第 3 章 版本与开发任务管理

本章学习目标

- 了解配置管理的基本内容以及软件产品发布计划的主要过程。
- 理解常用的软件产品版本命名规则。
- 掌握代码版本与分支管理的基本概念和过程以及分布式版本控制系统 Git 的使用。
- 理解特性开发任务管理流程、变更管理流程以及缺陷修复过程管理。
- 了解基于追踪与回溯的工作量与质量分析方法。

本章首先介绍配置管理的基本内容以及软件产品发布计划的主要过程；接着介绍软件版本管理，包括软件产品版本命名、代码版本管理、代码分支管理等；然后介绍特性开发任务管理流程、变更管理流程以及缺陷修复管理流程；最后介绍基于追踪与回溯的工作量与质量分析方法。

需要注意的是，本章介绍的版本与开发任务管理在很大程度上属于配置管理的范畴。配置管理是贯穿整个软件系统生存周期的重要质量保证活动，实现软件开发和演化历史的维护和跟踪并进行变更管理和控制。本章将主要介绍开发人员参加团队开发首先需要了解的版本管理与开发任务管理，配置管理中的构建管理与发布管理将在第 10 章介绍。

3.1 版本与开发任务管理概述

加入一个开发团队并参与软件开发任务首先需要了解开发任务是如何分配和管理的，不同开发人员的开发任务是如何相互协调的。为此，我们需要了解版本管理和开发任务管理，它们在很大程度上属于配置管理的范畴。本节首先对配置管理进行概述，然后对软件项目的版本发布计划进行介绍。

3.1.1 配置管理概述

软件开发过程中会产生许多软件制品，包括需求模型、设计模型、源代码、可执行文件、测试用例等开发产物以及技术文档、计划文档、会议记录等相关文档。另一方面，软件在开发和使用过程中不可避免地会发生变更。例如，客户或用户需求会发生变化，为此开发人员需要编写代码实现这些需求变更；测试人员在测试时或者用户在使用软件时发现了缺陷，为此开发人员需要修改代码以修复所发现的缺陷；软件为了保持市场竞争力需要引入新的特性，为此开发人员需要编写代码实现这些特性。一旦软件发生变更，就会产生一个新的软件版本。此外，软件开发通常都是以一种增量和迭代的方式进行的，在此过程中开发人员也需要不断编写和修改代码，从而产生一系列软件版本。为了确保软件开发和变更有序进行

并向客户发布正确的产品版本，需要有一整套相应的管理方法和工具，否则整个过程就会变得混乱而且容易出错。

配置管理为软件开发提供了一套方法、流程和工具来维护和管理软件的版本演化及产品发布，有效地存储和跟踪软件的所有变更与版本历史。如果没有配置管理的支持，开发人员可能会无法了解软件的演化历史或者获取特定版本的软件源代码，也无法协调多个开发人员的文档和代码修改行为，从而导致开发人员之间的修改互相干扰甚至交付错误的软件版本给客户等严重问题。软件的配置管理主要包含以下四个方面的内容。

- **版本管理**：规范软件版本命名，制定软件版本发布和迭代计划，跟踪软件的变更与版本历史并确保不同开发人员的修改不会彼此干涉。
- **开发任务管理**：开发任务主要包括由正向需求分解引出的特性开发任务以及由软件缺陷引发的缺陷修复任务。开发任务管理需要对开发任务进行规范描述，追踪开发任务的处理流程，同时对变更请求进行决策，跟踪变更请求的处理与实施。
- **构建管理**：管理软件的外部依赖（例如第三方库），对代码、数据和外部依赖等软件制品进行编译和链接从而生成可执行的软件版本，同时运行测试以检查构建和集成是否成功。
- **发布管理**：在软件构建结果的基础上打包形成可发布的软件版本并进行存档，提供回溯和审查，持续跟踪供客户和用户使用的已发布软件版本。

3.1.2 版本发布计划

企业的软件项目一般以一年为一个规划周期，进行软件项目的持续规划、开发与发布。如图 3.1 所示，企业可以规划每月发布一个迭代版本（Beta），每季度发布一个稳定的发布版本（Release）。所有的版本计划和版本发布都需要通过评审，而每个稳定版本的发布都需要通过阶段审视，从而完善并改进开发团队的管理方法与流程、开发人员的技能、工作量的计划与分配等。

版本发布计划需要明确版本定位、目标、商业价值，并着重规划里程碑，即在什么时候发布版本，以及估算用户故事（即一种简短的故事性的需求描述，详见 8.4 节）及其优先级。在此基础上，就可以通过估算开发团队在每轮迭代中完成的工作量来制定发布计划，即需要多少轮迭代完成版本发布以及在每轮迭代中需要完成哪些用户故事。一般而言，版本发布计划主要包括以下五个主要活动。

1. 规划里程碑

一般而言，开发团队与客户可以规划一个日期范围而不是一个具体日期来作为里程碑，如“我们希望在 7 月发布新版本，但是在 8 月发布也是可以接受的”。以日期范围作为里程碑可以让发布时间变得更加灵活，也增加了迭代开发的容错能力。

2. 估算用户故事

在敏捷开发中，一个用户故事所需的工作量一般以故事点作为基本单位来衡量。每个开发团队可以自己定义故事点的标准，例如，可以定义为以理想工作日（即一天中没有任何其他事情的打扰）为单位的工作量，也可以定义为针对用户故事复杂程度的量化数值。前者在实践中应用更加广泛，即将一个故事点定义为一个理想工作日。为了估算一个用户故事的工作量，通常可以采用一种迭代式的方法。首先，由项目组长随机抽取一个用户故事，

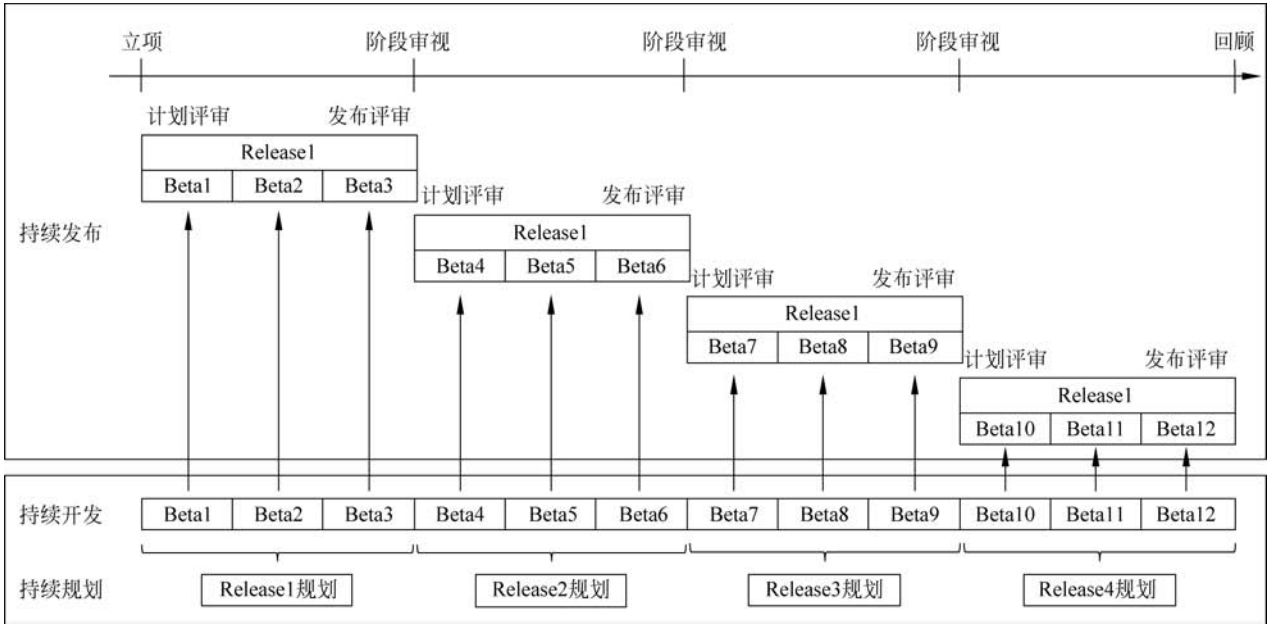

图 3.1　持续规划、开发与发布流程

并向所有开发人员解释该用户故事。开发人员根据需要可以向项目组长提问以便更好地理解完成用户故事所需的工作，而项目组长也要尽其所能地解答。接着，每个开发人员独立地在一个卡片上写下对该用户故事的故事点估算。当每个开发人员都完成估算后，所有开发人员同时翻开卡片。当估算值相差很大时，估算值最高和最低的开发人员解释其估算依据，并进行用户故事讨论。完成讨论后，所有开发人员进行下一轮的故事点估算并展示给所有开发人员，直到所有开发人员的估算值达到统一。通常情况下，在第二轮迭代后估算值就能达到统一了。

3. 排列用户故事优先级

为了制定发布计划，必须要明确各个用户故事的优先级。一般而言，可以通过多个方面的权衡来为用户故事进行优先级排序：考虑用户故事对客户带来的价值，例如，一个用户故事所蕴含的新特性可能会大幅提高用户满意度因此具有较高优先级；考虑用户故事所蕴含的不确定性和风险，例如，一个用户故事中存在性能指标上的风险，可能对软件系统的整体架构带来影响，因此需要优先处理；考虑用户故事之间的依赖关系，例如，一个用户故事是实现其他用户故事的基础，因此需要先实现。通常，高价值、高风险的用户故事具有最高的优先级，可以让客户尽快看到最有价值的特性，也能尽早消除不确定性从而避免后期风险的巨大影响。高价值、低风险的用户故事具有较高的优先级，低价值、低风险的用户故事具有较低的优先级，而低价值、高风险的用户故事具有最低的优先级。

4. 估算开发效率

通过一个开发团队在一轮迭代中能完成的故事点来衡量开发效率。一般可以通过三种方法来估算开发团队的开发效率。第一种方法是使用历史值，即当一个开发团队做过类似的软件项目且没有人员变动时，就可以使用之前项目的开发效率。第二种方法是执行一轮迭代以获取开发效率，但是这种方法会推迟发布计划，因此客户往往难以接受这种方法。第三种方法是猜测。如果一个故事点是一个理想工作日，可以通过估算完成一个理想工作日的工作需要多少实际工作日来估算开发效率。一般而言，剔除会议、报告演示、回复电子邮件等干扰事项，一轮迭代的三分之一到一半的时间可以作为开发效率。例如，一个开发团队由 5 人组成，一次迭代周期为 4 周(即 20 个实际工作日)，那么每轮迭代共有 100 个实际工作日，而该开发团队的开发效率可以估算为 30～50 个故事点。需要注意的是，开发团队的开发效率估算可以随着迭代的进行而不断地优化和完善。

5. 创建发布计划

创建发布计划时需要确定迭代轮数，并把用户故事分配到每轮迭代中。迭代轮数可以通过软件版本的总故事点数以及开发团队的开发效率来决定。例如，一个软件版本总共有 180 个故事点，一个开发团队的开发效率为 30 个故事点，那么可以预计总共需要 6 轮迭代开发就能完成版本发布。在分配用户故事时，先选择优先级最高的 30 个故事点并将这些用户故事放入第一轮迭代中，再选择优先级次高的 30 个故事点并将这些用户故事放入第二轮迭代中，如此进行分配直到分配完所有的用户故事到 4 轮迭代中。需要注意的是，版本发布计划完成后，也不是一成不变的。故事点、用户故事优先级、开发效率等会根据实际情况及时进行调整。例如，由于客户需求的变更，可能会导致用户故事的优先级发生变化。

版本发布计划将用户故事按照故事点和优先级分配到每轮迭代中，优先级高的用户故

事安排在较早的迭代中。在执行一轮迭代时，还需要版本迭代计划来规划每一轮的迭代，包括由项目组长组织所有开发人员讨论用户故事，直到开发人员理解用户故事并能从用户故事中分解出任务，以及由开发人员认领并承担各个任务的责任。确保完成任务是每个开发人员的责任。然而，如果有开发人员在迭代快要结束时不能完成所承担的任务，其他开发人员应该尽量提供帮助。此外，任务的职责承担分配在迭代过程中并不是不变的。例如，有些任务可能比预估的简单，而有些任务又比预估的困难。因此，任务的职责承担分配需要在迭代过程中适时地进行调整。

3.2 版本管理

软件开发和演化过程伴随着各种软件制品的持续变化，其中既包括源代码文件、配置文件、文档等原子性的制品文件，又包括模块、组件等复合性的软件制品乃至整个软件产品。版本管理需要将这些软件制品都置于系统性的管理之中，进行版本标识，追踪演化历史并确保开发人员在并行协作开发过程中不会相互影响。

3.2.1 产品版本号命名

不管是直接面向客户交付的软件产品（例如一个定制化的信息管理系统）还是通过应用市场分发的软件产品（例如移动应用），软件开发企业都需要经常进行发布版本的迭代更新。为了明确标识并区分不同的发布版本，软件开发企业需要为每一个软件发布版本分配一个唯一的版本号。为此，软件开发企业需要制定软件版本号命名规范，用于规范化管理软件产品的版本号。目前使用较广泛的是点分式版本命名规范，其格式是 M. S. F. B([SP][C])，共由六部分组成。

- 主版本号 M(Major Version)：标识产品平台或整体架构。M 版本号由两位数字组成，从 1 开始以 1 为单位递增编号到 99，不足两位不补位。当软件产品平台或整体架构发生变化时，M 版本号变化。
- 次版本号 S(Senior Version)：标识局部架构、重大特性或无法向前兼容的接口。S 版本号由两位数字组成，从 0 开始以 1 为单位递增编号到 99，不足两位不补位。当软件产品的局部架构、重大特性或无法向前兼容的接口发生变化时，S 版本号变化。当 M 版本号升级时，S 版本号清零。
- 特性版本号 F(Feature Version)：标识规划的新特性版本。F 版本号由两位数字组成，从 0 或 1 开始以 1 为单位递增编号到 99，不足两位不补位。当软件产品支持的特性集发生变化时，F 版本号变化。当 S 版本号升级时，F 版本号清零。
- 编译版本号 B(Build Version)：标识编译构建的版本号。B 版本号由三位数字组成，从 0 或 1 以 1 为单位递增编号到 999，不足三位不补位。当软件产品重新编译构建时，B 版本号变化。当 F 版本号升级时，B 版本号清零。
- 补丁包版本号 SP(System Patch Version)：标识累计一段时间的补丁，即把一段时间的补丁打包出一个补丁包。SP 版本号由三位数字组成，从 001 以 1 为单位递增编号到 999。当软件产品发布新的补丁包时，SP 版本号变化。当 B 版本号升级时，SP 版本号清零。

- 补丁版本号 C(Cold Patch Version)：标识一个补丁。C 版本号由两位数字组成，从 01 以 1 为单位递增编号到 99。当软件产品发布新的补丁时，C 版本号变化。当 B 版本号升级时，C 版本号清零。

其中，[]表示可选字段，()采用英文半角括号，仅有可选字段时用()。因此，以上各个部分中，补丁包版本号 SP 和补丁版本号 C 是可选的，而这两个版本号都是在维护阶段使用。版本号一旦明确，一般不允许随意修改。如果一定要进行修改，那就必须发起版本号变更申请，只有通过变更决策才能修改版本号。此外，还有一些其他的版本号命名规范，如语义版本规范 X. Y. Z(X 表示主版本号，标识出现不兼容的 API 变化；Y 表示次版本号，标识新增向后兼容的功能；Z 表示补丁版本号，标识修复向后兼容的缺陷)(Preston-Werner，2013)以及 VxxxRxxxCxx[SPC/CPxxx](V 标识主力产品平台的变化，R 标识面向客户发布的特性及变化，C 标识功能增强性小版本/修复缺陷的维护版本，SPC 标识补丁包，CP 标识一个补丁)。

3.2.2 代码版本管理

如果你是一个软件开发团队的开发人员，那么你所参与开发的软件经常都会处于不断的演化之中。考虑以下几个场景。

1. 代码演化历史跟踪

你可能花了两周时间，写了 6000 行代码来实现一个客户或产品经理所要求的新特性。你也可能花了两天时间，改动了 30 个文件来重构代码以提高软件的可维护性。你还可能花了 3 个小时，改动了 4 个方法来修复了一个测试人员报告的缺陷。这些软件开发活动都需要新增或修改代码。如果没有代码版本管理，你很快就会记不得为何、在何处以及如何修改的代码，从而导致自己以及他人难以理解代码的变动过程、开展代码评审以及及时发现修改过程中引入的缺陷。

2. 历史版本回退

新特性上线后如果不符合客户或市场要求，那么需要去除这一新特性，或者由于增加新特性、代码重构或其他原因而导致重要的功能不可用，这些情况下你可能想回到此前的一个稳定的代码版本上。如果没有代码版本管理，将很难准确而快速地回退到指定的代码版本，从而增加了代码维护的难度。

3. 多人开发协作冲突

你所参加的软件项目通常都是多人协作开发的，这意味着可能会有多个人修改同一个文件，从而造成冲突。因此，你可能花了很长时间修改了一个代码文件，然后发现这个文件已经被其他开发人员修改或者删除了；或者修改完代码提交后过了一段时间发现被其他开发人员覆盖了。这些情况都会导致协作冲突，从而造成修改内容丢失以及开发上的混乱。如果没有代码版本管理，开发团队，尤其是规模较大的开发团队，将难以管理和解决开发过程中的协作冲突。

4. 代码质量检查

你开发的代码可能有质量问题，例如，功能缺陷或者代码可理解性、可维护性上的问题。你可能把有质量问题的代码合入到了代码库，不仅可能造成潜在的缺陷隐患，而且可能会影响其他开发人员的代码理解和修改。因此，需要根据一些基本的规范要求来检查需要合入的代码，如代码自检、代码同行评审以及门禁检查等。只有通过这些检查的代码才允许提交

到版本库或者合入主干分支，从而达到保障软件代码质量的目的。如果没有代码的版本管理，开发团队将难以有效地管控开发人员所提交的代码质量。

以上这些场景都需要代码版本管理支持。版本控制系统是实现上述代码版本管理目标的一种有效途径。它可以存储、追踪代码修改的完整历史记录(即版本库)，同时提供多种机制帮助开发人员进行协同开发。在此基础上，还可以实现代码合入前的质量检查等门禁检查功能。版本控制系统主要分为两类：集中式版本控制系统和分布式版本控制系统。

集中式版本控制系统中的版本库集中存放在中央服务器上。如图 3.2 所示，当开发人员开始工作时，需要先从中央服务器拉取工作文件的最新版本；当开发人员完成工作后，需要将工作文件的更新提交到中央服务器。在此过程中，开发人员的客户端机器上不会存储完整的版本库，而只存储了所拉取的中央服务器上的文件快照。因此，开发人员必须联网才能工作，而中央服务器的单点故障会影响到整个开发团队。如果中央服务器宕机了，那么所有开发人员都无法拉取最新版本和提交更新，也就无法协同工作。此外，如果中央服务器在没有备份的情况下发生磁盘损坏，那么将丢失所有版本库数据。目前，集中式版本控制系统已经逐渐被分布式版本控制系统所取代，但在一些遗留系统中由于迁移代价过大还在继续使用。常用的集中式版本控制系统有 CVS 和 SVN。

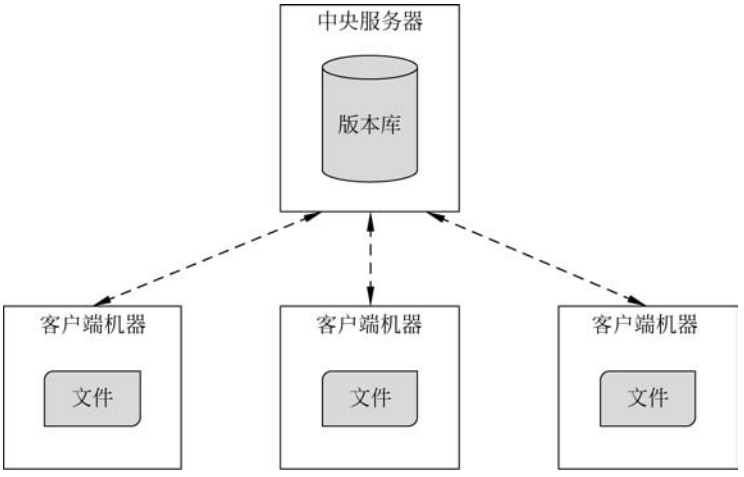

图 3.2　集中式版本控制系统

分布式版本控制系统中每个开发人员的客户端机器上都存储着完整的版本库。如图 3.3 所示，每个开发人员都有版本库的本地副本或者克隆，即每个开发人员维护着自己的本地版本库。当开发人员完成工作后，可以把工作文件的更新提交到本地版本库。因此，开发人员不需要联网就可以工作。理论上讲，在协同工作时，各个开发人员可以把各自的更新推送给其他开发人员，开发人员之间就可以互相看到各自的更新了。而在实际开发过程中，由于各个开发人员可能不在一个局域网内，或者某个开发人员并没有开机工作，因此很少在开发人员之间推送版本库的更新。取而代之的工作模式是在分布式版本控制系统中设置中央服务器，但这个中央服务器仅用来方便管理多人协同工作，即每个开发人员可以拉取中央服务器上的最新版本库，也可以将本地版本库的更新推送到中央服务器上的版本库。任何客户端机器都可以胜任中央服务器的工作，它和所有客户端机器没有本质区别。如果中央服务器

发生宕机或者磁盘损坏，丢失的版本库数据可以从各个开发人员的本地版本库中恢复。此外，分布式版本控制系统具有灵活的、强大的分支管理策略。目前，分布式版本控制系统是最流行的版本控制系统。常用的分布式版本控制系统有 Mercurial 和 Git。

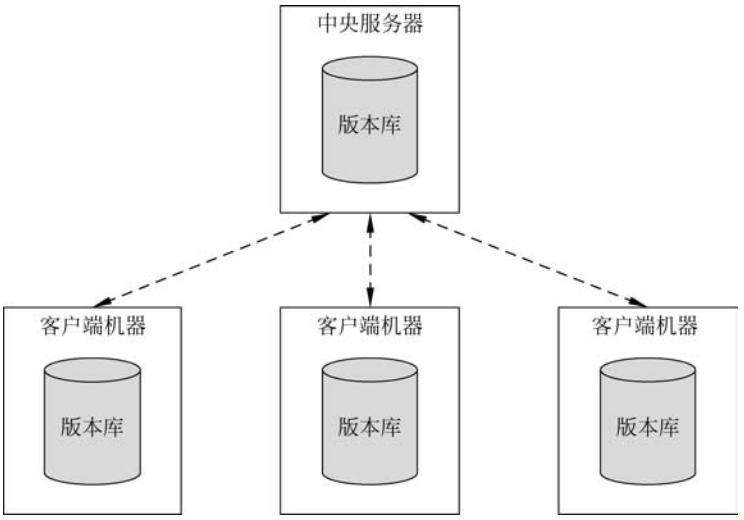

图 3.3　分布式版本控制系统

表 3.1 总结了集中式版本控制系统与分布式版本控制系统的主要区别。

表 3.1　集中式版本控制系统与分布式版本控制系统的主要区别

集中式版本控制系统	分布式版本控制系统
集中式架构	分布式架构
客户端本地没有完整版本历史	客户端本地保存完整版本历史
只能联网提交	可离线提交(本地仓库)
以目录的方式管理分支，不够灵活	强大的分支管理能力

Git 是目前最流行的分布式版本控制系统，其基本工作流程如图 3.4 所示。当你加入一个软件项目的开发团队后，可以通过 clone 命令将中央服务器上该项目的远程仓库复制到本地机器上，构成了本地仓库，而机器上项目文件所在的目录就是工作区。

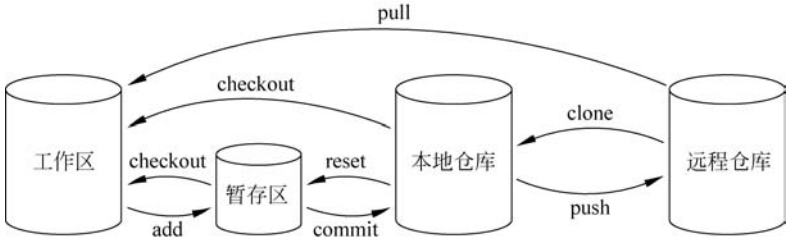

图 3.4　Git 基本工作流程

当你在工作区修改或者新增了项目文件后，可以通过 add 命令将指定的项目文件保存到暂存区，即暂时保存对指定项目文件的更改。当你完成了一件原子性的任务后(如修复了一个缺陷、重命名了一个类属性)，可以通过 commit 命令将暂存区中的文件提交到本地仓

库。当你需要把你的提交集成到项目并推送给其他开发人员时，可以通过 push 命令将本地仓库中的本地提交推送到中央服务器的远程仓库中。

当你发现一次提交的内容有错误并想撤销这次提交时，可以通过 reset 命令将暂存区重置到这次提交之前的状态，同时也可以选择是否将工作区也重置到这次提交之前的状态。当你在工作区改乱了某个文件并想直接丢弃对该文件的修改时，可以通过 checkout 命令将该文件重置为暂存区或者本地仓库中的文件内容。当你在工作区改乱了某个文件，并添加到了暂存区时，可以通过 reset 命令将暂存区中的该文件重置为本地仓库中的文件内容。

当你需要在其他开发人员的开发基础上继续协同开发时，可以通过 pull 命令将中央服务器上远程仓库的所有最新提交全部拉取到本地仓库，并与本地仓库进行合并。

由上述基本流程可见，提交(Commit)是记录项目文件历史并实施版本管理的最小单位，也是开发人员理解项目演化的基本单位。因此，提交的粒度需要保证每次提交只做一件事情，而不要同时把多件事情混在一次提交中(例如，在一次提交中修复了多个缺陷，或者在新增特性的同时修复缺陷)，否则将增加代码理解和评审的难度(如难以区分和确认各个缺陷分别是如何修复的)。此外，在 commit 命令中还需要指定该提交的描述消息，从而帮助开发人员更好地理解该提交的内容。一般而言，提交的描述消息需要包含四部分信息：类型、主题、主体以及链接。其中，类型描述了提交的类别，包括新功能、缺陷修复、重构、测试、文档、格式化等；主题是对提交的简要描述，一般使用祈使语气；主体是对提交的详细描述，包括该提交做了什么事情、为什么做这件事情等；链接记录了提交与其他软件制品(如开发任务、缺陷)的链接关系，便于追踪开发过程中软件制品间的链接关系(如可以追踪谁在哪些提交中完成了什么开发任务)。例如，图 3.5 是一个缺陷修复类型的提交的描述消息，该提交是对缺陷＃392 的修复。

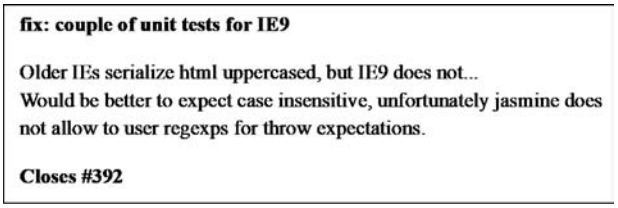

图 3.5　代码提交(Commit)的描述消息示例

3.2.3　代码分支与基线管理

一个软件项目开发团队中的开发人员往往各有分工，例如，有的开发人员在实现新功能，有的开发人员在修复缺陷，有的开发人员在发布新版本。此外，有时候开发团队可能需要在软件的主版本开发的同时，面向一些有特殊需求的客户或产品变体进行一些定制化开发，或者单独实验一些新技术。在这几种情况下，如果所有开发人员都在一个版本基础上进行开发，那么不同开发人员的提交会混杂交织在一起，导致项目难以协调和维护，版本也会变得混乱，导致项目难以进行持续集成和发布。例如，一个开发人员正在一个老的发布版本基础上修复一个重要的缺陷，并希望在修复后尽快发布新版本，而另外一些开发人员正在实现一些新特性从而导致整个软件暂时无法整体构建发布。为了解决这些问题，目前流行的分布式版本控制系统(例如 Git)都实现了代码分支管理，用于支持多个并行的互不干扰的

分支。开发人员可以在不同的分支上并行进行开发，互不干扰。这种代码分支管理功能能够更好地支持团队并行和协同开发。

软件开发实践中经常采用的分支类型包括 5 种：主分支(master)、开发分支(develop)、特性分支(feature)、发布分支(release)、补丁分支(hotfix)。主分支存储着随时可供在生产环境中部署使用的代码。当开发团队产生了一份稳定的、可供部署的代码时，主分支上的代码会被更新，并添加对应的版本号标签。由此可见，主分支主要对应版本发布。如图 3.6 所示，主分支上发布了三个版本：版本 0.1、版本 0.2 以及版本 1.0。开发分支在主分支基础上派生而来，是开发团队的主要工作分支，存储着开发团队日常开发过程中的代码提交，对应的是开发环境中的代码。开发分支上的代码提交即使有问题(例如实现不完整或包含缺陷)也不会直接影响到主分支，即不会影响到生产环境的稳定性。这也是需要区分主分支和开发分支的原因。

特性分支、发布分支和补丁分支是三种辅助分支，是用于在各种软件开发活动中解决特定问题的分支。与主分支和开发分支不同，辅助分支一般都是短期存在的。当开发团队需要开发一个新功能时，可以从开发分支派生出一个特性分支。一般而言，开发团队往往会同时开发多个新功能，这就需要从开发分支派生出多个特性分支，而开发人员就可以在相应的特性分支上进行新功能开发而互不干扰。如图 3.6 所示，开发分支上派生出了两个特性分支。当开发团队完成了新功能开发后，需要进行分支合并，即把特性分支上的代码合并到开发分支。在这个例子中，两个特性分支分别合并到了开发分支。通过分支合并，开发分支上的代码就包含特性分支中的新功能实现。

当开发分支上的代码已经基本稳定，并实现了版本发布所计划的新特性时，需要从开发分支派生出一个发布分支，为版本发布做好准备。具体而言，首先需要在发布分支上进行测试，如果发现有缺陷就需要在发布分支上进行修复，如果没有则可以准备版本发布的元数据(如版本号)。全部完成后，需要将发布分支合并到主分支，并产生一个新的发布版本。同时，也需要将发布分支合并到开发分支，从而确保开发分支上也更新了版本发布时的代码变更。如图 3.6 所示，发布分支从开发分支派生而来，最终合并到了主分支，并发布了新版本 1.0。与此同时，发布分支也合并到了开发分支。

当生产环境中的发布版本出现了缺陷，且需要马上修复时，需要从主分支派生出补丁分支，从而使得开发分支上的开发人员可以继续工作，而负责缺陷修复的开发人员可以在补丁分支上进行快速的缺陷修复而互不干扰。完成缺陷修复后，需要将补丁分支合并到主分支，并发布一个新版本，使得缺陷在新版本中得到了修复。同时，也需要将补丁分支合并到开发分支，从而保证开发分支上的缺陷也得到了修复。如图 3.6 所示，为了修复版本 0.1 中的缺陷，从主分支派生出一个补丁分支；在补丁分支上完成缺陷修复后，将补丁分支合并到了主分支并发布了新版本 0.2；同时，补丁分支也合并到了开发分支。

由以上 5 种分支的概述可知，分支合并是分支管理过程中的常见操作。然而，不同的分支可能修改了同一个文件的同一行代码，或者一个分支修改了被另一个分支删除了的代码。因此，如果在这种情况下进行分支合并，Git 就不能确定到底哪个分支的改动是正确的，即造成了合并冲突。为了便于开发人员解决合并冲突，Git 会在发生合并冲突的文件中标记出不同分支中的内容。开发人员之间需要对合并冲突进行协商，从而确定哪个分支的改动是正确的、哪个分支的改动可以被放弃掉或者是否需要组合两个分支的改动。

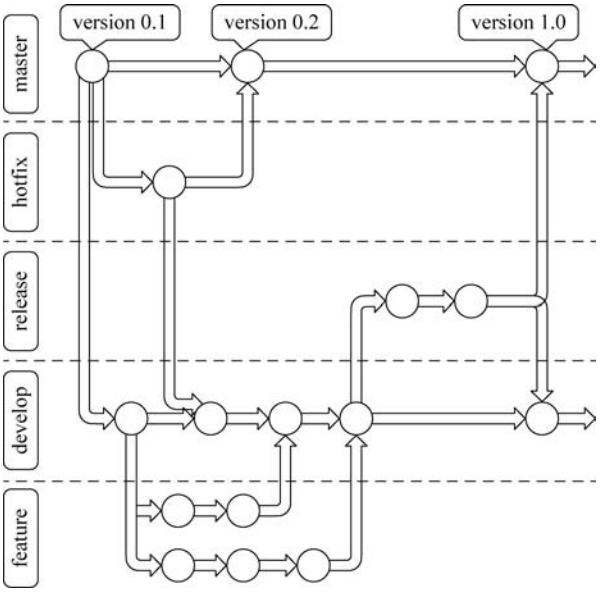

图 3.6　Git 分支管理示例

在分支管理中，一个分支被称为一条代码线(Codeline)。一条代码线的上级(即该代码线被派生出来的起源代码线)被称为它的基线(Baseline)。主线(Mainline)是没有基线的代码线。如图 3.6 所示，主分支就是主线。开发分支和补丁分支的基线都是主分支，而特性分支和发布分支的基线都是开发分支。可见，版本管理就是管理代码线、基线和主线的过程。具体而言，主线管理版本发布；代码线支持并行的独立开发，确保不同开发人员的修改不会彼此干涉；基线可以重现上级代码线对应的系统版本(如当接收到用户的缺陷报告时)，可以作为评估系统开发状态的依据(如开发进度、目前实现的需求)。

3.3　特性开发任务管理

对于一个特性开发任务，需要描述该特性的具体工作任务，指派完成特性开发任务的开发和测试负责人，同时还需要跟踪特性开发任务的开发和测试状态等。可见，软件开发过程需要管理特性的开发流程，实现特性的有效追踪，提升开发效率和软件质量。否则，软件开发过程将变得混乱，如特性的开发职责分配不清、特性的开发进度难以监控等。

3.3.1　特性描述

特性是可以给客户带来价值的产品功能。例如，校园一卡通系统中需要有食堂就餐、图书管理等多个特性。一个特性可以分解成多个用户故事，而用户故事是从用户角度对产品功能的详细描述，是更小粒度的功能。例如，图书管理可以分解成现场借阅图书、现场归还图书、在线预借图书等多个用户故事。以华为云软件开发平台 DevCloud 的特性开发任务管理流程为例，一个用户故事需要包含以下基本信息。

- 标题：对用户故事的简要描述。
- 描述：对用户故事的详细描述。

- 编号：用户故事的标识符。
- 业务编号：所属特性的标识符。
- 状态：用户故事的处理状态，一般有新建（表示用户故事刚创建）、进行中（表示用户故事已分配并在开发中）、测试中（表示在测试用户故事的实现）、已解决（表示用户故事已完成）、已关闭（表示用户故事的管理流程已结束）等。
- 模块：用户故事所属的产品模块。
- 迭代：用户故事所处的迭代。
- 特性组长：用户故事所属的特性的项目组长。
- 开发状态：用户故事所处的开发状态，一般有待启动、已启动、已完成等。
- 开发负责人：用户故事的开发负责人。
- 开发开始时间与结束时间：用户故事开发的开始时间与结束时间。
- 开发预估工作量：完成用户故事开发所预估的工作量。
- 测试状态：用户故事所处的测试状态，一般有待启动、已启动、已完成等。
- 测试负责人：用户故事的测试负责人。
- 测试开始时间与结束时间：用户故事测试的开始时间与结束时间。
- 测试预估工作量：完成用户故事测试所预估的工作量。

3.3.2 特性开发任务管理流程

在一轮迭代开始前，当由项目组长做好任务分解和下发，设置特性负责人、开发负责人和测试负责人，并设置用户故事的初始开发状态和测试状态后，特性开发任务的管理流程就开始了。由模块设计师进行核心用户估计的设计。由开发人员进行其他用户故事的设计。由测试人员进行测试方案与用例的设计。特性开发过程中提倡先设计后编码的原则。

完成如上的迭代前设计活动后，就开始了迭代活动。由模块设计师进行核心用户故事的代码编写与代码检视以及自测试。由开发人员进行其他用户故事的代码编写与代码检视以及自测试。由测试人员编写测试脚本并进行自动化测试，主要包括集成测试以及各专项测试（如压力测试和安全测试）。通过测试发现缺陷后，由模块设计师和开发人员进行缺陷的分析与修复，直到通过所有的测试。最后，由项目组长组织整个开发团队进行迭代回顾，总结与复盘特性开发任务的完成情况，包括设计、工作量、质量等问题，目的在于检视迭代中关于开发人员、过程和工具的情况、找出做得好的和潜在的需要改进的主要方面、制定在下一轮迭代中改进开发团队工作方式的计划。

需要注意的是，特性开发任务可以通过状态来自动流转到相应的负责人。例如，当开发负责人完成开发任务并将开发状态标记为已完成后，该特性将自动流转到测试负责人并通知测试负责人完成测试任务。此外，用户故事可以与完成开发任务的代码提交进行关联，从而实现代码到原始需求的反向追溯，也能实现责任到人的问题追溯。在 DevCloud 的管理流程中，基于用户故事描述所包含的信息，可以支持以下主要功能。

- **开发任务流程管理**：分配负责开发任务的测试和开发人员，制定开发任务的日程规划，监控开发任务的进度，提供开发任务完成情况的统计报表等。
- **交流与沟通**：与相关开发人员讨论、协商和评审开发任务的解决方案，把交流记录以邮件的方式通知相关开发人员，加快沟通与处理速度。

- **代码管理**：关联开发任务与代码提交，便于进行针对开发任务的代码评审与责任追溯，实现代码到原始需求的反向追溯。

3.3.3 变更管理流程

在软件开发过程中变更几乎总是无法避免的。例如，客户的需求可能会发生变化，面向市场的产品发布计划可能会因为市场或技术因素而发生变化。由于变更都会涉及一定的开发工作量，并对软件的稳定性造成影响，而且还可能涉及商业问题（例如，所提出的变更是否符合与客户签订的合同或产品的商业策略），因此开发人员不能随意地进行软件变更，以避免由此引发的技术或商业问题。为此，需要一种规范、系统和可控的方式来管理软件变更的流程，确保变更的规范性和可追踪性。

当一名利益相关者（例如产品经理、客户、开发人员）提交一个变更请求后，变更管理流程就开始了。变更请求中一般需要描述以下几个方面的信息：变更来源，即明确是谁发起了变更，如客户、产品经理等；变更原因，即明确为什么需要这次变更，如项目进度有延迟等；变更内容，即明确对什么进行变更，如版本计划变更、版本号变更、用户故事变更等。

收到变更请求后，需要由 CCB（变更控制委员会）来进行变更决策，即讨论并评审变更的合理性、变更的影响范围、变更的实现工作量等。并不是所有的变更请求都能通过变更决策。例如，客户提出的变更请求可能是由于对软件系统的误解而提出的。

当一个变更请求通过变更决策后，需要由项目组长发起一个特性开发任务，或者事务性任务来进行变更的实施与跟踪。如果一个变更请求涉及需求类任务，就发起一个特性开发任务，并按照特性开发任务的管理流程进行实施与跟踪。如果一个变更请求只是涉及辅助需求类任务的各类活动（如任务描述更改、任务暂停等），就发起一个事务性任务，包含该事务的描述、状态、责任人、预估时间、实际时间等，并进行实施与跟踪。

最后，需要进行变更归档，记录变更从请求到实施与跟踪的整个流程，便于变更的问题追踪。

3.4 缺陷修复过程管理

不管是用户发现的缺陷，还是开发人员或者测试人员发现的缺陷，都需要提交缺陷报告（有的企业称之为问题单）并进行跟踪。在问题单中，需要描述缺陷的现象、复现缺陷的步骤等，评估缺陷的优先级和重要程度，指派修复缺陷的负责人，与相关开发人员讨论缺陷的修复方案，跟踪缺陷的处理状态，关联修复缺陷的代码提交等。可见，软件开发过程需要管理缺陷修复的追踪系统，实现缺陷修复的有效追踪。否则，缺陷修复过程将变得混乱，如缺陷的职责分配不清、缺陷修复的进度无法监控等。

3.4.1 缺陷描述

缺陷是软件产品在测试和使用阶段发现的问题。例如，开发人员在测试现场借阅图书时发现借阅日期不正确的缺陷。常用的缺陷追踪系统包括 JIRA、Bugzilla 等。在不同的缺陷追踪系统中，问题单所包含的内容也不一样。

在 JIRA 中，一个问题单一般需要包含以下信息。

- **标题**：对缺陷的简要描述。
- **描述**：对缺陷的详细描述。一般需要从用户视角描述缺陷的现象或症状；需要说明错误码来辅助开发人员分析、定位和修复缺陷；需要详述发生缺陷的环境信息，是开发环境、测试环境还是生产环境；需要列举软件栈信息，包括对应的操作系统及其版本、数据库及其版本等；需要说明缺陷是否可以复现并详述复现的步骤；需要附上相关的测试脚本、补充截图、日志等信息。
- **状态**：缺陷的处理状态，一般有新提交（New，表示缺陷刚提交）、已分配（Assigned，表示缺陷已分配给相关开发人员）、未解决（Reopened，表示缺陷没有被修复而需要重新解决）、已解决（Resolved，表示缺陷已经被修复）、已验证（Verified，表示缺陷的修复已通过验证）、已关闭（Closed，表示缺陷的处理流程已结束）等。
- **优先级**：缺陷处理的优先级。优先级从低到高一般是 Trivial、Minor、Major、Critical、Blocker。可以用于在缺陷分配与处理时进行排序，更好地满足产品的开发进度。
- **处理意见**：缺陷修复的最终意见，一般有已修复（Fixed）、不是问题（Invalid）、无法修复（Wontfix）、无法重现（Worksforme）、以后版本解决（Later）等。
- **影响组件**：缺陷所影响的软件组件。
- **影响版本**：缺陷所影响的软件组件版本号。
- **修复版本**：缺陷修复所在的软件组件版本号。
- **负责人**：负责修复缺陷的开发人员。
- **评论**：关于缺陷的评论和讨论，便于分析和评审缺陷的解决方案。

图 3.7 是 JIRA 上 MapReduce 中的一个问题单示例。

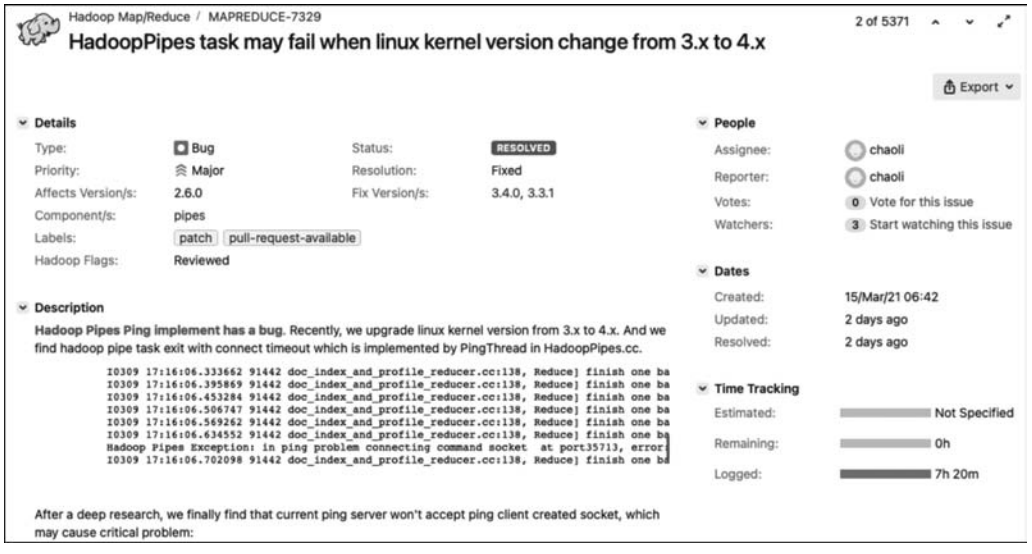

图 3.7　JIRA 中的问题单示例（MAPREDUCE-7329）

基于缺陷追踪系统中缺陷所包含的内容，缺陷追踪系统可以支持以下主要功能。

- **缺陷修复流程管理**：指定缺陷修复的优先级，分配负责修复缺陷的开发人员，监控缺陷修复的进度，提供缺陷修复的统计报表等。

- **缺陷交流与沟通**：与相关开发人员讨论、协商和评审缺陷修复的实现方案，把缺陷的讨论记录以邮件的方式通知相关开发人员，提高缺陷修复的沟通与处理效率。
- **代码管理**：通过在代码提交中引用缺陷标识符来关联缺陷与修复缺陷的代码提交，便于进行针对缺陷的代码评审与追溯。

3.4.2 缺陷修复处理流程

企业内部的缺陷处理流程如图 3.8 所示。一般由测试人员通过测试发现软件产品中的缺陷，并提交问题单。测试经理或者项目经理会对问题单进行审核与分流。对于包含错误的问题单，直接驳回给测试人员进行修改。对于重复提交的问题单，直接提交给测试人员进行关闭。对于审核通过的缺陷，由开发人员完成缺陷修复，并提交给 CCB（变更控制委员会）组长或者项目经理进行缺陷修复方案或者缺陷修复代码的审核。只有当缺陷修复方案涉及架构、流程等设计方案的改动，或者缺陷不需要修复时，才需要 CCB 进行仲裁审核。如果审核不能通过，直接驳回给开发人员进行缺陷修复的完善。缺陷修复通过审核后，由测试经理组织测试并分配测试人员进行缺陷修复有效性的验证。对于没有修改、不符合标准、缺少必需的证明材料（如修改了哪些文件的哪些代码并附上代码链接）的缺陷修复，测试经理可以驳回。最后，测试人员完成回归测试，并关闭缺陷报告。如果测试人员无法完成测试任务（如测试任务过重、生病等），可以驳回并由测试经理重新指派测试人员。

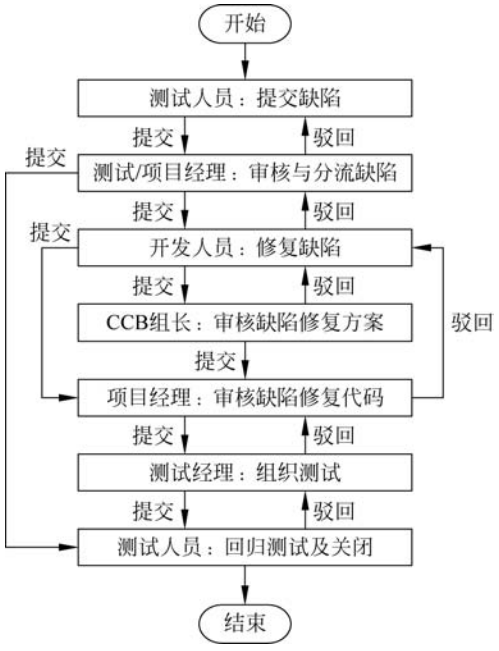

图 3.8 缺陷修复处理流程

3.5 基于追踪与回溯的工作量与质量分析

基于特性开发任务以及缺陷的各类追踪关系，可以支持多维度的工作量与质量分析。对于优先级为 Critical 及以上的问题单，需要组织相关人员进行回溯分析，寻找流程中的问题与根因并制定对策，从而改进软件开发与流程管理过程，以提升软件开发质量。

3.5.1 基于追踪的分析

首先，在单次迭代过程中，基于预计完成的用户故事数与实际完成的用户故事数的追踪关系，可以通过燃尽图分析随着时间的减少工作量的剩余情况，从而提供迭代进度的持续监控。如图 3.9 所示，迭代起点位于燃尽图的左侧最高点，发生在迭代的第 0 天；迭代终点位于最右侧，标志着迭代的最后一天。随着迭代的进行，实际剩余工作量曲线将在理想剩余工

作量曲线的上下方波动。如果实际剩余工作量曲线高于理想剩余工作量曲线，那么意味着剩下的工作量比预期多，即迭代进度落后于计划；如果实际剩余工作量曲线低于理想剩余工作量曲线，那么意味着剩余工作量少于预计，迭代进度快于既定计划。

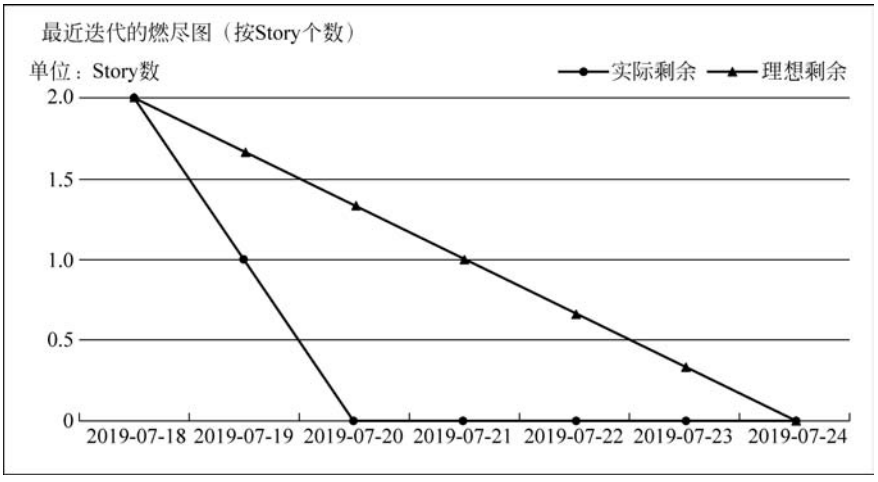

图 3.9　燃尽图示例

其次，对于一个特性开发任务与缺陷修复任务，基于预计工作量与实际工作量的追踪关系，可以实时监控特性开发任务与缺陷修复任务的处理进度，及时调整和优化开发团队内的工作分配，避免开发进度的延迟。对于开发团队成员，基于开发人员与用户故事之间的追踪关系，可以评估开发人员的用户故事完成数，衡量开发团队内开发人员的任务分配合理性，促进开发团队的工作量分配优化。

最后，基于缺陷与代码以及代码与用户故事之间的追踪关系，一方面可以快速而准确地分析每个用户故事中发现了多少个缺陷，并基于此适当调整测试预算的分配，从而提高用户故事的实现质量。一般而言，一个用户故事的缺陷数量越多，需要分配的测试预算越多。另一方面，可以评估开发人员引入缺陷的数量，评估开发人员的代码质量，促进开发人员提高其代码质量。

3.5.2　基于回溯的分析

任何问题的出现一定是流程中的某个环节出了问题。因此，对于优先级为 Critical 及以上的问题单，需要召集所有相关人员进行回溯分析，从而实现流程的优化改进。一般而言，回溯分析包括以下几个步骤。

1. 描述问题

描述所发生的问题、定位结果以及问题的后果，使参与回溯分析的相关人员更好地了解问题。例如，在 B2-2D05R 实验室进行 ABC 产品 V900R005C03B032 版本测试，测试人员在配置环境时发现“配置 MySQL 数据库服务端时没有任何提示直接退出，配置失败”，导致无法正常安装 HADR 环境，影响版本安装功能。经问题定位，发现此问题是修改问题单 WX1234579、WX1234581 时引入新问题造成的。该问题单回归不通过问题导致 V900R005C03B032 版本不能正式商用发布。

2. 成立回溯小组

回溯需要所有相关人员参加才能全面、系统地展开分析。一般而言，回溯小组需要问题单所在的特性组长担任组长，由 QA 人员担任引导员来组织大家进行回溯分析，其他小组成员还包括问题单开发责任人、问题单回归测试人、测试经理、项目经理等。

3. 分析流程

为了定位流程中的问题，必须沿着流程去找原因，必须清晰完整地展现出流程的细节。因此，回溯分析需要用简洁直观的形式表现出流程的结构概况，为后续的分析奠定基础。一种常用的流程分析方法 SIPOC 分析方法，如图 3.10 所示。其中，S 代表供应商(Supplier)，是向流程提供关键资源的组织，如某项目组。I 代表输入(Inputs)，是供应商提供的、必须满足标准的资源，如项目组修改后的脚本必须满足方案与编码规范。P 代表流程(Process)，是使输入发生变化成为输出的一组活动，如需要对修改后的脚本进行 Review(评审)、UT/ST(单元测试/系统测试)、归档、验证、确认等活动。O 代表输出(Outputs)，是流程结束后的结果即产品，需要对输出的要求予以明确，如提交回归的脚本需要通过回归测试。C 代表客户(Customer)，是接受输出的人、组织或者流程，如某测试组。基于 SIPOC 分析方法可以系统而全面地确认流程中的各个环节及其要求，并与实际流程的做法进行对比分析，从而找出哪些环节导致了问题，并识别出哪些环节存在潜在风险。例如，对于上述问题单回归不通过问题，分析出编码、Review、UT/ST、归档、验证、确认环节出现了问题。

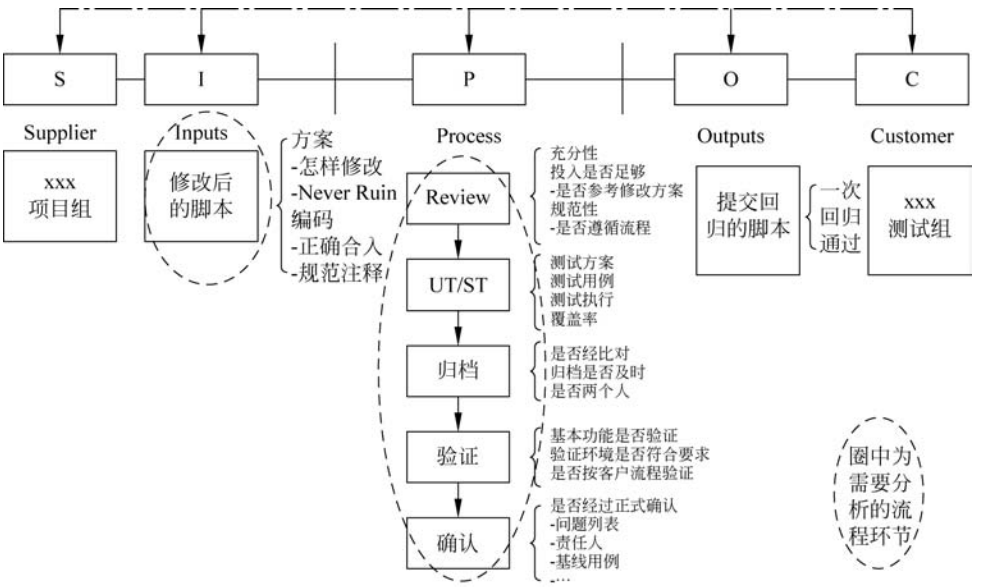

图 3.10　SIPOC 分析方法

4. 分析原因与确认要因

分析出问题出现的各个环节后，需要进一步分析导致环节出现问题的具体原因。一种常用的因果分析方法是鱼骨图分析方法，如图 3.11 所示。其基本过程如下：针对一个问题(作为鱼头，如上述的问题单回归不通过问题)，由回溯小组经过充分讨论列明产生问题的大原因(鱼骨主干，如 Review、UT/ST、归档等)，从大原因继续反复论证，列举出每个大原因

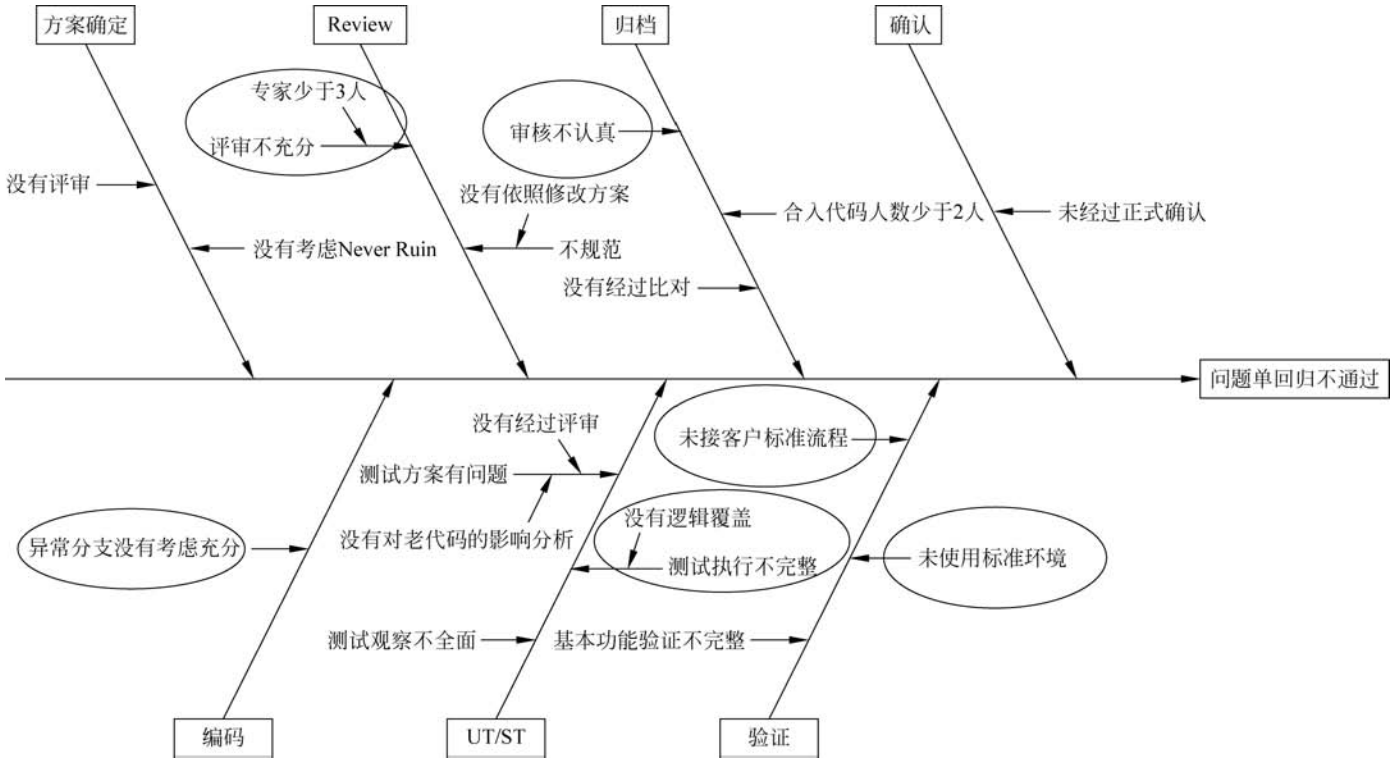

图 3.11　鱼骨图分析方法

产生的中原因，中原因再论证小原因，如此一层层论证分析下去，直到找出所有可能的原因，而不至于使导致问题的原因遗漏掉。最后，由回溯小组对鱼骨图上的原因进行充分讨论与反复论证，确认导致问题的要因，如导致上述的问题单回归不通过问题的要因有六个，即异常处理考虑不充分、ST 没有逻辑覆盖、审核不充分、测试代码未删除完整、未按照客户标准环境和流程测试、未经过正式确认。

5. 寻找根因

对于导致问题的要因，需要寻找其根本原因，从而制定相应的措施进行解决。一种常用的根因分析方法是 5why 分析方法，即对一个问题连续问 5 个“为什么”以追究其根本原因。在实际使用时，不限定只问 5 个“为什么”，主要是必须找到根本原因为止，有时可能只问 3 个“为什么”，有时也许要问 10 个“为什么”。图 3.12 是对于 ST 没有逻辑覆盖以及审核不充分这两个要因的 5why 分析过程，并最终找到了各自的根因。

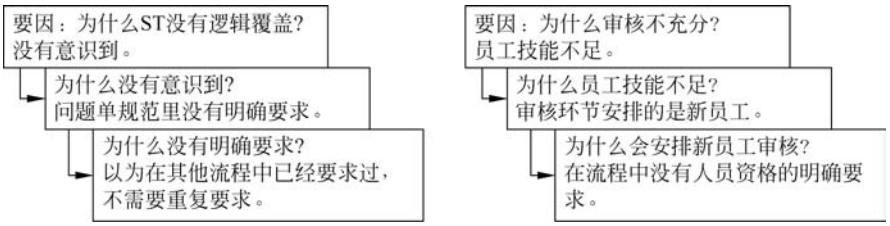

图 3.12　5why 分析方法

6. 制定对策

对于每个根因，需要制定相应的对策，指定责任人和完成时间，并落实到流程中，防止重犯同样的错误。例如，对于 ST 没有逻辑覆盖的根因（见图 3.12），相应的对策是：①在问题单流程规范中增加对 ST 逻辑覆盖和环境测试要求；②在问题单修改报告中说明 ST 逻辑覆盖情况。对于审核不充分的根因（见图 3.12），相应的对策是：①安排老员工（1 年以上）进行审核；②对新员工进行以老带新的方式进行培养；③在问题单流程规范中增加对问题单审核人员资格的要求。

小　　结

软件开发几乎总是伴随着持续的产品演化和代码修改，同时还需要支持团队中大量开发人员的并行协同开发。因此，我们需要通过规范化的版本管理来支持版本变更与开发迭代，同时对于由新特性开发和缺陷报告驱动的开发任务进行系统性的管理，从而保证软件产品质量和软件开发效率。为此，我们需要规划软件产品的迭代发布计划，并对发布版本的版本号进行规范化的命名，同时需要使用版本控制系统来管理代码版本与代码分支。此外，需要在软件开发过程中管理特性开发、缺陷修复、变更管理等任务，实现特性、缺陷、变更的有效追踪，提升开发效率和软件质量。最后，基于管理流程中的各类追踪关系，需要支持多维度的工作量与质量分析，反哺并改进流程管理中的各个环节。我们也需要对关键的问题进行回溯分析，采用 SIPOC 分析、鱼骨图分析、5why 分析等方法确认管理流程中的问题根因并制定对策，从而改进软件开发与流程管理过程。