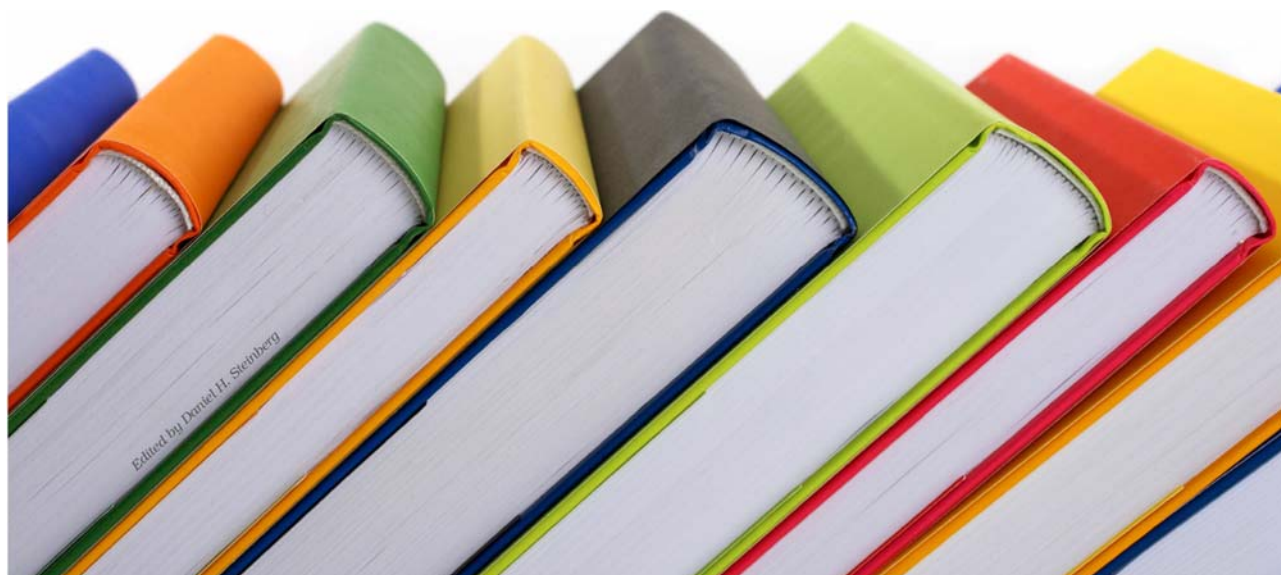


来自软件界思想领袖们的经验心得

ThoughtWorks文集

精选版



ThoughtWorks 公司 著

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

InfoQ中文站
www.infoq.com/cn

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

<http://infoq.com/cn/minibooks/thoughtworks-anthology>

QCon

旧金山 • 伦敦 • 北京 • 东京

QCon全球企业开发大会（北京站）
每年四月，我们一同期待！

大会整体评价

总结而言，这次大会是相当成功的，一次成功的大会不能缺少的两个要素：知名的嘉宾和精彩的Topics，无疑QCon北京很好地把握了这两个要素。

----- 淘宝网架构师、OSGi China User Group的负责人BlueDavy（林昊）

The first QCon Beijing conference concluded successfully, I was amazed by the overwhelming attendance. The Beijing debut is probably the biggest QCon gathering globally. I was also amazed by how efficient infoQ run the conference.

----- FreeWheel创始人/CTO、DoubleClick前CTO（于晶纯）

出乎很多人意料的是，本次大会门票竟然全部提前售罄。即便如此，到会议现场购票的人仍然络绎不绝。究其原因，前沿的技术话题和高水平的讲师是最大的吸引力。

----- Joyent中国区技术经理
《程序员》杂志前主编（郑柯）

往届讲师及评价



Martin Fowler

敏捷宣言缔造者，Thoughtworks首席科学家

来自CnBlogs网友Daniel的评价：Martin Fowler的两个演讲，关于DSL的和Ruby的，坚定了我以后的技术方向：Polyglot+Polyparadigm。



Randy Shoup

eBay高级架构师

来自手机之家架构师徐超前的评价：非常不错，都是实战的总结。InfoQ上发表过《可伸缩性最佳实践：来自eBay的经验》。Randy在不断用心的数据补充和完善该话题。



洪强宁

豆瓣网首席架构师

来自DBAnotes.net博主冯大辉的评价：第一时间看到豆瓣@hongqn大侠在QCon上做分享的PPT，击节赞叹！这是今年看到的最好的一份PPT。极有参考价值。如果非要形容一下的话，那就是：牛！



岳旭强

淘宝网资深架构师

来自网友Rocky_rup的评价：岳旭强在介绍淘宝网应用Java的历程汇总，激起我的共鸣，我在心中默默感叹：原来大家在实践中演化系统架构的经历是如此的相似！



Dylan

Dojo Tookit创始人



程立

支付宝首席架构师

.....



www.QConBeijing.com



QCon@cn.infoq.com

前言

在帮助客户实施敏捷的过程中，ThoughtWorkers 常被问到一个问题：有没有一套标准的“敏捷模板”可供快速入门之用？

作为一种强调持续改进的方法学，自然不会有一套放诸四海而皆准的“标准流程”；但对于希望采用敏捷方法的组织和个人而言，若有一组普遍适用的最佳实践作为基础，便能少走许多弯路，以期事半功倍之效。

摆在你面前的，正是这样一本“敏捷入门手册”。

本迷你书从《ThoughtWorks 文集》的 13 篇文章精选 5 篇编撰成集。这几篇文章有一个共同点：它们介绍的是一些最根本、最易施行、又最能立竿见影的敏捷实践。藉由这几篇各自独立而又相互关联的文章，我们希望帮助读者从持续集成和测试入手，建立行之有效的项目健康保障体系，并掌握必要的面向对象编程和重构技能，从而切实提升软件质量，并为更进一步的改进打下坚实基础。

作为项目自动化和信息发布的核心，持续集成在敏捷实践中具有举足轻重的地位。Dave Farley 的文章介绍了关于“一键部署”的概念框架，将持续集成的覆盖范围由开发阶段延伸至整个软件生命周期。在此基础上，“项目生命体征”一文介绍了如何运用沟通技巧来获悉项目的健康情况。文中给出了一些适用于敏捷项目的度量标准，并探讨了如何有效地展现度量结果。

以 QA 咨询师的视角，Kristan Vingrys 介绍了敏捷项目的测试生命周期及各种测试手段，并将其与瀑布式项目中的测试活动加以比对，使之更容易为传统组织的 QA 和测试人员所理解。James Bull 则在“实用主义的性能测试”一文中提出了一套切实可行的性能测试策略：他不仅指出应该从项目之初就定义性能需求，而且对“如何获取有用的性能需求”这一问题进行了探讨；他不是一味高喊“尽早测试”，而是实际探讨在哪里、如何运行这些测试。作为个人敏捷实践者的程序员们更不应该错过这套“对象健身操”。Jeff Bay 的“九诫”让我想起大师们对编程之道的追随者的训导。这些训诫虽简洁明了，真要身体力行却绝非易事。但如果照他的指引在一个 1000 行代码的微型项目中做完这套健身操，对于提升面向对象编程和重构能力都将大有裨益。

最后必须强调的是，本书的目标只是引导读者建立实施敏捷所需的基础。绝不应该于此自满停步。就在《ThoughtWorks 文集》中，另外的 8 篇文章即展示了一个敏捷的组织在流程和技术上的不懈进取。在精益求精的道路上，改进永无止境。

郭晓
总经理，ThoughtWorks 中国公司

目录

前言	I
对象健身操	1
1.1 九步迈向优秀软件设计	1
1.2 练习	1
规则	2
1.3 总结	7
项目生命体征	9
2.1 项目生命体征	9
2.2 项目生命体征与健康状况	9
2.3 项目生命体征与信息指示器	9
2.4 项目生命体征：项目范围增量图	10
范围增量的信息指示器示例	10
定义范围的度量单位	10
使用中期里程碑做为对照点来发现瓶颈	11
再释“范围增量图”	11
2.5 项目生命体征：交付质量	11
质量的信息指示器示例	12
再释缺陷数量图	12
2.6 项目生命体征：预算燃尽	13
预算的信息指示器示例	13
再释预算燃尽图	13
2.7 项目生命体征：当前开发状态	14
当前开发状态的信息指示器示例	14
定义开发状态	14
再释故事板与故事卡	14
2.8 项目生命体征：团队感觉	15
团队心情的信息指示器示例	15
再释团队心情图	15
一 键 发 布	16
3.1 持续构建	16
3.2 超越持续构建	16
3.3 全生命周期的持续集成	17
二进制文件的管理	18
3.4 第一道门——提交测试	19
3.5 第二道门——验收测试套件	20
3.6 部署准备阶段	20
3.7 后续的测试阶段	21
3.8 让过程自动化	22
3.9 总结	22
企业Web应用中的敏捷测试 和瀑布测试	24
4.1 简介	24
4.2 测试生命周期	24

4.3	测试分类.....	26
	单元测试.....	26
	功能测试.....	27
	探索性测试.....	27
	集成测试.....	28
	数据验证.....	28
	用户验收测试（UAT）.....	28
	性能测试.....	29
	非功能性测试.....	29
	回归测试.....	29
	产品校验.....	30
4.4	环境.....	30
	开发集成环境.....	30
	系统集成环境.....	31
	试机环境.....	31
	产品环境.....	32
4.5	问题管理.....	32
4.6	工具.....	32
4.7	报表与度量.....	33
4.8	测试角色.....	33
	测试分析人员.....	34
	测试脚本编写员.....	34
	测试执行员.....	34
	环境管理人员.....	34
	问题管理人员.....	35
	故障检测人员.....	35
4.9	参考文献.....	35
	实用主义的性能测试	36
5.1	什么是性能测试.....	36
5.2	需求采集.....	36
	要度量什么.....	36
	如何设定目标.....	37
	如何将性能测试融入日常开发流程.....	37
	开发者需要性能测试告诉他们什么.....	37
	找不到关注性能的客户怎么办.....	38
	如果客户不懂技术又非要坚持不可能的需求该怎么办.....	38
	何不让业务分析师一并采集这些需求.....	39
	小结.....	39
5.3	运行测试.....	39
	运行哪些测试.....	39
	何时运行测试.....	39
	在何处运行测试.....	40
	较小测试设备上的测试结果与生产环境的性能有何关系.....	40
	应该用多大规模的数据库来做性能测试.....	41

如何处理第三方接口	41
需要多少种测试案例	42
为何要多次度量响应时间和吞吐量	42
有必要测试所有功能吗	42
小结	42
5.4 沟通	43
谁需要知道测试结果	43
是否只需要写一份报告	43
小结	43
5.5 流程	43
如何把各种工作连接起来	44
如何确保不拖后腿	44
如何确保每个问题都得到解决	44
5.6 总结	44



1.1 九步迈向优秀软件设计

我们大都读过糟糕的代码。这些代码通常都难以理解、测试和维护。面向对象编程一直在鼓励我们摆脱过程式代码的桎梏，帮助我们以累加的方式编写代码，并重用已有的组件。但有些时候，我们似乎只是把原来陈旧的、充满耦合的设计从 C 程序搬到了 Java 中而已。这篇文章会向编程新手讲述一些非常实用的最佳实践。对于资深的程序员来说，这篇文章可以让你再回顾一下以往的最佳实践，也可以在培训新员工时，作为抽象原则的具体实现示例。

要想掌握优秀的面向对象设计并非易事；但一旦掌握，优秀的设计会在简单性上带来巨大的回报。从过程化的开发到面向对象设计之间的思维转换，要远比看上去的复杂得多。很多开发者都自以为擅长 OO 设计，但在实际应用时，还是不自觉地回到过程化的编程习惯上了，这已经是根深蒂固的观念。更糟糕的是，很多实例和最佳实践（甚至 Sun 在 JDK 中的代码）甚至鼓励人们采用糟糕的 OO 设计——其中一些尚有性能考量作为托辞，而另一些则纯粹是历史包袱。

优秀设计背后的核心概念其实并不高深。比如内聚性、松耦合、零重复、封装、可测试性、可读性以及单一职责。这七条评判代码质量的原则就已经被广泛接受了。然而真正困难的是如何把这些概念付诸实践。理解了封装就是隐藏“数据、实现细节、类型、设计或者构造”，这只是设计出良好封装的代码的第一步而已。因此，本文接下来是一系列实践规则和练习，它可以帮助你良好的面向对象设计原则变得更加具体，从而在现实世界中应用那些原则。

1.2 练习

在一个简单的项目里尝试一些比以前严格得多的编码标准。在这篇文章中，你会看到我给出的“九诫”，它们会迫使你更为严格地以面向对象的风格编写代码。在日常工作中遇到问题时，这些法则可以帮你做出更正确的决定，或者给你更多更好的选择。请你暂时放弃怀疑，在一个代码量为 1000 行左右的小项目里严格地遵守这些法则，到时你会体验到一种完全不同的软件设计的方法。在写完这 1000 行左右的代码后，练习就结束了，你就能放心地将这些规则当作指导原则了。

下面的练习有一定困难，尤其因为所有规则并不是放之四海皆准的法则。比如，很难保证所有的类的长度都不超过 50 行，例外总会出现。但是这种做法背后的价值在于，你应该把与某一职责相关的代码移到真正的、一流的对象中。这个练习的真正价值也在于此，即引发你在开发过程中进行这种思考。所以，现在开始扩展你想象力的极限吧，然后看一看你是不是已经以一种全新的方式思考你的代码了？

规则

下面是练习中要遵守的规则：

1. 方法只使用一级缩进。
2. 拒绝使用 `else` 关键字。
3. 封装所有的原生类型和字符串。
4. 一行代码只有一个 “.” 运算符。
5. 不要使用缩写。
6. 保持实体对象简单清晰。
7. 任何类中的实例变量都不要超过两个。
8. 使用一流的集合。
9. 不使用任何 `Getter/Setter/Property`。

规则1：方法只使用一级缩进

你是否曾经盯着一个体型巨大的老方法而感到无从下手过。庞大的方法通常缺少内聚性。一个常见的原则是将方法的行数控制在 5 行之内，但是如果你的方法已经是一个 500 行的大怪兽了，想要达到这一原则的要求是非常痛苦的。其实，你不妨尝试让每个方法只做一件事——每个方法只包含一个控制结构或者一个代码块。如果你在一个方法中嵌套了多层控制结构，那么你就要处理多个层次上的抽象，这意味着同时做多件事。

如果每个方法都只关注一件事，而它们所在的类也只做一件事，那么你的代码就开始变化了。由于应用程序中的每个单元都变得更小了，代码的可重用性开始指数增长。一个 100 行的，肩负五种不同职责的方法很难被重用。如果一个很短的方法在设置了上下文后，能够管理一个对象的状态，那么它可以应用在很多不同的上下文中。

利用 IDE 提供的“抽取方法”功能，不断地抽取方法中的行为，直到它只有一级缩进为止。请看下面的实例。

```
class Board {
    ...
    String board() {
        StringBuffer buf = new StringBuffer();
        for(int i = 0; i < 10; i++) {
            for(int j = 0; j < 10; j++)
                buf.append(data[i][j]);
            buf.append("\n");
        }
        return buf.toString();
    }
}

Class Board {
    ...
    String board() {
        StringBuffer buf = new StringBuffer();
        collectRows(buf);
        Return buf.toString();
    }

    Void collectRows(StringBuffer buf) {
        For(int I = 0; I < 10; i++)
            collectRow(buf, i);
    }
}
```

```

    void collectRow(StringBuffer buf, int row) {
        for(int i = 0; i < 10; i++)
            buf.append(data[row][i]);
        buf.append("\n");
    }
}

```

注意这项重构还能带来另一种效果：每个单独的方法都变得更简单了，同时其实现也与其名称更加匹配。在这样短小的代码段中查找 bug 通常会更加容易。

第一条规则在此接近尾声了。我还要强调，你越多实践这条规则，就会越多地尝到它带来的甜头。当你第一次尝试解决前面展示的那一类问题时，可能不是非常熟练，也未必能获得很多收获。但是，应用这些规则的技能是一种艺术，它能够将程序员提升到一个新的高度。

规则2：拒绝else关键字

每个程序员都熟知 if/else 结构。几乎每种语言都支持 if/else。简单的条件判断对任何人来说都不难理解。不过大多数程序员也见识过令人眩晕的层层嵌套的条件判断，或连绵数页的 case 语句。更糟糕的是，在现有的判断条件上加一个新的分支通常是非常容易的，而将它重构为一个更好的方式的想法却罕有人去提及。条件判断结构通常还是重复代码的来源。例如，状态标识经常会带来这样的问题。

```

public static void endMe() {
    if (status == DONE) {
        doSomething();
    } else {
        <other code>
    }
}

```

你有很多方式重写这段代码，去掉 else 关键字。例如下面的代码。

```

public static void endMe() {
    if (status == DONE) {
        doSomething();
        return;
    }
    <other code>
}

public static Node head() {
    if (isAdvancing()) { return first; }
    else { return last; }
}

public static Node head() {
    return isAdvancing() ? first : last;
}

```

在上面的例子中，第二段代码由于使用了三元运算符，所以代码长度从四行压缩到了一行。需要小心的是，如果过度使用“提前返回”，代码的清晰度很快会降低。《设计模式》[GHJV95]一书中关于策略模式的部分里有一个实例，演示了如何使用多态避免根据状态进行分支选择的代码。如果这种根据状态进行分支选择的代码大量地重复，就应该考虑使用策略模式了。

面向对象编程语言给我们提供了一种更为强大的工具——多态。它能够处理更为复杂的条件判断。对于简单的条件判断，我们可以使用“卫语句”和“提前返回”替换它。而基于多态的设计则更容易阅读与维护，从而可以更清晰地表达代码的内在意图。但是，程序员要做出这样的转变并不是一帆风顺的。尤其是你的代码中可能早已充斥了 else。所以，作为这个练习的一部分，你是不可以使用 else 的。在某些场景下可以使用 Null Object 模式，它会对你

有所帮助。另外还有很多工具和技术都可以帮助你甩掉 `else`。试一试，看你能提出多少种方案来？

规则3：封装所有的原生类型和字符串

整数自身只代表一个数量，没有任何含义。当方法的参数是整数时，我们就必须在方法名中描述清楚参数的意思。如果此方法使用“`Hour`”作为参数，就能够让程序员更容易地理解它的含义了。像这样的小对象可以提高程序的可维护性，因为你不可能给一个参数为“`Hour`”的方法传一个“`Year`”。如果使用原生变量，编译器不能帮助你编写语义正确的程序。如果使用对象，哪怕是很小的对象，它都能够给编译器和其他程序员提供更多的信息——这个值是什么，为什么使用它。

像 `Hour` 或 `Money` 这样的小对象还提供了放置一类行为的场所，这些行为放在其他的类中都不合适。在你了解了关于 `getter` 和 `setter` 的规则时，这一点会非常明显，有些值只能被这些小对象来访问。

规则4：一行代码只有一个“.”运算符

有时候我们很难判断出一个行为的职责应该由哪个对象来承担。如果你看一看那些包含了多个“.”的代码，就会从中发现很多没有被正确放置的职责。如果代码中每一行都有多个“.”，那么这个行为就发生在错误的位置了。也许你的对象需要同时与另外两个对象打交道。在这种情况下，你的对象只是一个中间人；它知道太多关于其他对象的事情了。这时可以考虑把该行为移到其他对象之中。

如果这些“.”都是彼此联系的，你的对象就已经深深地陷入到另一个对象之中了。这些过量的“.”说明你破坏了封装性。尝试着让对象为你做一些事情，而不要窥视对象内部的细节。封装的主要含义就是，不要让类的边界跨入到它不应该知道的类型中。

迪米特法则（The Law of Demeter，“只和身边的朋友交流”）是一个很好的起点。还可以这样思考它：你可以玩自己的玩具，可以玩你制造的玩具，还有别人送给你的玩具。但是永远不要碰你的玩具。

```
class Board {
    ...

    class Piece {
        ...
        String representation;
    }
    class Location {
        ...
        Piece current;
    }
}
```

```

String boardRepresentation() {
    StringBuffer buf = new StringBuffer();
    for(Location l : squares())
        buf.append(l.current.representation.substring(0, 1));
    return buf.toString();
}

class Board {
    ...

    class Piece {
        ...
        private String representation;
        String character() {
            return representation.substring(0, 1);
        }

        void addTo(StringBuffer buf) {
            buf.append(character());
        }
    }

    class Location {
        ...
        private Piece current;

        void addTo(StringBuffer buf) {
            current.addTo(buf);
        }
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            l.addTo(buf);
        return buf.toString();
    }
}

```

注意在这个例子中，算法的实现细节被过度地扩散开了。程序员很难看一眼就理解它。但是，在为 **Piece** 转化成 **Character** 的行为创建一个具有名称的方法后，这个方法名称和作用就相当一致了，而且被重用的机会也非常高——令人费解的 `representation.substring(0, 1)` 调用可以全部被这个具有名称的方法所代替，程序的可读性又迈进了一大步。在这片新天地里，方法名取代了注释，所以，值得花些时间为方法取一个有意义的名字。理解并写出这种结构的程序并不困难，你只需要使用一些稍微不同的手段而已。

规则5：不要使用缩写

我们总会不自觉地在类名、方法名或者变量名中使用缩写。请抵制住这个诱惑。缩写会令人迷惑，也容易隐藏一些更严重的问题。

想一想你为什么要使用缩写。因为你厌倦了一遍又一遍地敲打相同的单词？如果是这种情况，也许你的方法调用得过于频繁，你是不是应该停下来消除一些重复了？因为方法的名字太长？这可能意味着有些职责没有放在正确的位置或者是有缺失的类。

尽量保持类名和方法名中只包含一到两个单词，避免在名字中重复上下文的信息。比如某个类是 `Order`，那么方法名就不必叫做 `shipOrder()` 了，把它简化为 `ship()`，客户端就会调回 `order.ship()`——这能够简单明了地说明代码的意图。

在这个练习中，所有实体对象的名称都只能包含一到两个单词，不能使用缩写。

规则6：保持实体对象简单清晰

这意味着每个类的长度都不能超过 50 行，每个包所包含的文件不超过 10 个。

代码超过 50 行的类所做的事情通常都不止一件，这会导致它们难以被理解和重用。小于 50 行代码的类还有一个妙处：它可以在一屏幕内显示，不需要滚屏，这样程序员可以很容易、很快速地熟悉这个类。

创建这样小的类会遇到哪些挑战呢？通常会有很多成组的行为，它们逻辑上是应该在一起的。这时就需要使用包机制来平衡。随着类变得越来越小，职责越来越少，加之包的大小也受到限制，你会逐渐注意到，包中的类越来越集中，它们能够协作完成一个相同的目标。包和类一样，也应该是内聚的，有一个明确的意图。保证这些包足够小，就能让它们有一个真正的标识。

规则7：任何类中的实例变量都不要超过两个

大多数的类应该只负责处理单一的状态变量，有些时候也可以拥有两个状态变量。每当为类添加一个实例变量，就会立即降低类的内聚性。一般而言，编程时如果遵守这些规则，你会发现只有两种类，一种类只负责维护一个实例变量的状态；另一种类只负责协调两个独立的变量。不要让这两种职责同时出现在一个类中。

敏锐的读者可能已经注意到了，规则 3 和规则 7 其实是相同问题的不同表述而已。在通常的情况下，对于一个包含很多实例变量的类来说，很难拥有一个内聚的、单一的职责描述。

我们来仔细分析下面的示例。

```
String first;
String middle;
String last;
}
```

这个类可以被拆分为两个类。

```
class Name {
    Surname family;
    GivenNames given;
}

class Surname {
    String family;
}

class GivenNames {
    List<String> names;
}
```

注意思考这里是如何分离概念的，其中姓氏（family name）是一个关注点（很多法律实体约束中需要用到），它可以和其他与其有本质区别的名字分开。GivenName 对象包含了一个名字的列表。在新的模型中，名称允许包含 first, middle 和其他名字。通常，对实例变量解耦以后，会加深理解各个相关的实例变量之间的共性。有时，几个相关的实例变量在一流的集合中会相互关联。

将一个对象从拥有大量属性的状态，解构成为分层次的、相互关联的多个对象，会直接产生一个更实用的对象模型。在想到这条规则之前，我曾经浪费过很多时间去追踪那些大型对象的数据流。虽然我们可以理清一个复杂的对象模型，但是理解各组相关的行为并看到结果是一个非常痛苦的过程。相比而言，不断应用这条规则，可以快速将一个复杂的大对象分解成

为大量简单的小对象。行为也自然而然地随着各个实例变量流入到了适当的地方——否则编译器和封装法则都不会高兴的。当你真正开始做的时候，可以沿着两个方向进行：其一，可以将对象的实例变量按照相关性分离在两个部分中；另外，也可以创建一个新的对象来封装两个已有的实例变量。

规则8：使用一流的集合

应用这条规则的方法非常简单：任何包含集合的类都不能再包含其他的成员变量。每个集合都被封装在自己的类中，这样，与集合相关的行为就有了自己的家。你可能会发现作用于这些集合的过滤器将成为这些新类型中的一部分，或是根据它们自身的情况包装为函数对象。另外，这些新的类型还可以处理其他任务，比如将两个集合中的元素拼装到一起，或者对集合中的元素逐一施加某种规则等。很明显，这条规则是对前面关于实例变量规则的扩展，不过它自身也有非常重要的含义。集合其实是一种应用广泛的原生类型。它具有很多行为，但是对于代码的读者和维护者来说，与集合相关的代码通常都缺少对语义意图的解释。

规则9：不使用任何Getter/Setter/Property

上一条规则的最后一句话几乎可以直接通向这条规则。如果你的对象已经封装了相应的实例变量，但是设计仍然很糟糕的话，那就应该仔细地考察一下其他对封装更直接的破坏了。如果可以从对象之外随便询问实例变量的值，那么行为与数据就不可能被封装到一处。在严格的封装边界背后，真正的动机是迫使程序员在完成编码之后，一定要为这段代码的行为找到一个适合的位置，确保它在对象模型中的唯一性。这样做会有很多好处，比如可以很大程度地减少重复性的错误；另外，在实现新特性的时候，也有一个更合适的位置去引入变化。

这条规则通常被描述为“讲述而不要询问”（“Tell, don't ask”）。

1.3 总结

前面九条规则中，有八条都是非常简单易行的，能够帮助程序员获得面向对象程序设计的圣杯——数据封装。另外，有一条规则（不要使用 `else`，尽量简化条件判断逻辑）鼓励程序员适当地使用多态；还有一条规则是命名策略，鼓励程序员使用一致的、直接的命名标准，不要使用难以理解的缩写。

一言以蔽之，就是想尽办法消除代码中的重复。我们每天都要面对复杂的问题，而我们的目标是，用精炼的代码表达出简单而优美的抽象。

时间长了以后，你会发现在某些情境下，这些规则其实会彼此抵触，如果强加使用可能会产生糟糕的结果。然而，出于练习的目的，你不妨花上 20 个小时，在 100% 地遵守这些规则的前提下，编写 1000 行代码。这样你会发现，自己已经开始打破旧有的习惯了，并且改变了一些曾经墨守的游戏规则。如果遵循本文介绍的这些规则，你必然会遇到这种情形：编程时似乎看到了一个很明显的做法，但是并不应该采纳它，因为这种顺理成章的做法很可能是错误的。

将这些规则当作纪律来遵守，它们会帮助你解答更复杂的问题，同时加深你对面向对象编程的理解程度。如果按照这些规则的要求写上 1000 行代码的话，你会发现最终的结果和编写代码以前所期望的完全不同。尝试一下这些规则，看看最终的结果如何。如果还能够继续遵守它们，那么以后你就能非常自然地写出符合上面规则的代码了，而不必刻意地记住这些规则。

最后要说的是，有些人可能认为这些规则过于刻板，不可能在真实的系统中实施。他们都错了——在本书即将出版的时候，我们刚刚完成了一个超过 100 000 行代码的系统，它严格遵守本文中提到的所有规则。参与这个项目的程序员全部都认真地遵守了上述规则。最终每个人都非常高兴地看到：如果努力地拥抱简单性，那么开发过程会快乐得多。

Stelios Pantazopoulos, 迭代经理

在医疗领域，医生或者护士可以走到病人床前，观察病人的健康状况图表，得到病人的实时生命体征数据。通过这些数据，可以快速地判断病人的健康状况，并决定是否需要进一步的治疗。如果定制软件开发项目能像诊疗者那样拿到一个实时生命体征图表，岂不是非常棒？本文主要讨论如何获得这样一个接近于实时的项目生命体征数据，并及时通知项目团队成员和利益关系人。这样，团队就可以来初步判断项目的健康状况，并采取改正措施来处理导致项目健康状况下降的根源。

2.1 项目生命体征

项目生命体征是一些可收集到的定量度量数据，用于及时反映项目的整体运行状况。这些体征包括。

1. 项目范围增量（Scope burn-up）：对于某期限时所需要交付的项目范围情况。
2. 交付质量（Delivery quality）：最终交付的项目状况。
3. 预算燃尽（Budget burn-down）：根据项目范围交付状况统计的预算使用情况。
4. 实际开发状态（Current state of implementation）：已交付的系统功能情况。
5. 团队的感觉（Team perceptions）：团队对项目状态的看法。

2.2 项目生命体征与健康状况

项目生命体征是一组独立的度量数据，与项目健康不同。项目健康是根据项目生命体征的分析，对项目状态的总体判断。因此，“项目是否健康”是主观的，无法度量的。对于同一个项目生命体征，团队的两个成员对项目健康状况有不同的理解和结论。比如，项目经理可能更注重预算的使用情况，QA（质量保证）人员可能更强调于软件的交付质量，而开发人员可能更关注范围的显著增加。同时，“项目是否健康”与每个团队成员的看法有直接关系，所有人的看法都是相互关联且同样重要的，也是独立的。

对于团队来说，得到整体项目健康状态的最好办法就是收集并展示项目生命体征。如果没有这些体征数据做为参考的话，对于项目健康状况的判断只能说是“猜测”而已。

每个团队都要定义自身项目的项目健康状况。出于一致性考虑，团队成员要列出那些他们认为应该作为项目生命体征的信息。一旦识别出这些项目生命体征后，就要定义信息指示器来显示它们。

2.3 项目生命体征与信息指示器

信息指示器（Alistair Cockburn 提出的一个术语）是一个用于公开信息的显示板，向大家表

明项目的当前状况。它是显示项目生命体征的一个有效方法。对于项目生命体征来说，没有哪种信息指示器是“必须品”。本文建议，对每一被实践证明的确有效的体征都要有一个信息指示器。（当然，这并不代表该方法是显示项目生命体征的唯一方法。）

2.4 项目生命体征：项目范围增量图

项目范围增量图代表项目到最后期限为止所需要交付的范围状态。该度量应该表明项目范围的数字表示、范围完成的比率、以及交付的最后期限。

范围增量的信息指示器示例

图 2-1 中的范围增量图用于度量并显示系统已经完成多少，还有多少没完成。图中的信息包括：

6. 范围的度量单位（用户故事的数目）；
7. 每星期结束时的总范围（3月22日那周是55个用户故事）；
8. 成功交付的两个重要里程碑（里程碑1和里程碑2）；
9. 每个星期完成故事的跟踪曲线（3月22日那周共完成了55个故事中的15个）；
10. 该项目范围应该交付的最后期限（4月26日，第12个迭代结束时）。

基于易交流、可视化以及易维护性等方面的考虑，该图表应该由开发组长或迭代经理来管理，画在白板上，放在整个团队都非常容易看到的位置。

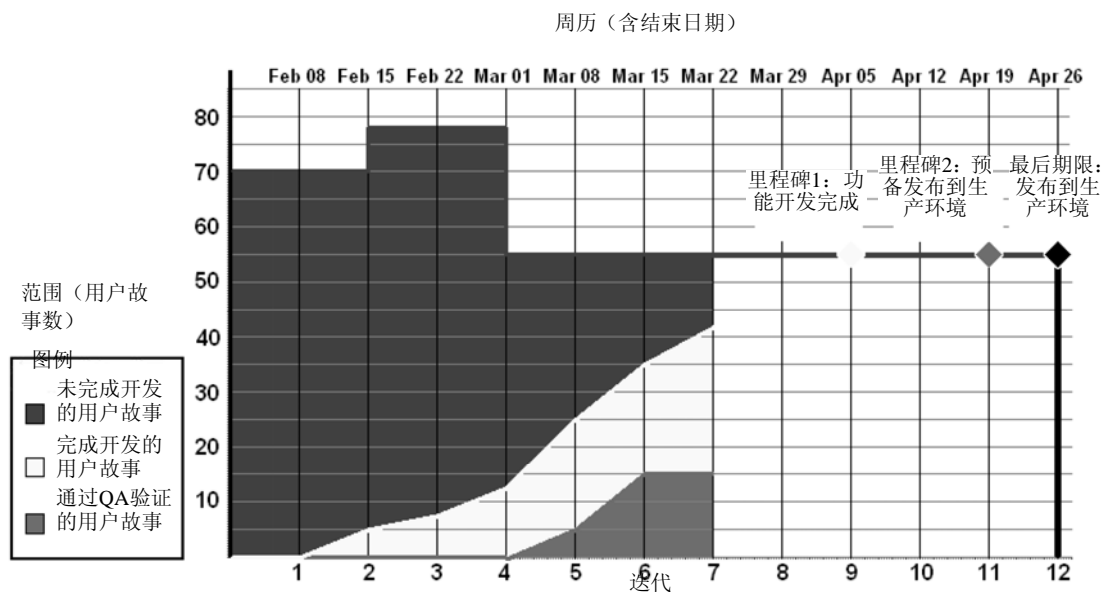


图2-1 项目范围增量图

定义范围的度量单位

为有了有效地得到项目范围增量，在定义范围的度量单位时，要取得所有项目成员的一致同意。而且这个度量单位会因具体项目不同而不同。在理想情况下，这一个度量单位在整个项目过程中不会变化。如果这个定义中途有变的话，历史数据就没有用了。

要避免使用小时数（或天数）作为度量单位。“度量范围”的意义在于了解还有多少内容需要完成，而不是还要多长时间能完成（即多少工作，而非多长时间）。如果使用时间的话，

就成了估计时间与实际时间之间的联系，这会导致难于有效地度量并显示范围。

使用中期里程碑做为对照点来发现瓶颈

基于中期里程碑的进度比率会显示出交付流程运行是否良好。通过这一数值的对比可以发现交付瓶颈。比值的不同表明流程中有瓶颈存在。例如，当“功能完成”里程碑的比值大于“等待上线”里程碑时，表示 QA 环节是一个瓶颈。

再释“范围增量图”

对于示例中的范围增量图来说，该项目范围的度量单位为用户故事。项目开始之前，团队对故事的度量单位做了定义，即一个故事如有下特征：

- 11. 它表示一个或多个用例的全部或部分实现；
- 12. 一个开发人员可以在2~5个工作日内实现它并完成单元测试；
- 13. QA人员可以对其进行验收测试，以确保它满足需求。

在项目开始时，为项目范围设定了两个中期里程碑。里程碑 1 是开发所有用户故事并完成单元测试（但不一定是完成了 QA 工作）的时间点。里程碑 2 是完成所有用户故事、通过单元测试并完成 QA 工作的时间点。这个项目范围增量图直接反映出了完成中期里程碑所经历的过程。这也简明解释了该项目是如何管理范围的。

- (1) 项目开始时，总范围只有 70 个故事。
- (2) 在第二次迭代时，增加了 8 个故事，使总范围变成了 78 个故事。
- (3) 在第四次迭代时，所有项目干系人在一起达成一致意见：根据该图表反映的历史趋势来看，团队不可以在预算内、以预期质量按时完成里程碑 1 和里程碑 2。并一致同意削减项目范围。在后续会议中，所有项目利益关系人同意将 23 个用户故事推迟到下一个版本，因此，当前的项目范围变成了 55 个用户故事。
- (4) 从第五次迭代开始，范围降为 55 个故事。
- (5) 当前处于第八次迭代。范围没有变化，仍旧是 55 个用户故事。团队成员不能确定他们能否完成里程碑 2，但他们决定不急于采取任何调整措施。

以下是用于产生范围增长图的原始数据。

迭代	范围	准备开发 或正在开发	开发完成，等待 QA 验证	开发完成，但有严重 bug	开发完成，通过 QA 验证
1	70	70	0	0	0
2	78	73	2	3	0
3	78	71	1	6	0
4	78	66	3	9	0
5	55	25	9	11	10
6	55	20	8	12	15
7	55	13	10	17	15

2.5 项目生命体征：交付质量

交付质量表示最终交付的产品的状况。该度量应该表明在项目范围内团队交付的质量有多好。

质量的信息指示器示例

缺陷数量图（图 2-2）是用于度量和显示根据严重性分组后的系统缺陷数量。该图包括以下信息：

- 14. 尚未解决的缺陷总数（到3月22日，即第七次迭代结束时，共计47个缺陷）；
- 15. 在发布之前必须修复的缺陷总数（33个高优先级缺陷）；
- 16. 可以推迟到下一个版本修复的缺陷总数（14个低优先级缺陷）；
- 17. 每周的缺陷数（前两次迭代是0，第三次迭代是14个，第四次迭代是8个，第五次迭代是9个，第六次迭代是20个）。

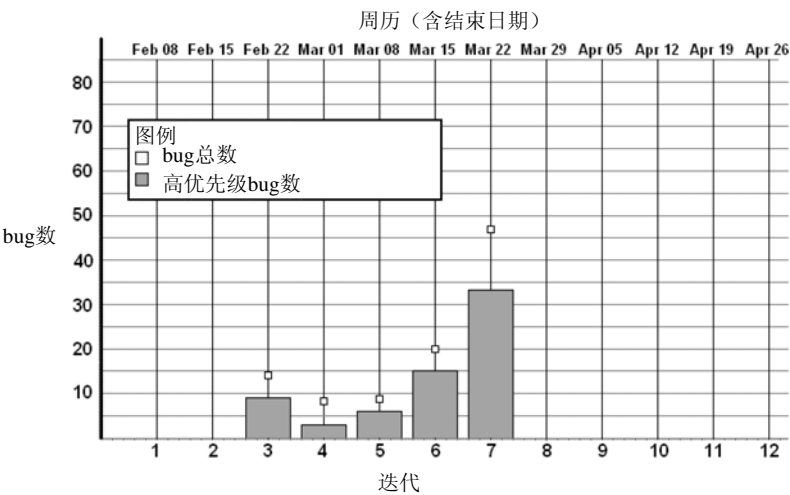


图2-2 bug数量图

为了方便交流、增强可见性以及易维护性，该图应该由测试人员维护，同样要在整个团队内可见。

再释缺陷数量图

当报告一个缺陷时，QA 人员应指定其严重性（低、中、高级严重）。严重的缺陷是指那些阻碍测试，必须立即修复的缺陷。高优先级是指在发布到生产环境前必须修复的缺陷。中优先级是指最好能在发布到生产环境之前修复。低优先级是指能修复更好，但不用必须修复。

每周一的早上，QA 人员会更新这个缺陷数量图。图中高优先级（High-priority）的缺陷是指那些严重性为“严重”和“高”的缺陷，低优先级（Low-priority）的缺陷是那些严重性为“中”或“低”的缺陷。

在本例中，缺陷数量在前两个星期是零，因为 QA 团队还没有组建，没人进行验收测试。以下是生成图表的原始数据。

迭代的最后一	严重 bug	高优先级 bug 数	中优先级 bug 数	低优先级 bug
天	数			数
1	0	0	0	0
2	0	0	0	0

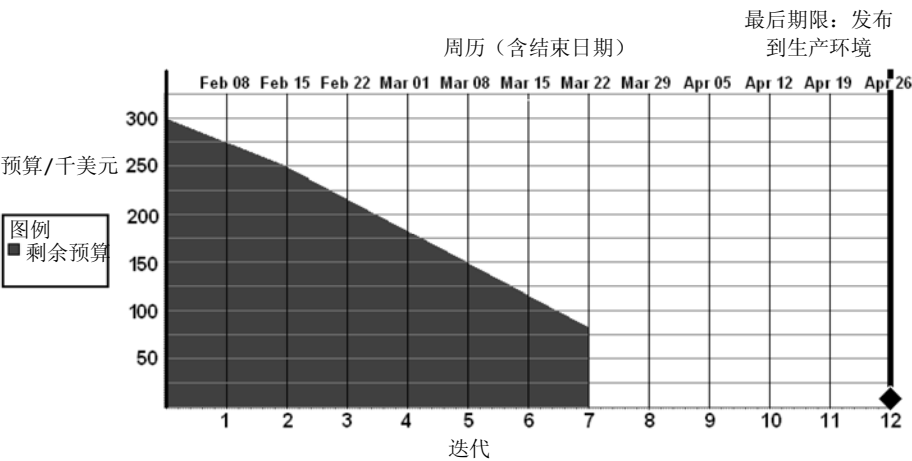
3	0	9	4	1
4	0	3	4	1
5	0	6	2	1
6	0	15	3	2
7	3	30	10	4

2.6 项目生命体征：预算燃尽

预算燃尽代表根据范围交付情况所反应的预算状况。该维度表明了项目有多少预算，预算使用的比率，以及剩余预算还能维持多久。

预算的信息指示器示例

预算燃尽图是一种度量和显示已花费的项目预算，剩余预算，以及预算花费比率的方法（如图 2-3 所示）。该图表示如下信息：



- 18. 预算的度量单位（千美元）；
- 19. 总预算（项目开始时为300 000美元）；
- 20. 当前预算花费额（到第七次迭代为止共计220 000美元）；
- 21. 当前预算剩余（到第七次迭代为止共计80 000美元）；
- 22. 每周的预算花费（前两次迭代为每周25 000美元，从第三次迭代开始，每周涨为33 333美元）；
- 23. 最终交付日期（4月26日，第12次迭代结束以后）。

为了促进交流、增强可见性和易维护性，该图表应由项目经理维护更新，并为整个团队可见。

再释预算燃尽图

前两次迭代中，共有八个项目成员，每个人都租用一台开发用电脑，团队共享同一个构建及源代码服务器。每个月在这些团队成员及租用的电脑和服务器的成本为 25 000 美元。在第三次迭代时，两个新的项目成员、两台租来的开发用电脑以及一个测试服务器被加入到项目中。这些东西使每周的预算成本变成了 33 333 美元。

2.7 项目生命体征：当前开发状态

当前开发状态代表系统交付的实时状态。它表示在项目范围内每一项目交付的实时状态。

当前开发状态的信息指示器示例

用故事板（图 2-4）以及故事卡（图 2-5）来度量和显示当前系统的开发状态。这些图与卡显示出以下信息：

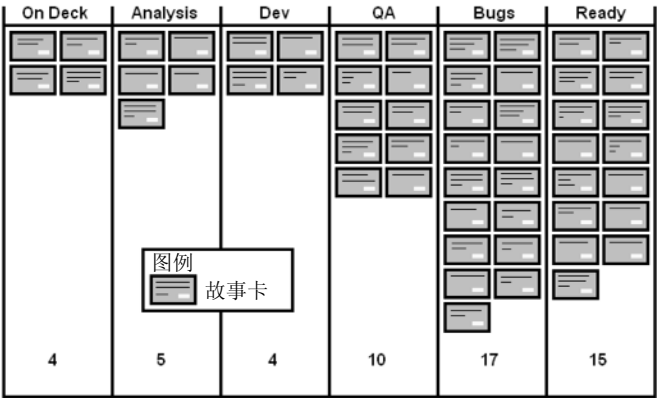


图2-4 故事板

- 24. 每个故事卡都在项目范围之内（一共55个故事卡）；
- 25. 每个故事卡可能的状态（On Deck，Analysis，Dev，QA，Bugs以及Ready）；
- 26. 每个故事卡当前的状态（某一时间，每个故事卡只能有一种状态）；
- 27. 每个状态下的故事卡数量（4个处于on Deck，5个处于Analysis，4个处于Dev，10个处于QA，7个处于Bugs，15个处于Ready）；
- 28. 哪个人员当前正在处理哪个故事卡（浅色标签上是人员姓名，表示John正在处理35号故事）。

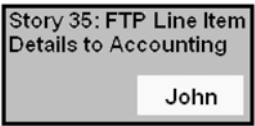


图2-5 故事卡

为了可见性、易沟通及易维护性，这个故事板应由分析人员、开发人员以及测试人员共同管理，并立于整个团队面前。

定义开发状态

对于每个项目来说，开发状态应该是唯一的。在图 2-4 中的状态并不一定适应于所有项目，也不必适应于所有项目。对于状态定义，唯一的标准就是取得团队所有成员一致同意即可。项目中期是可以改变这些开发状态的。而且在项目中期重新审视一下这些状态也是必要的，因为最原始的某些状态可能已经不适合于项目的当前情况了。

再释故事板与故事卡

故事板是第八次迭代的星期二下午 3 点 14 分的状态。下面是对每个状态的解释。

On Deck 故事的有待分析与实现

Analysis	故事正处于分析状态
Dev	开发并写单元测试
QA	故事已完成开发，并且完成单元测试，可以被 QA 人员检查了
Bugs	QA 人员对故事进行了检查，并且发现了缺陷
Ready	QA 人员对故事进行了检查，而且已发现的缺陷都被修复了

2.8 项目生命体征：团队感觉

团队感觉是团队对项目状态的感觉。该度量应该表明团队在项目交付的某个方面上的观点。

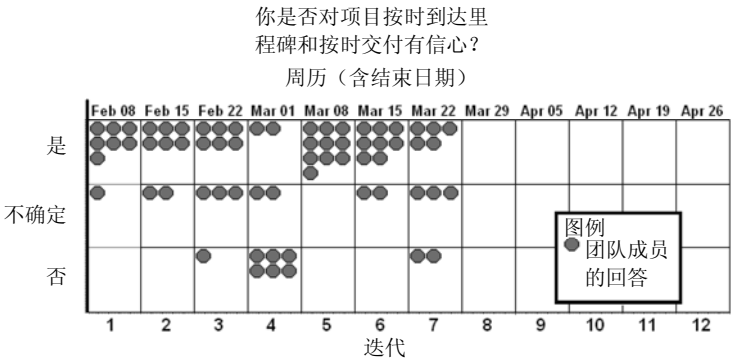


图2-6 团队心情图

团队心情的信息指示器示例

团队心情图（图 2-6）用于度量和显示团队成员对项目状态感觉。该图包括以下信息：

- 29. 每个星期在回顾会议上向团队成员提出的问题（“你对……有信心吗？”）；
- 30. 团队成员可能的回答（“是的，” “不确定，” 或“没有”）；
- 31. 每个团队成员的回答（“当在第六个回顾会议时，十个人中有八个人回答“是的”）；
- 32. 该图表应在每次回顾会议时由所有团队成员来更新，并让整个团队都能看到。

再释团队心情图

用圆点代表每个成员对问题的回答。收集回答时应使用匿名方式，以免互相影响，而且不应只有一个答案。

注：本项目中，项目成员数量发生了变化。前两次迭代，是八个成员，从第三次迭代开始，是十个成员。

3.1 持续构建

敏捷软件开发的核心实践之一就是持续集成（continuous integration，简称 CI）。CI 是指开发人员一旦将代码提交到版本控制系统之后，就进行构建，并运行一系列自动化测试套件的过程。

这一实践已被采纳多年，用于提供软件的高安全性：任何时候，正在开发的软件都要进行构建并通过所有的单元测试。这显著提高了团队对其开发的软件达到其质量要求的信心。对许多项目（甚至大多数项目）来说，这是最终交付的软件在质量和可靠性上向前跨进的一大步。

但对复杂项目来说，问题远不止是能否成功编译和通过单元测试这么简单。

较高的单元测试覆盖率当然好，它比项目需求更接近问题的解决方案，然而从某些方面来说，单元测试可以证明所写的产品代码的确是开发团队心中所想的解决方案，而并不会证明产品代码满足业务需求。

一旦软件被开发出来，它就要被部署。而对于现在大多数软件来说，部署过程不再是复制一个可执行文件就万事大吉了：除了软件本身之外，还需要对很多技术事项（如 Web 服务器、数据库、应用程序服务器和消息中间件等）进行部署及配置。

这样的软件通常都会有一个相当复杂的发布过程，会被部署到各种各样的环境中。比如，它首先可能会被部署在开发环境中运行，然后会被部署到 QA 环境，接着可能被部署到性能测试环境和试运行环境，最后才是生产环境。

绝大多数（“绝大多数”，如果不是“所有”的话）项目的这一过程都会有相当程度的人为干预。人们会手工管理配置文件，对那些需要部署的软件进行剪裁，来适应当时的部署环境。此时，人们常会漏掉一些操作步骤或忘记某些文件的所在位置，比如，你常常会听到这样的话：“我花了两个小时的时间才发现这个测试环境中的模板文件与生产环境中的位置不同”。

上面提到的持续集成是非常有用的实践，但叫它“持续集成”有些不恰当，可能“持续构建”更为恰当一些。那么，它在整个软件发布过程中处于什么位置？而整个系统的持续集成又应该是什么样子的呢？

3.2 超越持续构建

在过去几年实践中，我所在的团队建立了一个端到端的持续集成发布系统，只要轻轻点击一下鼠标，就可以将大型复杂系统部署到任何一个我们想要部署的环境。这种

方法大大减轻了发布时的压力，而且遇到的问题也减少了。在建立这种端到端的持续集成环境的过程中，我们总结出了构建过程的一般步骤，这让我们在项目一开始就可以很快建立这个持续集成系统。

这个流程的核心思想是：每个版本都要通过一系列的检验阶段来证明其质量；每成功通过一个阶段，就证明该版本的软件质量可靠性就增加了一些；通过最后一个检验阶段的版本就是可以被认为是满足发布条件的候选发布版本。对于敏捷软件项目，每次代码提交都会让其产生一个版本，而每个版本都有可能成为满足发布条件的候选软件。

在一个候选发布版本需要走过的整个流程中，有些环节对于大多数项目来说都是必要的，有些环节则可以根据项目的具体情况加以调整。

我们通常将这一流程及其中的所有环节并称为构建管道、管道式构建或者持续集成管道，此外它也被称为“分阶段构建”。

3.3 全生命周期的持续集成

在图 3-1 中是一个典型的全生命周期，它是一个持续集成管道，也是本方法的关键所在。经过各种不同类型的项目实践，最终证明，该方法具有普遍代表性。当然和所有敏捷实践一样，通常在具体项目中还需要剪裁和调整。

该管道式持续集成过程的起点是代码被提交到代码仓库的那一刻。持续集成工具（我们通常会使用CruiseControl）会发现这次提交，并触发以后的各个阶段：编译代码、运行自动化测试，如果测试通过，就会将其打包生成二进制文件¹，并将其保存到统一管理的二进制文件库中。

¹ 这里所说的“二进制文件”包括任何形式的经过编译的代码：.NET的assembly，Java的JAR、WAR和EAR，等等。重点在于：配置信息不包括在内。我们希望同样的二进制文件能在任何环境下运行。

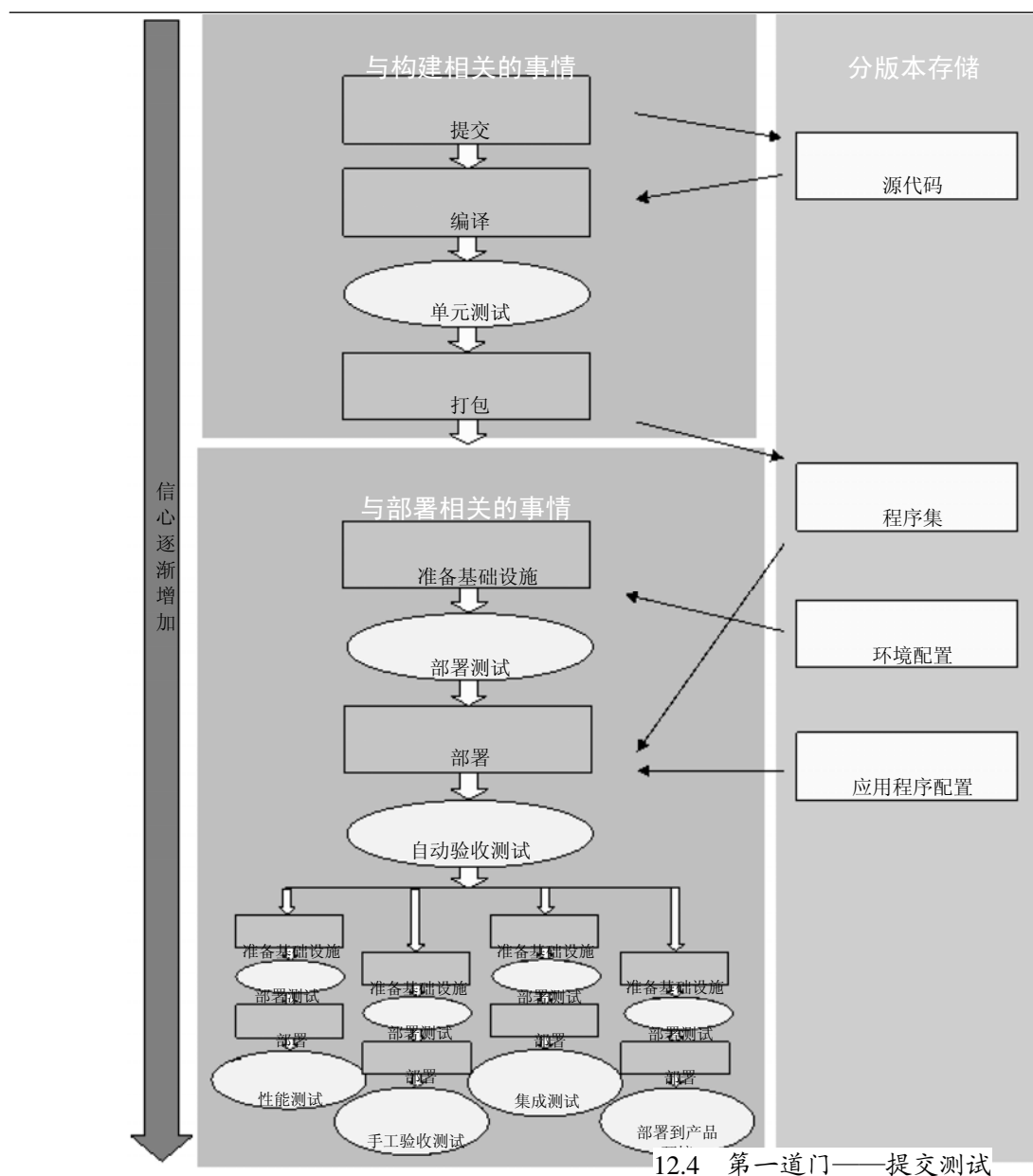


图3-1 持续集成管道

二进制文件的管理

本方法的根本点在于：每通过管道中的一个阶段，该版本就向完全发布靠近一步。使用该方法最重要的理由就是：让错误走到最后发布的机会最小化。

假如我们只保存源代码，那么当以后需要部署某一版本时，我们还要重新编译源代码，假如我们打算用这个版本重新做一次性能测试，这次重新编译就有可能在性能测试环境中得到与上一次不同的结果。可能是因为本次编译使用了与上一次不同的编译器版本，也可能因为使用了不同版本的动态链接库等等。我们希望尽可能消除在部署后才发现那些本该在提交测试和功能测试时就应该发现的错误的可能性。

“避免重复劳动”的指导思想会带来一些额外的好处：流程中每一步的脚本都趋于简单明了，而且鼓励将与环境相关的内容同与环境无关的内容分离。²

² 这一过程也隐含地反对“针对不同部署目标编译不同的二进制文件”的做法：这种与部署目标环境绑定的二进

然而，当以这种方式进行二进制文件管理时，需要注意有一个问题，即这很容易浪费大量的存储空间去保存那些很少用到的二进制文件。一般来说，我们会把它们压缩后保存，而且尽量不将它们保存于版本控制系统中（因为这不值得）。我们使用共享文件系统做为二进制文件仓库，将二进制文件保存在打过版本标记并压缩过的镜像中。迄今为止，我们自己编写脚本来做这件事，因为还没有找到现成的替代方法。

这些镜像文件都标有与之相对应的源代码版本，而这种标识可以表明源代码、二进制文件及相应配置信息、脚本文件等所有相关信息之间的关联关系。

这些受控的二进制文件形成了近期构建的缓冲区，当容量达到一定值后，我们会删除旧文件。一旦有某种原因使我们想将部署回滚到某一版本时，完全可以从该版本的标识中得到相关的源代码信息，并从代码仓库中找回源代码（但这种情况很少发生）。

3.4 第一道门——提交测试

这种自动构建管道由创建一个候选版本开始。而候选版本在任意一个开发人员向版本控制系统提交修改时产生。持续集成工具一旦发现开发人员的提交，立即编译代码并运行一组提交测试。这组提交测试包括所有的单元测试、一些冒烟测试，以及任何能够证明该版本质量满足发布候选条件的测试。

这些提交测试的目标就是快速失败³。开发人员要等到这些测试全部成功以后才能做下一项任务。从这一点来说，速度是维持高效开发过程的关键。而在持续集成管道中，失败出现得越晚，其修复成本就越高。因此，出于维持团队高效性的目的，在这组提交测试套件中，除了单元测试以外，对其他测试应进行适当挑选。

一旦这组提交测试全部通过，虽然管道中后续的阶段还没有开始执行，但我们可以认为本次提交成功，开发人员就可以开始做下一个任务了。

这不只是过程级别上的优化而已。因为在理想情况下，如果所有的验收性测试、性能测试以及集成测试可以在很短的时间里完成，我们就不需要管道式持续集成了。然而现实情况是：很多测试需要更长时间来执行，如果让整个团队停下来等着这些测试完全成功再继续工作的话，团队生产率无疑会显著下降。

将提交测试套件通过与否作为团队是否可以继续下一个任务的条件。当然，并不是说后续的阶段就不再关注了，而是希望团队时刻关注每次提交在管道中的执行情况，其目标是尽可能早地发现错误并将其修复，同时在长时间的测试执行过程中又可以让团队去做其他任务。

只有当这种提交测试的覆盖率足以捕获大多数错误的情况下，这种方法才可能被接受。如果大多数错误是被后续的阶段捕获的，那就表明需要加强一下提交测试。

我们希望开发人员能够尽早提交代码，可是这还取决于提交测试是否能够找到我们引入的绝大多数错误。要达到这一点，我们还需要通过不断试验来优化。

最初，我们可以将运行所有的单元测试作为提交测试，如果某个部分在后续阶段中经常失败，则再为其编写测试并将其加入到提交测试中。

制文件无法灵活地部署。但在很多企业系统中这种情况仍然屡见不鲜。

³ 俗话说得好：早死早投生。——译者注

3.5 第二道门——验收测试套件

单元测试是敏捷开发过程的关键部分，但仅有单元测试并不足以保证软件质量。全部单元测试都能通过的应用程序仍不满足业务需求的情况时有发生。

因此，除包括单元测试在内的提交测试之外，我所在的团队还依赖于自动化验收测试。这些测试根据用户故事的验收条件编写，用于证明我们所写的代码满足用户验收条件。

这些测试就是一种对系统进行端到端验证的功能测试。如果所开发的软件与其他外部系统有交互的话，我们会清除外部连接点来完成这种端到端的功能测试。

我们将验收测试的创建和维护工作也纳入到开发流程中，即只有根据验收条件写好验收测试并将其加入到验收测试套件中、且通过了该测试，才能认为完成了一个用户故事的开发工作。我们还尝试让这些测试更加易读，让非技术人员也可以理解。然而这超出了本文所讨论的范围，就不在这里深究了。

验收测试运行于一个受控环境中，而且可以由持续集成管理系统（通常是 CruiseControl）来监控。

在发布管理当中，验收测试是第二个关键点。我们的自动部署系统只会部署那些通过所有验收测试的软件版本。即任何版本都不可能绕过验收测试被部署到生产环境当中。

3.6 部署准备阶段

在某些项目中，可以实现部署自动化，即将应用程序自动部署到生产环境中；然而，对大型企业级应用程序来说，这基本上是不可能的。可是，如果我们能够将整个基础环境的管理和配置进行自动化，那会消除很多错误的来源，尤其是在企业级系统中手工操作如此之多的情况下更是如此。事情就是这样的，多少值得尝试一下，即便是只取得了部分成功，也能消除许多错误的来源。

我们采取了一些实用的方法来解决这一问题。例如我们通常会依靠标准服务器镜像、应用程序服务器、消息代理服务器以及数据库等。这些镜像表现为部署完整且包括基本配置信息的系统快照。

而这些镜像能以各种各样的形式存在，只要满足项目的需要即可。通常情况下，我们一定会有一个数据库脚本，用它来建立一个初始数据库结构并做一些数据初始化工作，还有一个标准操作系统或应用程序服务器的标准配置，它们可以作为委托过程的一部分被部署或建立在任何我们所选定的服务器上，而这个配置可能仅是一个文件系统的目录结构副本而已，所以总是具有相同的结构。

无论这些镜像是什么，其目的就是建立一个基础环境配置基线，以便以此为基础，对后续的变化进行维护，这样我们就迅速建立一个环境，而且会避免人工干预而产生错误。

但并不是每次部署软件时都重新建立这种初始环境。一般来说只在部署新环境时才使用，平时很少使用。但每次部署软件时，都会将其重置到基本点，以便让后续的部署工作建立在一个基线之上。

一旦这种基本环境准备好后，还要运行一些简单的部署测试。这些测试的目的在于确保当前的基础环境满足部署应用程序的基本要求，一般来说这些测试都非常简单，比如确保 DBMS

是存在的而且 Web 服务器可以响应请求等。

如果此时有测试失败的话，我们就会断定，要么我们的备份镜像有问题，要么我们的硬件环境有问题。

所有测试通过的话，我们就可以开始下一步的部署了。部署时，运行那些用于应用程序部署的脚本，它就会将已通过提交测试和验收测试的指定版本的二进制文件从二进制文件仓库复制到相应的路径去。

除拷贝文件之外，我们的脚本还能启动或停止应用程序服务器或 Web 服务器，运行脚本初始化或更新相应的数据库，可能还需要配置消息代理等。

基本上，部署过程共分为五步，其中四步是每次部署都需要的。

33. 从镜像安装基础结构。（只有初始新的服务器环境时才需要做这一步。）
34. 清理环境，使其重置到基本点，以便让后续的部署工作建立在一个基本点之上。
35. 运行部署测试，以确保基础结构满足软件部署的基本要求。
36. 将应用程序集放到相应位置。
37. 对其进行适当配置，满足应用程序的运行需要。

我们将构建或部署脚本根据其责任分成多个部分，使它们尽可能的简单，而且每个脚本都仅完成一个具体任务，并定义尽可能清晰的输入参数。

3.7 后续的测试阶段

如前所述，验收测试是项目生命周期中的一个关键里程碑。某版本一旦通过了验收测试，便可以部署到各种不同的环境中，假如它没能通过这一步的话，也就不需要浪费人力去做部署。这表明了一个原则，即只有通过彻底的测试被证明可以正常工作的代码才能发布。

到验收测试为止，持续集成管道都是自动执行的。即新的版本通过前一步的测试后，自动进入下一步，并运行该阶段的测试。

在大多数项目中，这种自动化方法在后续的步骤中就不那么适用了。因此，我们让后续的阶段成为可选项。即那些通过验收测试的版本可以选择性地进入人工验收测试阶段或性能测试阶段，亦或是直接部署到生产环境中。

对于这些阶段的部署工作来说，执行上面提到的部署过程五步骤有助于确保一个纯净的部署过程。当某版本被部署到生产环境时，这样的步骤应该已经被执行过几次，因此出现差错的机会很少。

我上一个项目中，每个装有我们的应用程序的机器上都有一个页面，用于显示候选的有效发布版本，还可以选择重新运行功能测试套件和/或性能测试套件。这为我们能够任意时刻无差错部署系统提供了高灵活性。

除了这些后续测试阶段的自动化程度可能有所不同外，整个过程基本是一致的。对于某些项目而言，可能有必要每次都运行性能测试，而对另外一些来说，可能就没有必要。其实，验收测试的后续阶段之间的关系以及自动化程度的高低并不是问题，而如何提供并管理这些持续集成过程的脚本才是需要考虑的事情。

3.8 让过程自动化

图 3-2 反映了自动化持续集成管道脚本。每个方框代表一个阶段，方框内的每一行代表了一个脚本来完成相应的功能。

大多数项目中，前两个阶段（提交测试及验收测试）通常可由诸如 **CruiseControl** 之类的持续集成工具来管理。

使用这种方法组织构建脚本的最大好处就是：每个脚本或元素都只负责做好一件事，而不是用相对复杂的步骤来管理整个构建过程。随着项目的演进与成熟，这一点对于确保构建过程的可管理性及易检验性是非常重要的。

至于如何写这些脚本并不在本文的讨论范围之内。实际上，这些脚本严重依赖于具体项目，不太可能引起读者的广泛兴趣。然而，当将这一过程应用于多个项目之后，我们发现它提供了可靠的、可重复的而且值得信任的部署过程，让我们可以在几秒或几分钟内完成此前可能需要数天才能完成的工作。

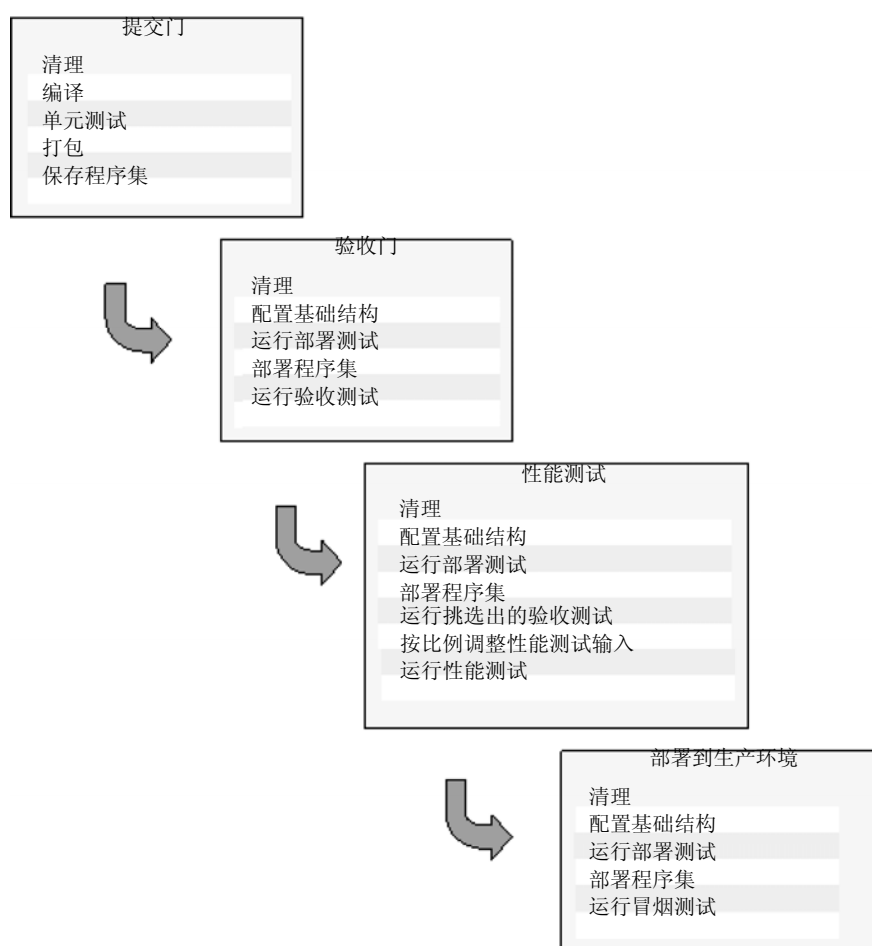


图3-2 过程示例


3.9 总结

如果你还没有使用持续集成方法进行构建，请从明天就开始吧。我们的实践证明，这是提高系统可靠性最有效的途径。持续集成在消除人为错误方面效果显著，不但可以提高生产效率，

而且提高交付质量，降低交付压力。⁴

⁴ 因成文较早，作者在项目中使用CruiseControl作为持续集成服务器，需要自编脚本去维护持续集成管道的配置，以及二进制文件仓库的管理工作。现在，Cruise (<http://cruise.thoughtworks.com>) 已经可以轻松完成这些事情。

企业Web应用中的敏捷测试和瀑布测试



Kristan Vingrys, QA 咨询师

4.1 简介

同是企业 Web 应用程序项目，一个用敏捷，一个用瀑布流程，它们的测试策略会有何不同？在二者中，测试的关注点都在于告诉业务客户这个应用程序做了哪些事情，同样也要消除应用程序作为产品交付以后的失败风险。它们的主要区别不是测试本身，而是何时执行测试、由谁执行测试。测试的每个阶段都可以在系统就绪后随时开始，无须等待前一个测试阶段完成。

从未涉足敏捷项目，或是刚启动某个敏捷项目并在寻找指导建议的读者都可以看看这篇文章，它正是为你们而写。文中的信息虽并非笔者新创，但亦是收集整理的结果，希望这些信息能帮助你们朝着更为敏捷的过程迈进。

敏捷项目的测试阶段跟瀑布项目的测试阶段几乎毫无二致。每个阶段都有准出标准，但是在进入某个测试阶段之前，是不需要等待整个应用程序完成的。只要应用程序所完成的部分足以进入下一个测试阶段就行。因为测试的对象是一个已完成的功能，而不是一个发布，所以测试阶段是在持续并行进行的。于是就有了一大堆回归测试。这便也意味着回归测试是测试自动化的基础。对于敏捷项目而言，环境与资源也是要注意的地方，因为对测试环境的需求会来得更早、更加频繁。

“快速失败”是敏捷项目的一句格言，它的含义是尽可能早地判断出应用程序无法满足业务需求。要做到这一点，就需要不断地对解决方案是否满足业务需求进行检测，一旦不满足，就要尽早把问题解决。敏捷团队成员——开发人员、测试人员、架构师、业务分析师以及业务代表等人都关注于尽早交付业务价值。所以，测试需要所有团队成员协力来做，它不仅仅是测试人员的责任。

4.2 测试生命周期

从测试生命周期就可以看出瀑布和敏捷项目之间最大的差异。瀑布项目对每个阶段都有严格的准入和准出标准，而且只有前一个阶段结束，才可以进入下一个阶段。而敏捷项目则会尽可能早的启动测试阶段，并且允许不同的阶段出现重叠。敏捷项目也有一些结构性的内容，如准出标准，但它没有严格的准入标准。

在图 4-1 中，你一眼就能看出敏捷项目和瀑布项目的测试生命周期差异所在。在敏捷项目中，业务分析师、测试分析师和业务代表等人一起讨论某个想法的行为，它如何适配

于整体目标，怎样去验证它是否完成了该做的事情。这些分析构成了功能测试、用户验收测试和性能测试的基础。做完分析之后便开始功能的开发，单元测试、集成测试、探索性测试、非功能性测试（及数据验证——如果有这一项的话）也纷纷开始。不过只有等系统可以作为产品运行时才开始进行产品验证。

测试阶段没有严格的准入标准，这就意味着只要时机合适，随时都能开始。因为每个测试阶段对于确保应用程序的质量都至关重要，所以便应该尽早开始每个阶段的分析工作，这可以帮助人们修正设计，找出问题，为以后节省出大量的时间。下面是敏捷项目的一些准出标准。

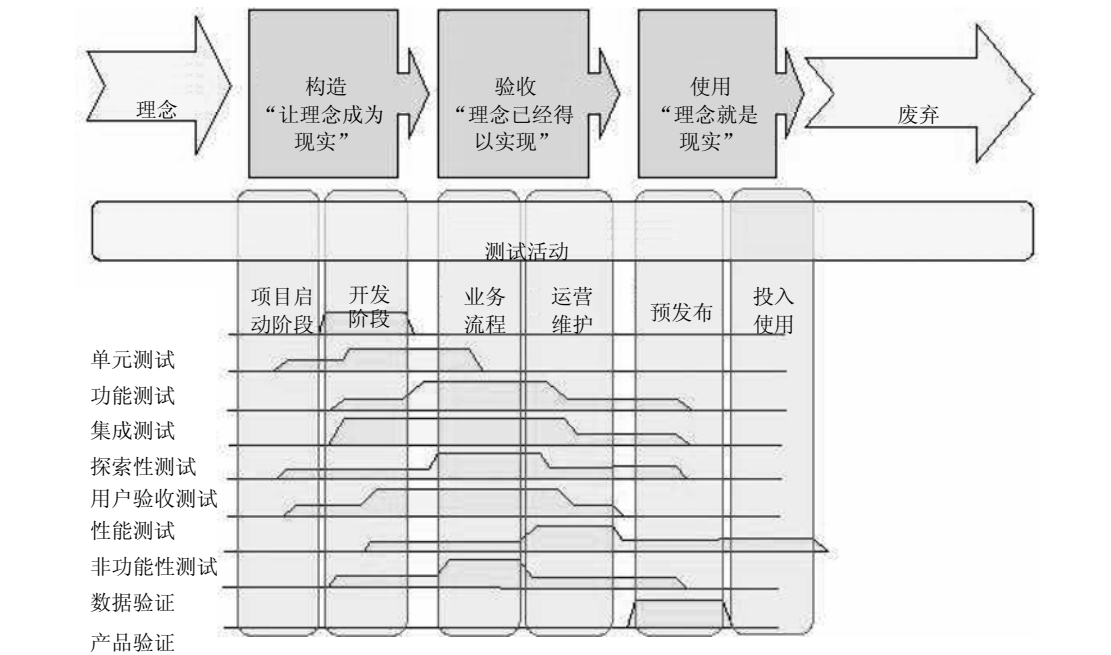


图4-1 敏捷项目和瀑布项目的测试生命周期

单元测试的标准：

- 38. 100%自动化；
- 39. 100%通过；
- 40. 超过90%的代码覆盖率；
- 41. 纳入持续构建。

集成测试的标准：

- 42. 100%自动化；
- 43. 100%通过；
- 44. 纳入持续构建。

功能测试的标准：

- 45. 90%以上自动化；
- 46. 100%通过；
- 47. 所有的自动化测试都纳入持续构建。

探索性测试的标准：

- 48. 测试分析师对应用程序的质量有信心。

用户验收测试的标准：

- 49. 业务代表认可应用程序满足需求；
- 50. 用户认可应用程序的可用性。

性能测试的标准：

- 51. 100%自动化；
- 52. 业务人员认可应用程序满足了业务性能需求；
- 53. 性能测试可以重复执行。

非功能测试的标准：

- 54. 业务人员认可非功能需求得到了满足；
- 55. 操作人员认可非功能需求得到了满足。

数据验证测试的标准：

- 56. 确信数据被正确移植。

产品验证的标准：

- 57. 应用程序在产品环境中正确安装。

瀑布项目的测试生命周期限制某个测试阶段只能在前面的测试阶段走完以后才能开始。从理论上讲这也是说得过去的，因为后面的测试阶段会依赖于前面的执行通过（如果某些功能不正确，就用不着再对它做性能测试了）。但是，要是非得等到所有功能都正确实现以后才开始性能测试，可就一点道理都没有了。敏捷项目会在适当的时候启动每一个测试阶段，这样可以尽早找出问题，让团队有更多的时间解决问题。但是敏捷项目的准出标准跟瀑布项目是一样的。除非功能都验证无误，否则就不能认为性能测试已经完成。

4.3 测试分类

敏捷项目跟瀑布项目的测试分类几乎是一样的。其差别主要在于大部分精力投入的地方和每个测试阶段所执行的时机。敏捷项目倾力关注单元测试和功能测试，从而为稍后执行的测试阶段创造出高质量的代码，于是后期就不会发现本应在早期发现的缺陷，并能专注于所需要测试的领域。而瀑布项目就有一个常见的问题，后期测试的焦点总是放在找出本来应该在前期被发现的缺陷身上。于是修复缺陷的成本提高了，测试工作的投入翻番了，测试的关注点也偏离了。

瀑布项目和敏捷项目另外一个巨大的不同在于测试自动化。敏捷项目力求在所有测试领域内都达到 100% 自动化。测试跟持续构建系统集成在一起，于是当代码发生变化时，这个变化会被自动检测到，应用程序被构建起来，然后所有测试都会被执行。

测试驱动开发（TDD）在敏捷项目中很常用，用这种方法的时候，测试用例比代码要先一步创建。于是我们就可以越来越多地看到为代码和功能创建的测试用例。用自动化测试来驱动开发，然后消除重复，这种做法可以保证无论复杂度多高，任何开发者都可以写出可靠的、没有 bug 的代码。TDD 常用的地方是单元测试，但也同样可以用于功能测试、集成测试、用户验收测试和性能测试。

单元测试

单元测试又称白盒测试，它要测试所开发出来的每一个模块。瀑布项目不关注这个测试阶段，而且大多数时候即便有的话也是随意为之。敏捷项目强调单元测试，而且会把所有单元测试都自动化。自动化的单元测试是敏捷项目的基础，对持续集成和重构起辅助作用。

单元测试应当考虑以下几点：

58. 用stub和mock来消除对外部接口的依赖；
59. 由创建代码的开发人员编写单元测试；
60. 单元测试要能自动化执行，而且要被包含在持续开发构建中；
61. 单元测试之间不能有依赖，让每一个单元测试都能独立执行；
62. 任何开发人员都要能在他自己的机器上执行单元测试；
63. 用代码覆盖率来判断哪些部分的代码没有被单元测试覆盖；
64. 在检入代码的修改之前，要保证单元测试100%通过。

任何测试失败都表示构建失败。

功能测试

功能测试常常跟系统测试相关联，它的重点是测试应用程序的功能（包括负面测试和边界条件）。在瀑布项目中，测试团队一般是在这个阶段开始测试工作。测试团队的成员会一直等下去，直到开发者完成了所有的功能，并通过所有单元测试之后才进入功能测试阶段。敏捷项目把功能拆分成故事，每次迭代开发一定数量的故事。每个故事都有一些验收标准，这些标准一般都是由业务分析师和测试分析师制定的，它们也可以看作跟测试条件类似。测试分析师会根据验收标准创建测试用例，用它们来检测代码行为的完成程度。故事一旦编码完成，而且单元测试运行通过以后，就可以运行功能测试来判断是否满足验收标准了。这就意味着在敏捷项目中，只要第一块功能已经完成编码，功能测试就可以启动，而且会贯穿整个项目的生命周期。

功能测试应当考虑以下几点。

65. 要能自动化执行，并且进入持续构建（如果测试运行时间很长，也可以只在开发持续构建中包含一小部分精挑细选的功能测试，而在系统集成持续构建中包含全部功能测试）。
66. 在编码之前写下测试意图，代码完成后对测试进行实现。
67. 把通过所有功能测试作为故事完成的条件。
68. 在应用程序安装到另外一个环境（如试机环境或产品环境）上的时候需要执行功能测试。

任何测试失败都表示构建失败。

探索性测试

探索性测试又称随机测试。瀑布项目在它们的测试策略中没有这种测试类型，不过大多数测试人员都会多少做一些探索性测试。在敏捷项目中这是个很关键的测试阶段，因为它可以用来检测自动化测试的覆盖率，并收集对于应用程序质量的反馈。它为测试人员和业务分析师提供了一种结构化的方式去使用、去探索系统，从而找到缺陷。如果探索性测试在某个功能区域内找到了大量缺陷，那这部分的自动化测试用例就要被审核。

探索性测试应当考虑以下几点：

69. 在系统集成环境中执行；
70. 在较高的层次（如在wiki中）上监测探索性测试活动；
71. 用自动化的环境设置方式来减少搭建环境的时间；
72. 将破坏性测试作为探索性测试的一部分。

集成测试

应用程序在作为产品部署的时候，需要各个部分的协作；集成测试就是把这各个独立的部分集成起来进行测试。瀑布项目不但会把应用程序的各个独立模块进行集成，还会把那些虽然不属于项目的一部分，但是要开发当前应用程序就必须用到的一些应用程序也做集成。在敏捷项目中，独立模块间的集成是被持续构建所覆盖的，所以集成测试的关注点就是那些不属于当前项目的外部接口。

集成测试应当考虑以下几点。

73. 在执行集成测试的时候，需要考虑到当前迭代还没有开发的功能；
74. 写一些集成测试来定位特殊的集成点，以协助代码调试，即便功能测试会调用到这些集成点也无妨；
75. 将集成测试自动化，放到系统集成持续构建中。

任何测试失败都表示构建失败。

数据验证

如果项目需要移植既有数据的话，就要进行数据验证。它可以保证现有的数据被正确地移植到新的 Schema 中，新的数据被添加进来，旧的数据被移除。这种类型的测试在瀑布项目和敏捷项目中是被同等对待的，除了敏捷项目会尽力把它自动化以外。

数据验证应该考虑以下几点：

76. 在系统集成环境、试机环境和产品环境中都要执行；
77. 尽可能地自动化；
78. 要把测试放到系统集成持续构建中。

任何测试失败都表示构建失败。

用户验收测试（UAT）

UAT 关注的是完整的业务过程，它用来确保应用程序能按照业务的处理方式工作并能满足业务需求。它同样还要从客户、消费者、管理员等各种用户的角度出发考虑应用程序的可用性。在瀑布项目中，这个阶段通常就被用来找出应该在早期阶段就被发现的 bug。业务人员也会在这个阶段验证开发团队交付的应用程序的质量。敏捷项目用 UAT 来确保应用程序满足业务需求，因为等到进入这个测试阶段的时候，代码质量已经较高了。在敏捷项目中，业务人员从早期的测试阶段就开始参与，所以他们对交付的东西有更多的信心。

UAT 应该考虑以下几点：

79. 首先进行手工测试，等它验证了系统行为以后再把它自动化；
80. 把自动化测试放到系统集成持续构建中；
81. 让应用程序的最终用户亲自将整个程序运行一遍，不过项目的测试人员要在旁边协助；

82. 在试机环境下执行UAT，用作验收；
83. 只要完成了一个业务过程或者一个主要的UI组件，就要执行UAT。

任何测试失败都表示构建失败。

性能测试

性能测试涵盖的范围比较大，不过一般可以分成以下三类。

84. 容量测试：独立测试核心功能组件的容量。例如，可以支持多少用户并发搜索？一秒钟能做多少次搜索？等等。容量测试被用来精确度量系统的极限，还可以对容量规划和系统的扩展性起辅助作用。
85. 负载测试：侧重于系统在负载下的表现。负载应该要体现出用户所期望系统可以经受得住的流量。
86. 压力测试：关注系统在压力下的表现。比较常用的技术是浸泡测试（soak test）；在浸泡测试中，系统会在一定的负载下持续运行一段时间，用来找出长期问题，如内存泄漏、资源泄漏等。压力测试还会覆盖到故障转移和恢复，例如让正在工作的集群中的一台服务器出现问题，检查是否可以做到故障转移和恢复。

瀑布项目直到项目接近尾声的时候才做性能测试，这个时候应用程序已经“完成了”开发，通过了单元测试和功能测试。而敏捷项目则会尽快启动性能测试。

性能测试应该考虑以下几点。

87. 在功能测试中设置一些性能度量，例如一个测试第一次运行要花多长时间，接下来每次又要花多久，把这些时间所占的百分比做一下比较（上升表示有问题，下降表示良好）。
88. 把一些性能测试放到系统集成持续构建中去做。
89. 一旦一个业务过程，或是某个规模比较大的功能或接口完工，就要做性能测试。
90. 只有在试机环境中运行的时候才签收性能测试。

任何测试失败都表示构建失败。

非功能性测试

非功能性测试所涵盖的范围很广，性能测试通常也属于这个话题。但是因为性能测试是企业解决方案中很重要的一部分，而且需要不同的资源和技能集，所以就被划出来单独成为一个测试阶段。非功能性测试一般都包含有这些内容：可操作性（包括监控、日志、审计/历史记录）、可靠性（包括故障转移，单个组件故障，完整故障，接口故障）以及安全性。无论瀑布项目还是敏捷项目，在这个测试阶段都会遇到重重阻碍，二者的差异不太突出。

非功能性测试应该考虑以下几点：

91. 非功能性需求一般都是很难捕获的，而且即便被捕获，也很难进行度量（例如，99.9%的无故障时间）；
92. 尽可能让所有的非功能测试都自动化执行，把它们也都放到系统集成测试环境中；
93. 在定义测试用例的时候，让真正要监控产品环境并对其提供支持的人也参与进来；
94. 在应用程序成为产品以后，非功能测试或监控还要继续。

回归测试

在瀑布项目中，回归测试无论从时间上还是费用上来看，都算得上是成本最高的测试阶段了。如果在项目晚期（如 UAT 阶段）发现了缺陷，再构建应用程序的时候，就得把所有单元测试、功能测试和用户验收测试重新运行一遍。因为大多数瀑布项目都没有自动化测试，所以回归测试的开销就很大。敏捷项目以持续构建和自动化测试来应对回归测试，使每次构建都可以进行回归测试。

回归测试应该考虑这一点：

95. 每次迭代结束时都做一下手工测试（如果规模很大的话，就进行拆分，做到每三四次迭代就能执行完一次），收集早期反馈。

产品校验

产品校验会在把产品交付给用户使用之前，审查产品环境中的应用程序，看看所有内容是否都已经正确安装，系统的操作性如何。而有些测试还是只能到了产品环境中才能完整运行的，最好尽快将其完成。无论是瀑布项目还是敏捷项目，产品校验的方式并无二致。

产品校验应该考虑以下几点：

96. 让最终用户来做产品校验测试；
97. 趁着产品系统还没有正式上线，从这个测试阶段的早期就要尽可能多地执行自动化回归测试。

瀑布项目和敏捷项目的测试阶段还是很相似的，主要差异就是每个测试阶段所关注的重点和执行时机。敏捷项目中有大量的自动化测试，用持续集成来减少回归测试对项目带来的影响。在瀑布项目中，如果发现应用程序的质量低下，那么在晚期再去执行前期的测试就是很常见的事情（如在 UAT 的时候作功能测试）。敏捷项目可以减少测试中的浪费，在早期发现问题，让团队在交付应用时增强信心。

4.4 环境

在开发过程的各个阶段都要用到测试环境，从而确保应用程序的正常运行。越到后期，测试环境与预期的产品环境就会越相似。测试环境一般都会包括一个开发环境，让开发者集成代码并运行一些测试。系统集成环境跟开发环境有些类似，不过它会集成更多的第三方应用程序，也许还有大批量的数据。试机环境几乎是产品环境的镜像，也是应用程序变成产品之前的最后一个阶段。

敏捷项目和瀑布项目所需的环境没太大区别，其中一个不同之处在于，敏捷项目从项目伊始直至项目结束，都要用到所有的环境。在敏捷项目中，保证所有的环境都一直正常工作也是很重要的。无论因为什么原因让某个环境出现故障，都要立刻让它重新工作起来。在这个话题上，敏捷和瀑布还有另外一点差异，那就是环境的计划和资源分配对它们的影响不同，尤其是当各种环境都被项目之外的团队进行管理的时候，其差异尤为显著。

开发集成环境

开发者在开发环境中集成代码，开发应用程序。瀑布项目对开发环境的重要性不会考虑太多：开发环境中的代码一直都不能工作，到了开发者需要彼此集成代码的时候才想起来要用，而这时项目已经接近尾声。在敏捷项目中，开发环境是整个开发工作中不可分割的一部分，在开始编码之前就必须准备就绪。这个环境会被用来持续地集成代码和运行测试套件。无论因

为什么原因造成环境故障，都要立刻修复。

开发环境应该考虑以下几点。

98. 集成代码、构建和测试的时间加起来不要超过15分钟。
99. 每个开发人员所用的环境跟开发环境要保持一致（硬件环境可以不一样，但是软件环境一定要一样）。
100. 所用的数据要尽可能跟产品数据保持一致。如果产品数据过于庞大，难以载入，也可以只截取一部分。在每个发布周期的开始阶段，这些数据要从产品数据中重新更新。
101. 管理这个环境的责任应该落在项目团队身上。
102. 向这个环境中部署的频率大约是以小时计算。
103. 自动部署。

系统集成环境

系统集成环境用来将所开发的应用程序和其他应用程序进行集成。在瀑布项目中，这个环境（如果有的话）只会在项目接近尾声的时候才会用到，而且倾向于多个项目共用一个集成环境。在敏捷项目中，一旦开始编码，这个环境就要准备就绪。应用程序会被频繁部署到这个环境中，继而开始执行功能测试、集成测试、可用性测试和探索性测试等等。应用程序的演示就是在这个环境中进行。

系统集成环境应该考虑以下几点。

104. 集成点应该被真正的外部应用程序所代替。外部应用程序应该处于测试状态，而非真正的生产版本。
105. 把产品环境的架构复制过来。
106. 在这个环境中所使用的数据应该是产品环境数据的副本，每个发布周期的开始阶段，都要从产品数据中重新更新。
107. 建立运行这个环境中所有测试的系统集成持续构建。
108. 管理这个环境的责任应该落在项目团队身上。
109. 向这个环境中部署构建的频率大约是以天计算。
110. 自动部署应用程序。

试机环境

试机环境用来验证应用程序可以部署为产品，而且工作正常。为了达到这个目的，试机环境应该完全复制产品环境，包括网络配置、路由器、交换机以及计算机性能等等。在瀑布项目中，这个环境往往需要“预订”，也要有一个计划，计划在这个环境中进行多少次部署以及何时进行部署。敏捷项目对这个环境不像对开发环境和集成环境那样依赖；不过在项目的整个生命周期中，还是需要常常进行部署。

试机环境应该考虑以下几点。

111. 这里的数据应该是产品数据的完整副本，每次应用程序部署前都要更新。
112. 用它来验收UAT，性能测试和非功能测试（稳定性、可靠性等等）。
113. 向这个环境中部署构建的频率大约是以迭代计算，如每两周一次。
114. 管理这个环境的人应该就是管理产品环境的人，让他提前接触应用程序并进行知识传递。
115. 自动部署应用程序。

产品环境

产品环境是应用程序正式上线时的环境。产品校验测试就在这个环境中执行，同时会做一些度量以检验测试成果。瀑布项目和敏捷项目在这里的区别不大。在敏捷项目中，发布过程会尽力做到自动化，为向产品环境中频繁发布提供条件。

产品环境应该考虑以下几点。

- 116. 在应用程序正式上线之前（或者刚刚上线之后），在产品环境中做回归测试。
- 117. 要做度量，以检验测试成果，例如在前三个月到六个月之前，用户报告了多少问题，问题的严重性如何。

无论是哪种项目，要保证时间期限，就都得做到在需要用到环境的时候就有一个环境可供使用。瀑布项目会设法遵守严格的计划，便于对环境做安排。敏捷项目的灵活性更强。也许有了足够的功能就可以进入试机环境了，或者业务人员会决定产品提前上线。为了做到向试机环境快速部署，然后进入产品环境，就要有系统集成环境以及相关过程的辅助。

4.5 问题管理

问题包括缺陷（bug）和变更请求。瀑布项目有着严格的缺陷和变更请求管理，而敏捷项目饱含变化，所以就没有那么严格的变更管理。如果有变更，就创建一个（或是一些）故事，放到待处理事项里面。高优先级的故事会放到下一次迭代中完成。

缺陷管理在敏捷项目中依然适用。如果有人发现一个正在开发故事有缺陷，大家就会进行非正式的沟通，发现缺陷的人把它告诉开发人员，缺陷会立刻被修复。如果某个缺陷并不属于当前迭代中开发的故事，或者属于当前故事，但并不严重，那就用缺陷跟踪工具记录下来。这个缺陷会被当做故事处理；也就是说，会创建一张故事卡，让客户排优先级，放到待处理故事里面。团队需要进行权衡：要找到问题所在，为大家理解这个问题并安排优先级提供足够的信息，但又不能在一个对客户而言优先级并不是很高的缺陷上面花太多时间。

瀑布项目和敏捷项目的缺陷内容（描述、组件、严重程度等）是一样的，除了一个字段以外：敏捷项目多了一个字段——业务价值，如果可能的话就用币值描述。这个字段表示如果缺陷被解决的话可以带来多少业务价值。将业务价值跟缺陷关联以后，客户就更好地判断这个缺陷跟新功能相比是不是更有价值，是不是应该有更高的优先级。

4.6 工具

所有项目都要用到工具，只是程度不同。瀑布项目用工具来强化过程以及提高效率，这有时会造成冲突。敏捷项目用工具辅助提升效率，与过程无关。在敏捷项目中，所有的测试都应该可以在任何团队成员的个人环境中运行，也就是说，所有人都可以使用那些自动化测试用例的工具。所以敏捷项目中会经常用到开源产品，这又意味着使用这些工具需要不同的技能。开源工具不像商业工具那样有齐备的文档和完善的支持，用这些工具的人要有很强的编码能力。如果有人编程能力偏弱，就可以通过跟人结对来提升个人技能。在敏捷项目中也可以使用商业工具，但是大多数商业工具在开发的时候都没有考虑敏捷过程，所以跟敏捷项目匹配起来就不太容易。而且要让商业工具跟持续集成配合，可能要写很多代码才行。

项目中应该考虑为下面一些测试任务使用工具：

- 118. 持续集成（如CruiseControl, Tinderbox）；

-
- 119. 单元测试（如JUnit, NUnit）;
 - 120. 代码覆盖率（如Clover, PureCoverage）;
 - 121. 功能测试（如 HttpUnit, Selenium, Quick Test Professional）;
 - 122. 用户验收测试（如Fitness, Quick Test Professional）;
 - 123. 性能测试（如JMeter, LoadRunner）;
 - 124. 问题跟踪（如BugZilla, JIRA）;
 - 125. 测试管理（如Quality Center）。

4.7 报表与度量

度量数据是用来衡量软件质量和测试成果的。在瀑布项目中，有些测试度量指标需要在测试之前就把所有测试用例都写好，而且仅在应用程序开发完毕时进行一次测试。这种指标包括：每个测试用例执行的时候发现多少缺陷，每天执行的测试用例会发现多少缺陷。这些度量数据收集起来以后，使用来判断应用程序是否已经就绪并可以发布。在敏捷项目中，功能完成的时候测试用例就已经写好且运行完毕，这就意味着用来度量瀑布项目的一些指标是无法应用在这上面的。

回到收集度量数据的原因上来——衡量软件质量和测试成果，你可以看看下面这些概念。

- 126. 用代码覆盖程度量测试效果；这对于单元测试尤其有效。
- 127. 在探索性测试阶段发现的缺陷数量可以说明单元测试和功能测试的效果。
- 128. 在UAT阶段发现的缺陷表示先期的测试并不像UAT一样充分，我们应该关注业务过程，而不是软件的bug。如果UAT发现了很多功能性问题，而不是软件的bug，这就表示团队对故事或是变化的需求理解不足。
- 129. 故事完成以后所发现的缺陷数量能够作为衡量软件质量的好手段。这些缺陷包括在集成测试、非功能测试、性能测试和UAT测试中发现的缺陷。
- 130. 缺陷重现率。如果缺陷常常重现，软件质量就很低。

4.8 测试角色

测试角色并不是跟单个资源一一对应的。一个资源可以担任多个测试角色，一个测试角色也可以由多个资源负责。下面列出的这些角色是确保项目测试效果所必需的。一个优秀的测试人员应该具备所有这些角色的特征。敏捷项目和瀑布项目都有这些角色，只是扮演这些角色的人不同。在敏捷项目中，所有团队成员都会扮演一些测试角色，在图 4-2 中展示了一个例子，你可以看到在敏捷项目中，每个团队成员都是怎样扮演各个角色的。这并不是强制性的规定；每个团队各有差异，不过这种做法也算得上是不错的组合。

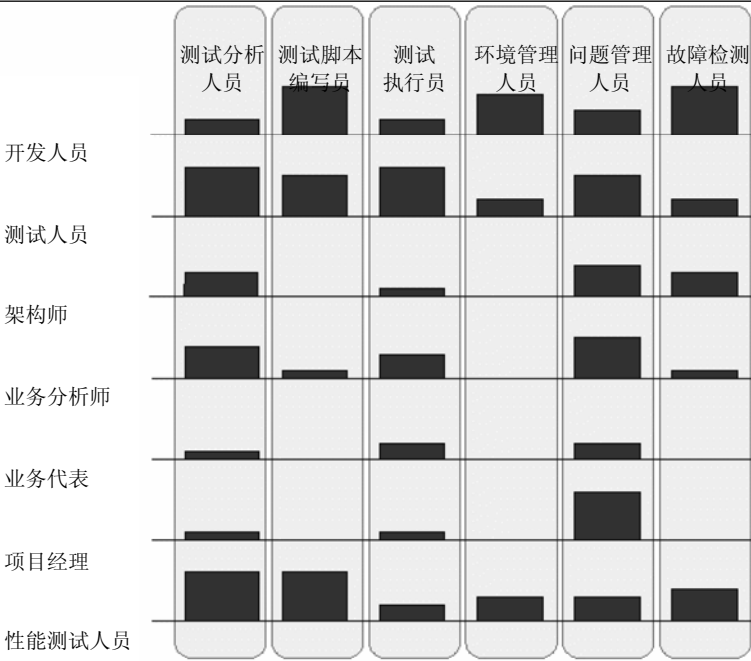


图4-2 不同团队成员的测试角色

测试分析人员

测试分析人员要了解需求、架构、代码等各个产物，从而判断哪些需要做测试，哪些是测试要重点关注的地方。

在瀑布项目中一般是有一个（或多个）资深的资源来担任这个角色。他们检查相关文档（需求、设计、架构），编写测试计划，编写高层的测试用例描述，然后把所有的东西都交给一个初级员工，让他填补详细的测试脚本。敏捷项目鼓励所有团队成员一起担任这个角色。开发人员的关注点是通过分析代码和设计来编写单元测试，但是他们也会协助业务分析师或者测试人员编写功能测试，还会参与非功能测试和性能测试的分析。业务分析师和测试人员紧密协作，编写功能测试和用户验收测试，并执行探索性测试。客户/最终用户会被邀请参与用户验收测试。

测试脚本编写员

该角色就是编写详细测试脚本的人。这些脚本可能供手工执行，也可能被自动化。瀑布项目中的脚本编写员就是个初级员工，他根据测试计划和测试用例描述来编写手册，每一步都描述的很详尽。自动化脚本编写员就得是更资深的人了，开发人员也会参与单元测试用例的编写。敏捷项目会大量使用开发人员来编写测试脚本，主要是因为测试脚本是自动化执行的。

测试执行员

不管是手工测试还是自动化测试都有这个角色，不过在自动化的时候，这个角色的扮演者就是一台电脑。测试执行员会执行详细测试脚本，判断哪些测试失败，哪些测试通过。瀑布项目一般都用测试人员来做这件事情，而敏捷项目则鼓励所有人都来参与，尤其是测试人员、业务分析师和客户。

环境管理人员

这个角色管理测试环境，包括应用程序运行的环境以及支持自动化测试的基础架构。他们还会关注外部接口和用作测试的数据。这个角色在瀑布项目和敏捷项目中很相似。

问题管理人员

问题出现以后就要解决。这个角色可以帮助筛查问题，确保它们被正确地创建，有各种属性，如严重程度、优先级、组件等等。这个角色还要管理问题的生命周期，并提供工具支持。这个角色在瀑布项目和敏捷项目中很相似。

故障检测人员

这个角色当问题出现的时候就要去做故障检测工作，判断是不是软件缺陷。如果是软件缺陷，他们就要去找出问题根源、可能的解决方案和变通措施。这个角色在瀑布项目和敏捷项目中很相似。

敏捷团队所注重的是让各个角色得到充分发挥，而比较少关心谁在做什么事情、谁对哪些事情负责。测试人员和其他团队成员之间没有界限，他们共同的目标是生产出更高质量的软件，每个成员都要尽一切可能帮助达成这个目标。在敏捷团队中，测试人员可以从所有人那里得到帮助，而他们可以帮助其他人提高测试技能。这种方式能够确保团队中的每个人都在为交付高质量应用程序而付出。

4.9 参考文献

Test-Driven Development: By Example by Kent Beck (Addison-Wesley Professional, 2002)

“Exploratory Testing Explained v.1.3 4/16/03,” copyright 2002–2003 by James Bach,
<http://www.James@satisfice.com>

James Bull, QA 咨询师

性能不彰的软件不仅不能让人们的生活变得轻松，反而会妨碍企业的正常运转，并且让使用它的人心烦。无论实现了多少功能，这样的软件都只能让用户感觉质量很糟糕。

一旦你的软件给用户留下了糟糕的体验，他们不会等到你下一次的改进，而会决定掏钱去购买别人的软件。

假如一个系统又快又可靠，伸缩性又好，而另一个在这几方面表现都不好的话，用户当然会选前者。我们这里所说的“性能测试”不仅针对性能，而且通常还包括对伸缩性和可靠性的测试，因为这些测试往往可以同时、用同一套工具来完成。在这篇文章里，我将会介绍如何确保最终产品能够具备这些好的属性——我通常把它们统称为性能。

5.1 什么是性能测试

大家应该都能同意：良好的性能不只是好东西，更是一个值得为其花钱的东西。现在的问题是：应该在哪里做性能测试？如何让性能测试帮助我们写出性能良好的软件呢？

性能测试应该囊括确保产品性能符合要求所需的一切行动。这里有四个关键点：需求、产品性能数据、沟通和流程。

这四点中一旦有所缺失，性能测试的效果就会大打折扣。假如仅仅做测试，情况并不会真正有所改观，因为你并不知道系统应该有多快。所以，你需要采集需求。假如测试的结果没有得到有效沟通，那么就没人知道问题的存在，也就不会采取任何行动来解决问题。即便我们采集到了需求，对产品做了测试，也把结果告知相关人员，工作仍旧没有结束。假如项目计划中没有给“解决性能问题”留出空间，或者没有一套流程根据测试结果计划后续的工作，那么你还是没办法对最终交付的软件产生太多影响——在这种情况下，你耗费大量成本做性能测试，结果只是让你知道产品的性能究竟有多糟糕或多强大，却对这个结果束手无策。

我们想要的不仅仅是知道结果，更希望所获得的信息能对我们正在开发的软件产生影响，从而保证我们能够满足——或者至少接近——用户对性能的要求。下面我将逐一讨论这四个关键点。

5.2 需求采集

性能需求采集的重要性经常被人们低估。在这一节里，我将尝试阐明几个重要问题：要度量什么？如何知道我们需要什么？以及如何得到确实有用（而非帮倒忙）的数据？

要度量什么

最重要的性能度量点有两个，最大吞吐量以及给定吞吐量下的响应时间。一个好的做法是：分别度量几种不同吞吐量下的响应时间，从中分析负载对响应时间的影响。如果响应的及时性非常重要，那么在确保满足响应时间要求的前提下所能达到的吞吐量可能就会明显低于最大吞吐量。你需要通过度量找出两项数据：当响应时间恰好可以接受时的吞吐量，以及达到预期吞吐量时的响应时间。伸缩性度量的关键则在于：随着数据规模、用户数量或者运行系统的硬件变化，起初得到的性能度量数据会发生怎样的变化。

可靠性的关键度量点是：当负载量高得超乎寻常，或者连续运行了很长时间以后，系统是否仍然正常工作。

如何设定目标

要想知道系统需要达到怎样的吞吐量，你首先需要知道有多少用户会使用这个系统，以及他们的使用模式。用户会多频繁地使用某个功能？这个功能需要多快完成？

业务用户会知道这些问题的答案。你应该让他们明白，你会经常需要向他们咨询这方面的事。然后你应该建立一个良好的沟通流程，以确保信息的获取畅通无阻。

总而言之，你需要有一个可靠的流程与机制来获得所需的信息，及时获知支撑业务需求所需的性能指标。如果不经常去计算这些数据，就有可能发现你正在朝着已经过时的目标努力。

弄清当前需要负载的吞吐量之后，下一个需要考虑的就是响应时间。在结合 UI 考虑这个问题时，人们常会有钻牛角尖的想法。既然用户界面要在几秒钟之内响应，那么功能自然必须在更短的时间内完成。但事实并非如此。UI 应该立即响应，告知用户：他们的请求已经得到处理；但实际的处理未必马上完成。在整个过程中，系统的其他部分应该照常工作。

响应时间的目标应该主要针对用户界面，并且数值越低越好。而且，不应该期望所有功能都能在同样的一段时间内完成。

如果对前面所说的还不明白，下面我将简单介绍一个采集性能需求的流程。

如何将性能测试融入日常开发流程

理想情况下，项目组每周应该召开一次会议，确定当前的性能需求。参加这次会议的人应该包括项目经理、关注性能的客户、资深开发者、以及性能测试人员。如果某些性能需求明显无法达到或者完全不合理，开发者需要在第一时间指出。客户的参与是为了提供业务上的信息与知识，从而帮助判断需求的合理性。项目经理需要知道团队做了哪些决定，并提供一些方向性的指导。至于性能测试人员，他们显然应该在场，这样他们才知道需要测试什么。

接下来，你需要找到适当的讨论对象。开发团队需要从客户中找到一个联系人，与他一道决定性能需求，这样才能确保客户和开发者都清楚目标所在。不要把性能需求看作神圣不可侵犯之物，和所有需求一样，它们也应该是开发者与客户对话的起点，双方需要共同讨论决定最终的目标。

一旦需求确定下来，就能决定当需求得到满足时如何向客户展示，并跟其他的任务一样对编写测试的工作进行评估和计划。

开发者需要性能测试告诉他们什么

开发者的需求有很多种，但背后的驱动力总是一致的。如果某段代码需要返工，他们就需要

更多的信息来了解当时的情况。这些信息可能来自代码检查工具，也可能来自线程转储，甚至来自日志。他们可能需要知道与应用程序服务器相比数据库的忙碌程度，或是负载达到峰值时网络的忙碌程度。

预先回答所有这些问题可能并不值得一试，因为这会需要很大工作量。我们能做的是：当问题出现时，弄清哪些信息会有助于开发者解决问题，然后把获取这些信息的任务加到你的任务列表上，并告知客户。此时你就可以考虑以下问题：从此刻开始为所有测试获取信息是否容易，这是否是针对眼下的特定问题所做的一次性测试。

如果开发者的需求是以这种方式在会议上提出的，那么，所有人都会知道这些需求的存在。客户可以为这些需求排优先级，可以把它们纳入项目计划。最终性能测试将满足各方的需求：它让客户对正在开发的软件保持信心，它也能帮助开发者找到并解决性能问题。

找不到关注性能的客户怎么办

如果找不到一个关注性能需求的客户，就会有以下风险。首先，正在开发的软件可能不符合业务要求，项目可能彻底失败。其次，不管最终的产品如何，客户都可能说它不符合要求，因为他们感觉开发团队没有征求他们的意见。最后，这可能会在团队内部造成紧张气氛，开发团队会觉得自己被迫做不必要的工作，因为需求不是来自客户——不管项目经理的担心是否正确，这种想法都有可能出现，并导致必要的工作没有被完成。亦或相反，开发者们浪费时间去做了不必要的工作。

如果客户不懂技术又非要坚持不可能的需求该怎么办

这种可能性总是存在：客户希望产品的性能达到某个水平，而达到这个水平是不可能或者不经济的。这时你就需要提出一些中肯的问题，把对话引导到真实的业务需求上来，从而打消客户不切实际的要求。

如果客户的要求是关于吞吐量的，可以考虑的问题有：每个工作日处理多少事务？这些事务的时间分布如何？是平均分布还是有明显的峰谷之分？每个周五下午会有集中访问，还是说峰值的出现没有特别的模式可循？

关于响应时间，可以考虑的问题有：用户界面的响应时间会对系统的处理能力造成什么影响？能不能把界面与实际的计算操作分离？比如说，可能有这样一种场景，用户输入一些数据，然后进行较长时间的数据处理。此时用户不希望一直等到处理完成，而是希望立即输入下一段数据。所以这时合理的期望不是在一秒钟内完成数据处理，而是将用户界面与数据处理分离，让系统在后台处理前一段数据，同时让用户在界面上输入更多的数据。

通过上述方式，我们就能让开发者和客户共同寻找一个对业务价值有意义的性能水平，并且分清什么是当务之急以及什么是锦上添花。我们都曾遇到下面这种情况：在项目的现有条件下，客户急切希望的某个性能目标不可能达到或是需要付出高昂的代价。如果相关的分析能尽早开展，客户就有可能在更早的时候做出决定，从而使这些目标成为可能。

如果客户期望的目标不能达成，他们会对最终交付的系统感到失望，哪怕系统其实足以满足业务需求。上述这些讨论有两方面的作用，不仅让开发团队了解客户的真实需求，而且让客户自己也有一个清晰的目标。这样一来，只要系统达到了双方认可的目标，客户就会感到满意。有这些讨论作为基础，客户就不太会坚持不切实际的期望；如果他们仍然感到失望，至少那也是出于合理的原因。

何不让业务分析师一并采集这些需求

采集性能需求时不一定需要业务分析师在场，原因有几点。首先，此时功能需求的采集应该已经完成了；其次，即使业务分析师在场，开发者还是不能缺席，因为只有开发者才清楚分析性能问题需要获得哪些信息，也只有他们才能判断获得这些信息的途径和难度。性能测试人员应该提出前面介绍的这些问题，以此推动讨论进行，他们也能够判断每个需求是否容易测试。所以，当这些人坐在一起讨论时，业务分析师就可以把时间花在其他更有价值的地方。

小结

需求采集是为了让所有人都清楚最终交付的产品需要有怎样的性能才能支撑业务目标。之所以要让客户参与，是因为他们最了解自己的业务，这样才能确保采集到的需求足够准确。而且通过讨论也能帮助客户清晰自己对性能的需求，从而有效管理他们对系统的期望。

5.3 运行测试

下面我将简单讨论需要运行哪些测试以及何时运行它们。

运行哪些测试

所有频繁进行的用户操作都应该有对应的测试。这些测试应该记录吞吐量、错误率和响应时间的统计数据。然后你还应该复用这些测试，从而构建更复杂的测试。所有这些测试应该一起执行，尽可能地模拟真实情况，这样你就能从中获悉产品的性能状况。

准备好这些测试以后，就可以在不同的用户量、不同的数据规模下运行它们，观察性能数据的伸缩情况。如果可能的话，还应该在不同的机器数量下进行测试，从而了解增加硬件能给性能带来多大提升。从这几方面，你就能获知产品的伸缩能力。

最后，你还应该在超负荷的情况下进行测试，从而找出系统的失败点。还应该在用户分布情况基本不变的前提下加快用户操作速度进行测试。此外长时间运行性能测试有助于了解系统的可靠性。

何时运行测试

答案显然是越频繁越好。但显然这里有个问题，性能测试的本质决定了运行它们需要很长时间。尤其是可靠性测试，只有运行的时间足够长才有意义。所以不太可能为所有的构建都运行全套性能测试。我们既希望给开发者提供快速的反馈，又希望对系统进行全面的测试。

一个解决办法是找一台专门的机器，为最新的构建执行一组有限的性能测试。如果测试结果明显有别于前一构建，本构建就应该被视为失败。从中得到的结果虽不能代表系统的真实性能，但可以作为一个早期预警系统，让开发者们能很快知道自己的工作是否严重影响了产品的性能。

整套的性能测试应该尽可能频繁地在完整的性能环境下运行，可能每天运行几次。如果对环境的访问受到限制，在晚上执行性能测试也是个不错的折衷办法。

可靠性测试显然需要更长的时间，而且通常必须与其他性能测试运行在同样的环境下，也就是说没办法在工作日里进行。所以如果没办法找到一个专门用于可靠性测试的环境，在每个周末运行可靠性测试也是个解决办法。

在何处运行测试

如果可能的话，应该尽量让性能测试环境模拟真实的生产环境。如果生产环境太过庞大而无法整体模拟，那么就应该让性能测试环境模拟生产环境的一个部分，然后将真实的性能需求等比压缩到性能测试环境的水平。

如果无法得到专用的性能测试环境，事情就会变得比较棘手。如果必须和功能测试团队共享环境，那么可以考虑在夜里执行性能测试。此时，最好针对性能测试和功能测试使用不同的数据库，并用脚本来切换数据库，这样两组互不相容的测试就不会互相干扰。当然，这样做的前提是你能够在台式机上运行你的应用程序并编写性能测试。

在夜间运行测试需要注意，此时的网络情况常会与平时有所不同。网络可能不像白天那么繁忙，因为人们没有在工作；但也有可能数据备份或是别的批处理任务被放在晚上进行，因此占用大量网络流量。如果在性能测试的过程中突发大量网络活动，测试结果有可能受到明显的影响；如果性能测试和造成网络占用的任务恰好被计划在同一时间进行，测试结果就可能总是受到影响，以至于你看不出这种影响的存在，除非换个时间来运行测试。为此，应该安排在正常工作时间也运行一次性能测试，这样你才知道运行测试的时机是否是对结果造成明显影响的原因。如果不同时间运行测试的结果有很大差异，那么你可以尝试模拟正常工作时间的平均网络流量，或者看这种结果差异是否保持一致，如果差异始终一致，就在查看结果时将其考虑在内即可。

根据我的经验，如果对测试环境存在明显的争用现象，就很有必要对其加以管理，谁在什么时候使用系统，使用哪个数据库，都应该事先规划好。有时测试环境下运行的测试会失败，在本地机器上却能通过，这时就需要把环境切换到性能测试的数据库，在问题修复之前其他 QA 都不能使用该环境。由于共享一个测试环境会限制运行性能测试的频度这一客观困难的存在，我们应该尽量尝试用一个单独的环境来运行性能测试，即便这个环境的配置与生产环境不很相似也没关系。

假如你面前摆着两个选择：一个测试环境与生产环境差异较大；另一个测试环境很接近生产环境，但只能在有限的时间（例如半夜）使用。你会怎么选？正确的答案是两个都要。你可以在独占的环境里编写性能测试，并且不受阻碍地频繁运行它们，这样你就可以把这组测试加入到持续集成系统中。接近生产配置的测试环境则是一个有价值的参考，你可以将其中得到的测试结果与从日常频繁运行的非生产系统中得到的测试结果相对比，从而了解这些测试结果与系统的最终性能究竟有何关系。

较小测试设备上的测试结果与生产环境的性能有何关系

一个常见的问题是测试环境的配置和生产环境不同。必须知道，如果测试环境和生产环境毫无相似之处，那么也就无法判断硬件对系统性能的影响。所以，如果不得不用一个较小的环境来做性能测试，应该怎么做？我的建议是模拟生产系统的有代表性的部分。下面我将介绍具体的做法。

以一个大容量 web 应用程序为例。系统的基本架构可能包括几台应用程序服务器、几台 web 服务器和几台数据库服务器。假设生产系统有 N 台数据库服务器（都是很高配置的机器）、 $2 \times N$ 台应用程序服务器（配置较高）和 $4 \times N$ 台 web 服务器（配置较差），那么你就可以考虑我的办法：准备一台数据库服务器，其性能大约是生产数据库服务器的一半；准备一台应用程序服务器，其配置和生产环境的应用程序服务器一样；再准备一台 web 服务器。

现在你就拥有了一个应用程序服务器与数据库服务器的组合，两者之间的相对性能比与生产环境一致，绝对性能值则大概降低了一半，并且 web 服务器的配置也不足。

在这个环境下，你可以直接访问应用程序服务器，借此了解应用程序的性能，从而掌握一台应用程序服务器所能达到的性能水平。然后你可以通过 web 服务器来访问，并且选择一种让 web 服务器成为瓶颈的测试场景，于是你就可以掌握一台 web 服务器所能达到的性能水平，而不会受到应用程序的影响。根据这两组数据，你应该能比较准确地判断在生产环境下各种服务器的配比是否合适，并且对抱有一定程序信心的生产系统的性能有所预估。

需要记住的是由于每个 web 请求只能由一个应用程序服务器/数据库服务器的组合来处理，因此当服务器数量增加时，只有吞吐量的提升是能够确定的。而响应时间只会因为每台服务器的 CPU 计算能力或者内存增加而提升——假设系统负载水平保持不变的话。其原因在于，更快的 CPU 能以更快的速度处理更多请求，而更多的内存则让更多的信息得以缓存。

当然，以上讨论都基于一系列假设：机器配置始终保持不变，CPU 都是同一家厂商制造的，操作系统都一样，数据库/web 服务器/应用程序服务器的组合也保持不变。

请时刻牢记，测试环境与生产环境在机器配置、软件等方面差异越大，对真实系统的性能估计就越不准确。在前面的例子中，你可以把测试环境看作生产环境的一部分，从中估算出的生产环境性能就不会谬以千里。如果机器的配置与生产环境毫无可比性，用的软件也全然不同（例如把 Oracle 改成 MySQL，把 JBoss 改成 WebSphere），你仍然可以用这个环境来度量性能的变化情况，但根据测试结果估算出的生产环境性能数据就将非常可疑。

应该用多大规模的数据库来做性能测试

在做性能测试时必须记住，数据库的规模会显著影响从表中取出记录所需的时间。如果一张表没有合适的索引，当数据规模较小时可能还看不出问题；然而，一旦有几千行以上的生产数据，性能就会严重降低。

应该先与关注性能的客户交流，争取拿到一份生产数据库的副本，这样就可以针对它来进行测试。在这个过程中要注意数据保护，并对拿到的数据库做适当的清理，删除或修改其中的私密信息。

你还应该与客户探讨数据规模发生变化的可能性。数据量会大致保持在现有水平上吗？还是很可能会增长？如果会增长，增长的速度会有多快？只有了解这些信息，你才知道是否应该用一个比现在大得多的数据库来做性能测试。

要得到一个更大的数据库，最好的办法就是使用稳定性测试创建新的数据库。在稳定性测试中，你应该会创建新的用户和新的交易数据。如果这组测试整个周末都在顺利运行，那么你就能得到一个适用于未来情形的数据规模了。

如何处理第三方接口

如果系统用到很多第三方接口，性能测试最好不要直接去使用这些第三方系统。原因有两点：首先，第三方系统可能并不适合成为性能测试的一部分；其次，即便第三方系统提供了测试环境，依赖你无法控制的第三方系统会降低测试的可靠性。

最好的办法是用一个单独的测试来获知第三方系统的平均响应时间，然后为它写一个 mock 或者 stub，直接等待那么长的一段时间然后返回一个固定的响应。当然也可以直接返回响应，不过这样就会让测试失去了一些真实性，因为有了等待第三方系统的时间，应用程序服务

器就能够更快地释放数据库连接或者网络连接，这会给最终的测试结果造成差异。

需要多少种测试案例

这是个重要的问题，因为不恰当（过多或者过少）的测试量会严重歪曲测试结果。如果测试案例太少，所有相关的信息都会被缓存起来，系统就会显得比实际情况要快；如果测试案例太多，缓存就会溢出，系统就会显得比实际情况要慢。

多少种测试案例才是合适的呢？你需要和关注性能的客户讨论系统的预期使用情况，并且如果可能的话，请分析现有系统的使用日志从而找到一个答案。比如说，如果要测试的场景是“从应用程序获取顾客信息”，那么要在测试中覆盖到的顾客数显然和正常操作中涉及的顾客数有关。如果正常情况下系统中每天有 5% 的顾客信息记录会被取出，那么你的测试也就应该覆盖这么多的顾客。

为何要多次度量响应时间和吞吐量

一般来说，在从空闲状态开始增加负载量时，系统响应时间不会有什么变化。随着负载量不断上升，到达某一个点之后，尽管单位时间内处理的事务量仍然在上升（即吞吐量继续上升），但每个请求的响应时间会受到影响而逐渐上升。当服务器达到能力上限时，起初吞吐量会保持不变，而同时响应时间显著上升；最终吞吐量会急剧下跌，因为计算机已经无法承担所要求的如此大量的工作。这时响应时间将会飙升，整个系统会濒临死机。

在这个过程中，我们对几方面的信息感兴趣。首先，我们希望知道系统最大吞吐量出现在哪个点。除此之外，我们还希望知道最佳响应时间、当响应时间正好符合要求时的负载量以及负载达到事先测量的最大吞吐量的 80% 和 90% 时的响应时间等信息。

有了这组数据，你就可以限制每台应用程序服务器的连接数，从而确保系统性能始终保持在性能需求所规定的水平以上。可以看到，当负载量逼近极限时，响应时间会急剧上升；而当负载量达到 80% 甚至 90% 时，响应时间却没有太多变化。如果你必须确保某个性能水平，这一现象应该始终牢记于心。

有必要测试所有功能吗

对系统的所有功能进行测试基本上是不现实的。重点在于要覆盖最常用的功能。所以你需要识别出系统的主要使用场景，并针对这些场景创建不同的测试。

比如说，在线购物网站最主要的使用模式应该是“浏览”和“购买”。来购物的人不全会浏览很多页面，浏览的时间通常也不都会太长。所以你需要创建一个“浏览”的测试脚本和一个“购买”的测试脚本。为了让测试脚本更贴近真实，你需要知道用户浏览商品的平均数量、每次购买的平均商品数以及正常情况下一天时间内被浏览过的商品数占商品总数的百分比。

小结

关于性能测试，有很多话题可以讨论。需要度量什么？需要多频繁的测试？需要编写多少脚本？需要多少数据？在与关注性能的客户进行日常讨论时，主要应该确保这些重要的问题被提出来，并保证能得到你需要的信息。如果你认为事情的发展方式会严重影响测试的进展，也应该找时间与项目经理和客户沟通。

5.4 沟通

就测试的结果进行沟通是很重要的。所以接下来的问题就是，我们究竟在沟通些什么？所谓就测试结果进行沟通，可不止是报告几个数据那么简单。如果你这样做的话，团队里的所有人在身担其他任务的情况下都得花时间来分析这些结果数据。如果先对测试结果做一些基本的分析和解读，并给出一段概述，其他人理解起来就会容易得多。

所以你需要根据讨论出的性能需求和目前的性能水平来解读测试结果。首先，你需要指出系统性能与目标的接近程度（与目标的差距有多少，或是超出目标多少）。其次，你需要说明产品的性能是否发生了重大变化。不管这种变化是否导致产品无法达到性能目标，都应该将相关的信息告知所有相关人员。引起这种变化的原因可能是新增了一大块功能，这时对产品性能的影响是无可避免的，几乎没有能够改进的空间；但也可能是因为别的小问题，例如数据库缺少了某些索引，这样的问题应该很容易解决。

谁需要知道测试结果

有三组人需要了解测试结果：开发者、项目经理以及客户。开发者和项目经理应该在测试运行完毕之后立即知道结果，这样他们就能在问题出现之初尽快合理地将问题解决。另一方面，没必要每天都拿一些小问题去打扰客户，否则当你说到真正重要的问题时他们就不会全神贯注；但也不应该疏忽与客户的交流。你可以每周安排一次会议，定期把性能测试的结果通报给客户。

此外，要记住不同的人会对不同的信息感兴趣。客户和项目经理可能希望看到比较高层面的概述，而开发者则对原始数据（以及给定时间内的响应数量等内容更感兴趣。如果能把适当的信息提供给不同的人，就能更有效地沟通产品的状态。

是否只需要写一份报告

不完全是。你当然可以写一份报告，然后用邮件发给所有人，但问题是大部分人很可能不会看这个报告，于是你想要传达的信息也就丢失了。报告只是一种用来帮助你传达信息的工具，而不是你的最终目的。

用一个网站向所有人展示最新的性能测试结果是很实用的。然后当你走到某个人的座位跟前讨论性能测试结果时，你就可以打开这个网站，把你发现的情况指给他看。不幸的是，大多数人并不善于看测试报告，所以为了确保你的信息能够传达到位，唯一的办法就是走到别人的座位旁边，或是拿起电话，向别人解释整个测试报告。

小结

你的目标是建立这样一种沟通机制：由于性能需求已经很清楚，因此无须拿着每次测试的结果去问客户是否可以接受；在每周介绍项目的当前性能状况时，你只需指出测试结果中的异常之处，并向客户解释异常情况出现的原因；如果某个区域的性能特别差，但经过判断这不是一个严重的问题，你就应该告诉客户为什么这块功能比较慢，为什么项目经理认为这不是一个高优先级的问题；如果客户不同意这个决定，那么项目经理和客户就应该坐下来具体讨论当前的情况。

5.5 流程

性能测试经常被放在项目结束前进行，这种安排严重影响了性能测试的效果。性能测试中最

重要的事就是要定期地进行测试。如果直到项目最后几周才做性能测试，那么你将有很多事要干，而时间却非常紧迫。大部分时间会被用于编写测试脚本，并得到一些和产品有关的数据。这时你就会处于一种尴尬的境地，你大概知道系统运行得多快，但基本无从知道它是否足够快，而且也没有时间做任何改进。

当第一段代码被编写出来，性能测试就应该开始了。虽然这时可能还没有任何可供测试的东西，但还是有很多事可以去做。你可以向开发者了解他们将要使用的技术，评估合适的工具，找出功能足以测试当前产品的工具。此外还需要识别出关注性能的客户，并且与他们一起启动需求采集的流程。

如何把各种工作连接起来

从这个阶段起，你的工作就开始进入一个循环。每周开始时，你会第一时间与关注性能的客户开会，讨论当前正在开发的性能需求，同时介绍你的测试计划，以及如何展示需求得到了满足。客户也可以在这时要求更多的测试。剩下的几天里，你可以为最近完成的功能编写性能测试，维持已有的自动化测试，以及查看测试结果。一周将要结束时，你再次和关注性能的客户开会。这个会议有两项任务：首先是向客户展示本周编写的性能测试，并和客户讨论这些新的测试是否能表示产品满足早先提出的性能需求；其次是与客户一起查看现有性能测试的最新结果。

如何确保不拖后腿

只要按照这个每周的循环在工作，一旦性能测试的进度滞后，你很快就能清楚地看到。这时要想赶上进度，你可以增加用于性能测试的资源，也可以减少工作量。至于具体怎么做，很大程度上取决于性能需求对于项目有多重要。

你可以建立一个任务列表，将本周与客户所决定执行的测试任务都写在上面。然后你就可以与客户一起对这些测试排定优先级。每周你尽量完成列表上的任务。如果这样做下来导致测试覆盖率很成问题，那可能你需要投入更多人手来做性能测试；但也有可能在扔掉一些高难度、低优先级的测试之后，你完全能够保证足够的测试覆盖率，而又不会拖项目后腿。

如何确保每个问题都得到解决

必须在项目开始之初就和项目经理沟通，决定如何修复性能问题。你需要确保项目经理认同你的工作方式和采集到的性能需求；你还要确保项目经理同意将性能问题作为 bug 提出，并且一旦性能问题出现就会有所行动，否则你就只能在项目结束时对着一大堆已知的性能问题徒呼奈何了。毕竟，如果性能问题出现之后不采取措施去解决，那么即使测出当前的性能水平也是毫无意义的。

5.6 总结

这个流程最大的好处在于它能确保你知道自己手上有什么、需要什么，而且你能肯定系统的每个部分都有测试覆盖，从而大大增加了发现问题解决问题的机会。让性能测试与开发同步，对每个功能都有测试覆盖，这样如果性能出了问题你就有时间应对。有一份性能需求在手上，你就能判断当前的系统是否需要改变。这份需求是客户根据业务流程和规模制订的，所以整个团队都对它有信心，大家也会乐于花时间来解决问题，因为他们知道这是一件有价值的工作。

线下定期举行的技术讨论俱乐部

Qclub

分享

交流

我们影响有影响力的人



主题：围绕InfoQ关注的领域设定话题

形式：1个主题、1+个嘉宾、N个参会人员

参会者：有决策权的中高端技术人员

人数：限制人数，保证每个人的发言权利

频率：每月一次