根本不喜欢自己手头的任务。最糟糕的情况是,冲刺快要结束时整个团队还自我感觉良好,但实际上代码一团糟,客户都在敲门要账了,这只能说此时的团队已经根本不在乎项目成败了。

每个人发完言后,站立会议就结束了。Scrum主管要特别留意的是,不要让大家的发言跑题,因为大家会非常容易陷入问题的讨论当中。如果有人提及在代码中看到了有着奇怪有趣表现的问题时,所有开发人员会立即不由自主地开始猜测可能的原因了。要清楚会议中有其他与会者,难道测试分析人员真的需要旁听有关Microsoft Visual Studio几乎使用了机器的所有可用内存的讨论吗?当然不需要。正确的做法是,Scrum主管先记录问题,会后再组织相关的人一起开展讨论。

1.4.4 冲刺演示会议

在冲刺日历上,冲刺演示是一个关键的会议。这个会议要在真实的产品环境下演示所有已经完成的故事,也就是说那些符合完成定义的故事。开发团队所有人员必须到场,也可以邀请其他干系人,比如管理层和销售团队的代表。此外,任何对项目进展感兴趣的人都可以自由观摩演示。所有项目都应该持有这种积极开放的态度。

从Scrum面板上整理好所有已经完成的用户故事,逐个解释每个故事的功能范围以及该故事为整个项目的业务所提供的价值。故事所属特性实现之后的表现就是应用行为上有了变化。接下来就需要在实际系统上部署产品来为客户展示这些变化了的行为。要欢迎与会者的提问,但是要防止出现太多偏离主题或无关的讨论。要保持整个讨论始终聚焦在正在展示的用户故事上,并在演示结束后再与提出问题的人作进一步深入讨论。会上收集到的所有改进的建议和意见都要加入到产品积压工作中,这样才可以作进一步排序和计划。有时也会发现演示会议上收集到的某些改进建议的优先级会非常高,但这种情况在实际项目中难得一见。

不要害怕演示,演示肯定会激励后续的进度。没有人会在什么都没做的情况下就去中止演示,但是切忌绕开完成的定义去做演示。坦诚公开进度,就不必隐藏任何问题。演示出现问题时,根据图表和度量标准来解释可能的原因即可。

在演示之前的某个时间点冻结代码也是个好主意,这样可以防止开发人员最后时间为了多争取一些故事点数而仓促修改。会议之前要预留合理的时间来搭建演示环境和进行预演,不要为了眼前的一些改进而稀里糊涂地给代码引入技术负债。

自律是一种始终能够拒绝眼前诱惑而选择长期利益的能力。

——Mike Alexander, 健身专家

1.4.5 冲刺回顾会议

冲刺演示会议后,需要对整个迭代过程做复盘来评估整个冲刺的成功程度。有些团队成员可能在这个冲刺中取得了很好的战果,而有些人却可能经历了一个糟糕的冲刺。冲刺回顾会议有助于团队从冲刺过程中整理出那些需要保持的好行为以及那些应该避免的不好行为。冲刺回顾会议的输出文档不应该在会后就被束之高阁。应该在下一个冲刺结束时用它作对比,看看哪些需要的改进完成了,哪些错误没有再反复。

会议期间,团队应该回答以下问题。

- ☐ 做得好的标准是什么?
- ☐ 做得差的标准是什么?
- ☐ 需要开始做什么?
- ☐ 需要停止做什么?
- ☐ 需要继续做什么?
- ☐ 是否遇到任何意料之外的事情?

首先从积极的方面开始,让每个团队成员都说说冲刺中哪些地方做得比较好。大家也许对冲刺进度很满意,或者对工作质量很自豪。

接下来,每个团队成员需要讲讲冲刺中哪些地方做得比较差。也许是有些任务延迟交付了,因为任务实际比评估时看起来要困难得多。无论是什么问题,肯定都可以找到解决方案。只要对事不对人,所有有关问题的畅所欲言都没错。团队成员之间不能出现指责,要以恰当的方式建设性地指出问题。讨论问题的目标只是为了改善流程和产品。

团队很可能有一些事情没做但应该在后面加入到流程中。也许是代码还没有正式的单元测试,团队成员都认为应该在现阶段引入单元测试。Scrum主管应该在回顾会议上及时记录所有的建议以便在后期采取行动。

同样地,团队也很可能会发现有些事情不能再继续。很典型的例子就是,在会议上讨论不要跑题,不要在开发泳道已经饱满的情况下还加入新的故事。后面这个问题很常见,可以通过给特定的泳道增加容量限定来解决。通过强制团队最多同时开发三个故事,可以鼓励团队尽快完成已经开始的工作,而不是再开始一些新的工作项。

有些重复性的工作做好了也会有收益。如果因为准备充分取得了不错的冲刺演示效果，那就应该记下来并在以后继续这个好习惯。让人非常意外的是，通常人们会很快忘掉好习惯，而坏习惯却很容易如影随行。

最后，团队成员要一起回忆冲刺期间是否遇到了一些意料之外的事情，不论好或坏。对于不好的意外事件，要讨论出一个行动计划以避免将来出现同样的问题，而好的意外事件能让团队知道有些行为是好的习惯并应该继续坚持。

最后团队成员还需要一起讨论并为下一个冲刺的候选工作项排列优先级。冲刺回顾会议输出的文档不应该在会后就抛之脑后，而是应该根据文档记录内容尽快采取行动。

故事点数三角测量法

冲刺期间，团队成员可能发现有些用户故事的实际工作量与冲刺计划会议上的估算有些或多或少的偏差。那么在冲刺回顾会议结束前，花上5到10分钟，根据故事实际工作量，应用故事点数三角测量法来重新评估用户故事的估算是有好处的。

若干个冲刺后，团队会得到一组统计数据，可以对比每个故事实际工作量和估算点数。举个例子，也许团队会得到如表1-1所示的数据。

表1-1 一个假定项目的实际工作量统计数据和用户故事估算数据的对比表格

故事点数	平均实际工作量（小时）	最小实际工作量（小时）	最大实际工作量（小时）
1	5.5	1	19
2	9.5	2	23
3	17	7.5	40
5	36	20	76
8	56	40	86
13	88	68	154

根据上面表格的统计数据，如果一个用户故事估算为一个点数，而实际花费了六十个小时，那么这个故事很可能应该是一个八个点数的故事。如果开发情况并没有好转（比如缺席的开发人员还没有回来），那么就完全可以认为现有的一个点数的估算是不正确的。相反，如果估算这个故事为八个点数，那么团队的工作速度不会受到影响，团队能承诺的故事总数也不会减少。^①

此外，只需要关注那些明显偏离的故事估算。因为如果一个点数的故事实际工作量在两个或三个点数的实际工作量范围内时，它的实际工作量也不太可能再发生更大的偏差。

1.4.6 Scrum 日历

为了清楚起见，表1-2中的日历展示了一个冲刺期间所有典型的Scrum会议。

① 假设原有一个点数的故事是三个开发人员一起做，结果其中两个人因事缺席后，剩下的一个人最终没有在一个冲刺中完成该故事，那么这个冲刺团队承诺的故事数就会少一个。结合前几个冲刺完成的故事数目求取平均值得到的团队速度也会下降。作者在这里想描述这种因主观或客观原因导致估算不正确的情况，上面这种实际和估算数据对比的统计表格可以给团队后续的估算一些指导和参考，以减少这种偏差对项目的影响。——译者注

表1-2 一个假定项目的Scrum日历，用来组织一个冲刺中的所有会议

日期（2013年4月）	时 间	会议类型	与 会 者
4月2日，周二	13:00～15:30	冲刺计划会议	开发团队 产品负责人
4月3日，周三	09:30～09:45	每日站立会议	开发团队
4月4日，周四	09:30～09:45	每日站立会议	开发团队
4月5日，周五	09:30～09:45	每日站立会议	开发团队
4月8日，周一	09:30～09:45	每日站立会议	开发团队
4月9日，周二	09:30～09:45	每日站立会议	开发团队
4月10日，周三	09:30～09:45	每日站立会议	开发团队
4月11日，周四	09:30～09:45	每日站立会议	开发团队
4月12日，周五	09:30～09:45	每日站立会议	开发团队
4月15日，周一	09:30～09:45	每日站立会议	开发团队
4月16日，周二	10:00～11:20	冲刺演示会议	任何人
	11:30～12:00	冲刺回顾会议	开发团队
	13:00～15:30	冲刺计划会议	开发团队 产品负责人

通过上面的Scrum日历可以看出，一个冲刺的演示和回顾会议以及下一个冲刺的计划会议几乎需要一整天来完成。这一天也被称为冲刺交接日，用来维护冲刺的关注点，有时候这种交接也会分布在前后两天：一个下午和第二天的上午。上面表格中，如果冲刺演示和回顾会议被安排到周二的下午，新冲刺的计划会议就要移到周三的上午。

日历中另外值得注意的一点是，每日站立会议的时间问题。如果安排的时间太早，与会者可能会因堵车或者其他事由迟到。同样，如果安排的时间太晚，要把与会者从紧张的工作中拉出来也会不容易。

可以把这些会议都加入到Microsoft Outlook或其他日历应用中，并将所有相关人指定为与会者，这样就可以保证所有人都能看到相同的会议日程以及提醒。

1.5 Scrum 和敏捷的问题

敏捷流程并不是保证能够挽救所有失败项目的仙丹良药。任何软件开发流程的目标都是每次都能保证成功交付软件，但是软件依然需要编写代码才能实现。任何文档都无法掩盖软件产品是基于可工作代码的事实。

本书的目的是为开发人员讲解如何创建自适应的软件方案。这意味着这个方案要能很好地适应所有软件都遇到的各种变更。寄希望于方案的第一次尝试就能满足客户的所有需求是不现实的，因此变更是无法避免的。敏捷流程（也包括Scrum）的目标都是拥抱变化，并寻求各种方式来确保客户能够对已经实现的软件行为提出变更。没有这些流程，用户可能只好不得不接受一个无法达成目标的不合格方案。