代码管理
核心技术及实践

刘冉 肖然 覃宇 ◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书首先通过系统化的介绍和比较，从整体上讲解了代码管理工具和系统的历史和发展。其次分别从小型团队、中大型团队、分布式大团队、基于微服务的团队及开源团队的角度总结了代码管理的核心技术及实践经验，其中包括不同类型的团队对代码管理工具和系统的选择，以及代码管理的流程、策略和技巧，还有一些代码管理工具和系统的难点、痛点等，包括如何选择分支策略、如何管理多产品线的代码、代码备份策略，以及如何在大型团队中将代码从 Subversion 迁移到 Git 等。本书可帮助读者在现实中从团队的大小及代码管理模式是集中式还是分布式、开源还是闭源等各个角度去了解和思考代码管理的核心技术和实践经验，从而帮助团队建立起一套高效的代码管理系统、策略和流程。

本书的读者对象主要是每天都需要使用代码管理工具的程序员、代码管理工具和系统的管理人员，以及团队的技术领导人员。

图书在版编目（CIP）数据

代码管理核心技术及实践 / 刘冉，肖然，覃宇著. —北京：电子工业出版社，2018.1
ISBN 978-7-121-32849-7

Ⅰ. ①代… Ⅱ. ①刘… ②肖… ③覃… Ⅲ. ①软件开发 Ⅳ. ①TP311.52

中国版本图书馆 CIP 数据核字（2017）第 247838 号

策划编辑：董　英
责任编辑：徐津平
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：13.75　　字数：251 千字
版　　次：2018 年 1 月第 1 版
印　　次：2018 年 1 月第 1 次印刷
印　　数：2500 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

我从 2004 年开始直到现在都在从事软件开发工作，经历了没有代码版本管理、代码集中式管理，以及现在的分布式代码管理，在这一过程中，我深刻体会到代码管理在软件开发中的重要性。近几年，随着软件开发规模越来越大，开发团队的规模也随之扩大，出现了越来越多的分布式团队，工程效率问题也越来越突出，比如 QCon 在 2016 年首次举办了“工程效率提升”专题。由此可见工程效率已经成为现代软件业中一个无法让人忽视的问题。

在工程效率这个范畴里，代码管理占据了举足轻重的地位，因为代码是开发人员每天工作的主要对象和内容，如果不能有效地管理，必然会影响开发人员的工作效率。随着团队规模的扩大，代码管理对团队工程效率的影响也越来越大。而高效的代码管理就像一根纽带，把所有程序员有效地串联起来，让程序员可以更高效地协同开发、编写代码，完成软件的开发工作。我们在咨询工作中遇到的很多客户都对使用代码版本管理有各种问题和困惑。出于以上原因，我们觉得有必要基于经验写一本代码管理实践相关的图书。鉴于时间有限，在本书中我们只选择了自己认为核心的技术及实践。

本书首先通过系统化的介绍和比较，让读者从整体上系统地了解代码管理工具的历史和发展。然后分别从小型团队、中大型团队、分布式大团队、基于微服务的团队及开源团队的角度，总结了代码管理的核心技术及实践，其中包括不同类型的团队对代码管理工具的选择、代码管理的流程、策略和技巧，以及一些代码管理工具和系统的难点和痛点等，可帮助读者在现实中从团队规模的大小、集中式还是分布式、开源还是闭源等角度去了解和思考代码管理的实践经验。

全书共分 3 部分，其中第 1 部分主要系统化地介绍了代码管理的历史和分类，列举并简单比较了业界常用的各种代码管理工具和系统，以及迁移工具等基础知识，以帮助读者更好地选择代码管理工具。主要以集中式代码管理工具 Subversion 为主，并以一个虚拟小团队的工作流程介绍小团队的代码管理实践，最后总结了我们经历过的传统中大型团队的代码管理的核心技术及实践。第 2 部分以介绍当前流行的分布式代码工具 Git 为主，结合大型软件项目和分布式开发团队介绍了当前流行的分布式软件开发中代码管理的核心技术及实践。第 3 部分主要介绍了正在兴起的微服架构下的代码管理实践，以及一种越来越重要的软件开发模式：开源模式下的各种代码管理核心技术及实践。

阅读提示：本书不是介绍代码管理工具的专业书籍，所以不会对书中提到的代码管理工具或系统进行全面性和系统性的介绍，所以读者需要对书中提到的代码管理工具或系统全面和深入地进行学习，并阅读与其对应的专业书籍，比如 Subversion 的 *Version Control with Subversion*、Git 的 *Pro Git* 等。如果读者来自一个大型团队，则可以略过第 2 章的独立小团队的内容，在剩下的章节中找到有用的知识点。如果读者来自一个小型团队，那么可以将第 3、4、5 章作为兴趣阅读，但是在尝试里面的一些核心技术和实践之前一定要认真思考，因为它们很可能并不适应读者现在的团队环境和规模。它们更像是一把双刃剑，所以不妨将这些内容作为未来团队扩张之前的知识储备。

书中难免存在一些错误和不妥之处，敬请谅解并欢迎指出，我们将及时修改并发表在勘误中，谢谢。

刘冉

2017 年 10 月 12 号写于成都

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32849

目　　录

第 1 部分　基础与传统

第 2 部分　当前与流行

第 3 部分　发展与未来

第 1 部分

基础与传统

第 1 章

代码版本管理工具与系统

1.1 引言

自从 20 世纪中叶有了现代软件编程语言以来，对软件代码的管理一直是软件开发中的一个不大不小的问题。随着软件规模的不断扩大，开发人员也不断增加，特别是 20 世纪末互联网的爆发及 21 世纪初移动互联网的崛起，使得软件系统开发已经从相对独立的开发模型变为一个复杂的开发模型，比如需要多种语言、使用大量的第三方代码库、出现大规模的分布式团队等，导致代码管理出现了类似于 20 世纪软件危机的代码管理危机。为了解决这种代码管理危机，近几十年以来，不同的软件公司或者个人做了各种努力，开发了各种代码管理工具和系统，用于解决在不同的时代和情况下遇到的各种代码管理问题，比如著名的 Subversion 和 Git 就是为了解决不同的问题而产生的。

1.2　代码版本管理工具的历史

1.2.1　第 1 代：本地代码管理

在 20 世纪中叶，由于软件开发相对封闭和高端，只有少量大型企业、政府和学校拥有昂贵的计算机，能使用计算机进行编码的人相对较少，软件规模也不大（相对于现在而言），代码数量也不多，所以软件代码一般以一台计算机为单位，并且直接存储在计算机系统本地。由于一个软件系统的开发周期比较长，并且有可能由多人分别开发，对于代码更改的日志跟踪与回归、代码的锁定和合并都有需求，所以在那个时代就出现了 SCCS（1972 二年）、RCS（1982 二年）等基于计算机系统本地的代码管理工具。但是随着软件规模的扩大，软件开发人数逐渐增多，本地代码管理工具已经无法适应大规模的软件开发，所以出现了第 2 代基于服务器-客户端模型的代码管理工具。

1.2.2　第 2 代：中心服务器代码管理

20 世纪七八十年代，软件开发飞速发展，特别是 PC 的出现和普及，现代语言如 C（1972 二年）、SQL（1978 二年）、C++（1983 二年）、Lisp（1984 二年）、Erlang（1986 二年）等的涌现，以及信息化革命中对大量信息系统软件的需求，推动了大量的商业软件公司和开源软件组织的出现，使得软件开发成为一个规模化的社会活动。而规模化给软件开发带来了诸多问题，其中一个就是如何在规模化的情况下管理软件代码。为了解决这个问题，一些开源组织和商业软件公司开发了第 2 代代码管理工具和系统，比如 CVS（1986 二年）、ClearCase（1992 二年）、Visual SourceSafe（1994 二年）、Perforce（1995 二年）、Subversion（2000 二年）等。

这一代代码管理工具和系统的主要特点是基于中心服务器端和客户端，软件开发者通过客户端从服务器端获取代码，并将自己的代码通过客户端提交到服务器端进行存储，代码服务器端会记录所有的代码提交日志并供开发者获取和查看。多个开发者可以同时获取同一个服务器上的代码，并向这个服务器提交代码，如果遇到冲突，就需要自己修复冲突

后再进行提交。

虽然这一代中心服务器端代码管理工具有以上优点，并且解决了规模化软件开发带来的一些问题，但是仍然存在不少限制，如下所述。

◎ 在无法连接服务器的情况下，无法查看以前提交的日志和比较代码版本（在网络十分缓慢的情况下工作也十分困难），很多开发者无法完成正常的代码开发工作。

◎ 不支持本地分支策略，导致开发分支创建缓慢，分支管理困难，并且一旦创建就很难修改，不适用于快速迭代开发流程。

◎ 由于一般只有一个中心服务器端，一旦发生灾难性问题，所有日志都会丢失，所以需要经常备份（有些大型公司在有条件的情况下自己实现了高可用的代码服务器来做灾备）。

◎ 如果软件代码量过于庞大，则可能出现速度缓慢的情况，因为每次查询、比较和提交等操作都需要和服务器通信，会对代码服务器造成很大的负载。

1.2.3 第 3 代：分布式代码管理

20 世纪 90 年代和 21 世纪初，随着大规模开源软件系统、互联网及智能移动开发的出现，软件开发进入了一个新纪元，其特点如下。

◎ 规模越来越大，代码量与代码提交量巨大。

◎ 开发人员越来越多，可能分布在不同的城市，甚至不同的国家。

◎ 组成部分越来越复杂，软件系统开发会使用多种不同的语言、技术及开发模型。

◎ 随着软件开发方法的普及（比如 Scrum、XP、Lean 等），人们需要对软件代码版本和分支更加灵活与方便地进行管理。

为了解决这些问题，出现了第 3 代分布式代码管理工具，比如 Git（2005 年）和 Mercurial（2005 年）。第 3 代代码管理工具结合了第 1 代和第 2 代的优点并实现了分布式的代码版本

管理，其特点如下。

◎　在没有和服务器连接的情况下仍然可以查看日志、提交代码和创建分支。

◎　支持本地代码分支，可以快速方便地实现各种分支管理。

◎　对代码可以进行分布式管理，从而可以实现分代码库管理的负载分流。

当然，当前这些工具还存在一些局限，相对于之前的代码管理工具学习曲线较高，管理理念与前两代区别较大，其中包括如下内容。

◎　代码基于分布式的方式进行管理。

◎　灵活的分支策略，包括轻量的本地分支与远端分支。

◎　灵活的代码版本管理。

◎　较复杂的代码同步策略。

1.3　常用的代码管理工具

代码管理工具是代码管理的基础，不同的管理工具有不同的特点、局限、成本及理念，而且对于一个团队或者公司来讲，一般选择了一种代码管理工具就意味着已经选择了这种代码管理的理念和体系，并且使用的时间越久就越难以更换，所以需要充分了解不同的工具，从而能够选择适合自己团队或者公司的代码管理工具。

1.3.1　Perforce

Perforce 是 20 世纪相关行业中认可度最高的第 2 代商用代码管理工具，其主要客户包括 Google、微软等大型软件公司（虽然 Google 和微软在代码量超过 Perforce 的管理极限时自己重新开发了专属的代码管理系统，但是也深受 Perforce 的影响，直到现在还有 Google 员工宣称 Google 使用和 Perforce 一样的单一中心代码库）。但是 Perforce 价格昂贵，中小

型公司、开源社区和开源项目无法承受，所以 Apache 基金会在 21 世纪初推出了开源、免费的 Subversion 来为这些用户提供服务。

1.3.2 Subversion

Subversion 由 Apache 基金会开发和支持，因为 Subversion 有着开源、免费、稳定及简单实用等特性，所以迅速替代 CVS 成为第 2 代代码管理工具中的开源主力产品。众多开源软件在 21 世纪前 10 年大规模使用了 Subversion，这推动了 Subversion 的高速发展和逐步成熟，也使得很多商业公司和商业软件系统从一些商业代码管理工具（比如 ClearCase、VSS、Perforce）转向了 Subversion。由此催生了一些商业版本的 Subversion 服务器系统，比如 Virtual SVN、CollabNet Subversion Edge 等。但是 Subversion 并不适合超大型跨团队、跨地区的开源项目，所以第 3 代代码管理工具 Git 诞生了。

1.3.3 Git

Git是Linus Torvalds为了替代BitKeeper来管理Linux系统源代码而专门开发的一款开源、免费的分布式代码管理工具。当Git在Linux上获得成功之后，开源社区转而使用Git。Linus也开始对外宣传Git，包括到Google宣讲Git[1]，最终Google针对Git开发了自己的Gerrit代码管理系统来管理自己的Android系统代码。在开源社区，GitHub的出现也直接导致了Google Code、Sourceforge等云代码管理系统的没落。Git是分布式代码管理工具中最为成功的典范，它开源、免费、分布式、轻分支、功能强大，受到开源软件开发人员的欢迎，但是其学习曲线高的缺点也让不少开发者望而却步。由于Git在开源领域及大型开源系统中的成功，现在越来越多的大型商业公司（比如大型电信开发商、金融IT部门等）也开始从Subversion、ClearCase等传统的第 2 代代码管理工具转向第 3 代分布式代码管理工具Git。Git并不是唯一的分布式代码管理系统，其他分布式代码管理系统还包括Mercurial、Monotone、TFS等。

1 https://www.youtube.com/watch?v=idLyobOhtO4

1.3.4　Mercurial

Mercurial作为与Git类似的分布式代码管理工具，虽然社区、核心功能、用户群等都不如Git，却拥有一群忠实的用户，因为它也有一些与众不同的特点，比如基于Python开发、实现代码更为简洁等。Facebook就是Mercurial的用户之一，它为了提高Mercurial的性能以支持其超大规模代码库，开发了两个Mercurial的扩展，即hgwatchman和remotefilelog[1]。

1.3.5　Microsoft GVFS （Git Virtual File System）

Microsoft GVFS[2]是微软最近公布的一个开源项目，仍在开发过程中。它是微软为了解决使用Git管理一个独立的大型代码库所存在的问题而基于Git开发的一个项目。其解决的问题包括clone需要同步整个代码库，无法只同步或者更新一个库中的部分文件等。使用GVFS可以管理一个独立库中所需要的部分代码，从而可以管理一个超大型的独立代码库。微软已经开始使用GVFS作为正式产品项目的代码库管理工具，其中包括Windows系统，然后将其开源到GitHub上并希望获得社区的帮助来共同开发这个Git的扩展系统。微软还发布了支持GVFS的定制Git[3]客户端来支持GVFS，而且当前除了Visual Studio Team Services支持GVFS，Tower和GitKracked等第 3 方代码管理系统也宣称支持GVFS。

常用的代码管理工具对比如表 1-1 所示。

表 1-1　常用的代码管理工具对比

	版权与价格	架构与模型	支持的系统	优势/特点	限制/缺点	主要的用户群及产品
Subversion	免费	集中式	Linux、Windows、Mac	简单、易用	本地代码的每个目录下面都会多出一个目录与一些用户版本控制的文件	各大开源社区、各种传统的开源产品、中小软件公司

1　https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/

2　https://github.com/Microsoft/gvfs

3　https://github.com/Microsoft/git/releases

续表

	版权与价格	架构与模型	支持的系统	优势/特点	限制/缺点	主要的用户群及产品
Perforce	免费/收费	集中式	Linux、Windows、Mac	简单、易用，本地代码库中没有多余、无关的文件与目录	昂贵	各大主流的商业公司，比如微软、Google 等
Git	免费	分布式	Linux、Windows、Mac	支持分布式； 支持快速本地分支； 支持本地离线管理代码，比如提交、查看日志； 能删除分支	学习曲线较高， 初学者容易出错	各大开源社区、各种新开源软件
Mercurial	免费	分布式	Linux、Windows、Mac	隐藏分支，简单易用	社区规模不大； 用户规模较小； 对非英文字符支持有点问题； 不能删除分支； 核心功能较少，需要通过扩展才能获得更多的功能	某些开源社区和中小软件公司
GVFS	免费	分布式	Windows	支持单一超大型代码库	微软开发并发布的，需要特定的 Git 服务器支持，现在仍在开发过程中	微软公司和一些开源社区中的项目

1.4 常用的代码管理系统

在一个项目选择好一种代码管理工具后，其代码管理理念和体系就已基本定型，然后需要选择一个代码管理系统用于实施。对于某些开源工具，比如 Subversion 和 Git 等，在一般情况下都不会在真实的项目中直接使用这些工具提供的原生命令，比如直接使用

svnserve -d 和 git daemon，而是需要选用一种集成了其他功能的管理系统，比如权限管理、Web 管理、安全协议、备份系统等。因此充分了解并选择一个代码管理系统才能真正有效地实施代码管理的策略和理念，帮助团队或公司有效地管理代码。

1.4.1　Virtual SVN Server

由于Windows用户众多且商业公司对商业SVN Server的需求，Visual SVN Limited开发了商用的Virtual SVN Server。Virtual SVN Server主要针对使用Windows的商业用户，提供ActiveDirectory统一登录、多站点代码库复制、Web管理界面、登录和操作日志系统等功能。Virtual SVN Server有标准版和企业版，其中标准版是可以免费使用的，比企业版少了很多高级功能 [1]。技术能力较薄弱的中小企业可以选择Virtual SVN Server标准版，大型企业（代码量大、人员众多）则需要选择企业版，且技术能力很强的企业可以基于Linux等免费系统搭建免费的Subversion服务器。

1.4.2　GitLab Server

GitLab是一个类似于GitHub的Git系统，是一个包括权限认证、代码审查、问题跟踪及wikis等的一个综合代码管理系统，而且提供了和GitHub类似的云代码管理服务。最重要的是它还提供了GitLab Server下载，可以进行私有部署和使用。GitLab Server与Virtual SVN Server类似，一共提供了两个版本：社区开源免费版和企业收费版 [2]。对一般的中小型企业而言，社区开源版的功能足以满足其日常工作需求，并且社区开源版支持单机 25000 个用户及高可用集群。但是对于中大型商业、企业而言，还是应该选择其企业版来搭建自己的内部私有Git代码服务器。

1　https://www.visualsvn.com/server/licensing/

2　https://about.gitlab.com/features/#compare

1.4.3 Gerrit Server

Gerrit 是 Google 为了管理和维护其 Android 项目源代码而开发的基于 Git 的源代码管理系统，有代码管理、权限管理、Web 管理界面和代码审查等功能。但是 Gerrit 与 GitLab 的设计理念是不同的，所以适用人群也与 GitLab/GitHub 不一样。Google 设计 Gerrit 时延续了传统的基于中心端代码服务器来管理代码的理念，开发人员只需专注于开发代码并完成代码提交（git commit）和代码推送（git push），Gerrit 就会自动生成代码审查请求，最后在代码通过审查以后由审查人员或者管理人员将代码合并到主干分支中。这与 GitLab/GitHub 等需要在代码提交后手动选择代码提交来进行 Merge Request 及 Pull Request 完全不一样。Gerrit 之所以被这样设计，主要是为了最大限度地降低分布式代码管理系统的使用难度，屏蔽一些手动操作，让程序员尽量少改以前的代码提交和管理流程。由于 Gerrit 的这种机制，程序员对每次代码提交都十分重视，而不是随意提交，不像 GitLab/GitHub 那样可以随意定制不同数量的代码提交来生成一个 Merge Request 及 Pull Request。所以，在代码提交和审查方面，Gerrit 不如 GitLab、GitHub 灵活多变，它是固定不变的，导致只有大规模或者十分关注每一次代码提交的软件项目才会考虑使用它，比如 Android、OpenStack 等，而一般的开源或者中小型项目还是会选择 GitLab、GitHub。

常用的代码管理系统对比如表 1-2 所示。

表 1-2　常用的代码管理系统对比

	版权与价格	支持系统	优势/特点	主要用户群/产品	主要开发语言
VirtualSVN Server	免费/收费	Windows	简单、易用； 很好地支持 Windows； 商用支持	各商业软件开发公司或团体、商业软件系统	C
GitLab Server	免费/收费	Linux、Windows、Mac	权限管理简单； 类似于 GitHub； 支持 Merge Request； 集成缺陷跟踪	各开源社区、各种传统的开源产品、中小软件公司	Ruby

续表

	版权与价格	支持系统	优势/特点	主要用户群/产品	主要开发语言
Gerrit Server	免费/收费	Linux、Windows、Mac	权限管理灵活；支持基于代码提交的 Push Code Review	各大开源社区与大型软件系统、大型商业软件公司	Java

1.5　从 Subversion 迁移到 Git 的常用工具和方法

Subversion 是 Git 出现前在开源社区及中小型软件公司中使用最为广泛的代码管理工具，但在 Git 特别是 GitHub 等这样的代码管理系统出现后，Git 一举成为了开源社区的新宠，并在开源界超过 Subversion，成为排名第一的代码管理工具。而且随着 GitLab 和 Gerrit 的出现，Git 也逐渐被商业公司重视，并逐步蚕食商用软件代码管理工具的市场份额。所以当前有大量的软件系统、开源社区和商业公司都需要使用 Git 替换以前所使用的代码工具，比如 Subversion、Perforce 等。但是很多大型系统往往代码量巨大、开发人数和团队众多、权限管理复杂、开发时间很久，导致拥有大量的日志，所以很难迅速切换到 Git 上，而是需要逐步地从老的代码管理系统迁移到 Git。对于从 Subversion 到 Git 的迁移，现在已经出现了一些开源、商用的工具来辅助，比如 SubGit 和 git-svn。

1.5.1　SubGit

SubGit 是一个可以将 Subversion 平滑迁移到 Git 的工具，它通过对本地和远端的 Subversion 代码库创建镜像库，让用户可以根据需要来选择是使用 Subversion 还是 Git，并且支持一次性地将 Subversion 导入 Git 库中或者集成 Atlassian Bitbucket Server 来使用。

对于 SubGit 的镜像功能，它提供了一个双向通道，可以在不关闭 Subversion 代码服务器的情况下完成迁移工作。

通过 SubGit 的导入功能可以快速地将大型的 Subversion 代码库导入 Git 库，并且会保留所有的日志信息。SubGit 提供的命令行工具可以完成无人值守的自动化迁移。SubGit 对于开源项目是免费的，但是对于商业项目最多免费支持 10 个终端用户。

1.5.2 git-svn

git-svn 是 Git 官方提供的一个让代码库在 Subversion 与 Git 之间进行转换的简单工具。它提供的是双向管道，既可以把已有的 Subversion 库转换为 Git 库，也可以把一个 Git 库转换为一个 Subversion 库。但其默认用法是将一个远端的 Subversion 库动态地转换为一个本地 Git 库，本地 Git 库使用 git svn fetch 命令时可以从远端的 Subversion 库同步代码，而当使用 git svn dcommit 命令时，就可以将本地的 Git 库中的代码同步到 Subversion 库中，而且它支持标准的 Subversion 库结构，比如 trunk/branches/tags 等。

1.5.3 手动

不使用任何第三方工具来迁移是最快的一种迁移方法，但其最大的缺点就是无法保留 Subversion 代码库以前的日志，所以只适用于小型团队或者不需要保留以前提交的日志的团队。其步骤也很简单：首先使用 svn export 命令将 Subversion 代码库中的代码导出，然后将代码导入本地 Git 代码库中，接着使用 Git 命令添加、提交和上传到代码库就可以了。

从 Subversion 到 Git 的迁移对比如表 1-3 所示。

表 1-3　从 Subversion 到 Git 的迁移对比

	版权与价格	架构与模型	支持系统	优势/特点	限制/缺点	主要用户群
SubGit	收费	基于代理服务器动态转换	多种系统	商用支持； 支持代码镜像； 支持大规模代码库导入	商业收费，对于中心端配置管理者有一定的学习曲线	商业用户

续表

	版权与价格	架构与模型	支持系统	优势/特点	限制/缺点	主要用户群
git-svn	免费	基于本地代码仓库动态转换	多种系统	Git工具库默认支持； 简单、免费	对于每个开发人员有一定的学习曲线	大部分普通用户
手动	免费	手工转换	所有系统	可以随意定制	流程较复杂，容易出错	中小规模用户

关于SubGit与Git-Svn的更多对比请参见SubGit官方文档 [1]。

1.6　常用云端代码管理系统

随着互联网的普及和开源软件的发展，基于云管理代码已经成为开源软件社区和开源项目的主流方式，比如 Sourceforge、Google Code、GitHub 等，因为它有着免费、随时随地可以访问等优势，并且有一些专门支持这些云代码管理系统的第三方扩展系统，比如 Travis-CI 是一个可以很容易地从 GitHub 上同步代码并进行测试、部署的持续集成系统。这些第三方系统让云端管理系统变得越来越强大。并且随着云代码管理系统开始提供收费服务和私有仓库，它也在逐步吸引一些商业软件公司的使用。当然出于对安全性、健壮性等方面的顾虑，商业用户群还不是十分庞大。随着云代码管理系统的不断发展，特别是在安全性、健壮性方面的增强，会有越来越多的商业用户使用云端代码管理系统。

1.6.1　Sourceforge 和 Google Code

Sourceforge 和 Google Code 都是最为成功的第 1 代支持 Subversion 的云代码管理系统。其中 Sourceforge 是历史最为悠久的 Subversion 云代码系统，曾经一度成为最大的开源社区

1　http://www.subgit.com/documentation/gitsvn.html

（现在已经被 GitHub 取代），很多优秀的开源项目都被放在上面，比如 notepad++、FileZilla、eMule、Apache OpenOffice 等。但是由于 Sourceforge 使用起来比较烦琐，所以后来 Google 推出了 Google Code，简单易用的特性及 Google 的强大技术后盾，使其很快成为开源界的新宠。不过好景不长，随着 GitHub 的出现，以及大量项目从 Google Code 迁移到 GitHub，Google Code 已于 2016 年正式关闭。若你仍在使用 Sourceforge 则不用担心，因为 Sourceforge 现在依然对外提供服务。

1.6.2 GitHub

GitHub 是一款基于 Git 和 Ruby On Rails 开发的第 2 代云代码管理服务系统，一经发布就受到开源界的喜爱和追捧，到 2016 年已经成为世界排名第一的云代码管理服务系统和开源社区。GitHub 提供了丰富的代码管理功能，比如代码托管、问题跟踪、代码协作评审、团队开发管理、Markdown 页面编辑器等。用户可以免费在上面创建自己的公开代码库，创建私有代码库时则需要付费。

由于GitHub创建得比较早（2008 年），并且提供免费及易用的Git云代码管理服务，所以拥有了许多世界级的开源软件，比如Node.js、Docker、Ruby On Rails、jQuery、Bootstrap、PowerShell等，并且不少大型软件公司在GitHub上创建了专有主页用于发布自己的开源软件，比如Microsoft[1]、Google[2]、Facebook[3]、Twitter[4]、Netflix[5]、阿里巴巴[6]、淘宝[7]等。

1.6.3 GitLab 和 Bitbucket

GitLab 除了提供了开源、免费的 GitLab Server，还提供了和 GitHub 类似的云代码管

1 https://github.com/microsoft
2 https://github.com/google
3 https://github.com/facebook
4 https://github.com/twitter
5 https://github.com/netflix
6 https://github.com/alibaba
7 https://github.com/taobao

理服务系统。它将协作聊天、问题跟踪、代码审查、持续集成（CI）和持续交付（CD）集成到一个统一界面以方便用户使用，并且提出了从“想法”到“产品”的完整开发流程链条（IDEA>>ISSUE>>PLAN>>CODE>>COMMIT>>TEST>>REVIEW>>STAGING>>PRODUCTION>>FEEDBACK），从而尽可能地帮助用户完成更多的工作，所以它十分适合各种中小型开源或者创业软件团队采用。

而 Bitbucket 是 Atlassian 公司推出的一款服务于专业团队的云代码管理服务系统，它延续了 Atlassina 的主打产品 JIRA、Confluence 等的专业风格，并且能与 JIRA、HipChat、Bamboo 进行集成。所以它应该是正在使用 JIRA、Bamboo 等产品的团队和公司的首选。

最后，GitLab 和 Bitbucket 都提供了免费的私有代码仓库服务，而 GitHub 则未提供。

1.6.4　Coding、码云、阿里云 Code

国内用户如果使用国外的 GitHub、GitLab 和 Bitbucket 等，则存在速度慢、连接不稳定、不支持中文等各种问题，因此选择国内的云代码管理服务系统也是很明智的，比如 Coding 和码云。

Coding 除了提供了类似于 GitHub 的基于 Git 的免费和收费的云代码管理服务，还提供了项目管理、WebIDE 及移动客户端。码云则是基于 GitLab 开源软件开发的，并且在 GitLab 的基础上做了大量的改进和定制开发，目前已经成为国内最大的代码托管系统，因为它允许用户免费创建公有和私有的云端代码库。

云端代码管理系统的对比如表 1-4 所示。

表 1-4　云端代码管理系统的对比

	版权与价格	优势/特点	限制/缺点	主要用户群
Sourceforge	免费	历史悠久； 传统的经典开源项目很多	只支持 Subversion	一些在使用 Subversion 开源社区和开源项目的用户
GitHub	免费/收费	当前最大的免费云代码管理平台，使用人数多，支持良好	服务器在国外，只支持 Git	世界上大部分开源社区、开源项目及很多商业用户

续表

	版权与价格	优势/特点	限制/缺点	主要用户群
GitLab	免费/收费	开源、免费，集成了各种开发流程管理的统一平台，并且可以私网部署	其云服务器在国外，只支持 Git	大部分中小企业，其需求是成本低和可以私网部署
Bitbucket	免费/收费	同时支持 Git 和 Mercurial，支持免费的私有代码库	服务器在国外	一些商业用户
Coding	免费/收费	服务器在国内，支持项目管理和 WebIDE	服务器在国内	国内的免费和商业用户
码云	免费/收费	服务器在国内，免费创建私有代码库，支持项目管理	服务器在国内	国内的免费和商业用户
阿里云 Code	免费/收费	服务器在国内，现在支持免费创建私有代码库，以后可能需要收费，支持项目管理	正在公测	国内的免费和商业用户

第 2 章

独立小型团队

2.1 启程：团队与项目

前面介绍的这些代码管理工具和系统只是业界众多工具和系统中的部分典型和流行的工具和系统，不同类型的项目和团队在选择、使用代码管理工具和系统时仍然存在很多困惑和问题。本章将通过一个小型团队案例来展示小型团队应该如何选择代码管理工具，在使用这些工具的过程中遇到的问题及解决方法，借此来帮助大家解决自己的困惑和问题。

本案例的主角是一个创业型的独立小型团队。某大型公司成立一个了小型的内部创业团队，团队一共有 7 个人，其中开发人员有 4 个人，其他角色有 3 个人，他们主要负责开发一个创新的基于 J2ME 的嵌入式系统。由于是创业型项目，公司的策略是低投入、低成本、快速试错，所以整个项目都尽可能选用开源、免费和新型的管理工具和技术栈等。由于公司层面上使用的是 ClearCase，使用和管理成本较高，所以团队选择了免费、简单、易用的 Apache Subversion。

2.2 痛点与需求

团队成员由于第 1 次使用 Subversion，所以遇到了不少问题。

◎ 如何选择和搭建 Subversion Server？

◎ 日常工作模式是什么？

◎ 分支策略是什么？

◎ 如何备份及备份策略是什么？

2.2.1 如何选择和搭建 Subversion Server

由于低成本的要求，团队需要选择低成本的 Subversion Server，需要有完善的搭建文档和支持版本的平台。根据这个要求，这个团队并没有选择商用的 Subversion 服务器，而是使用原生的 Subversion 来直接搭建代码服务器，随着时间的演进产生了 3 种方案。

方案 1：基于 Windows 共享目录搭建代码服务器

一开始，项目组全部使用 Windows 系统。为了快速搭建服务器，选择了基于 Windows 共享目录的方式来搭建代码服务器。

首先选择了一台 Windows 服务器，共享 D 盘的共享目录，并在 D 盘中创建一个目录 D:\repos 用于存储服务器端的代码库，并通过 svn 命令创建真实的代码库目录。

实例代码：

```
$ mkdir D:\repos
$ svnadmin create D:\repos\j2me
$ svn mkdir -m "Initial sure repos" "file:///D:/repos/j2me/trunk"
"file:///D:/repos/j2me/branches" "file:///D:/repos/j2me/tags"
```

注意：如果有时间和能力来配置服务器，则还可以配置 Subversion 和 HTTP 服务器，

然后使用对应的地址进行访问，比如 https://j2me/trunk、https://j2me/branches 和 https://j2me/tags。

Subversion 支持的协议如下。

◎　file:///

◎　http://

◎　https://

◎　svn://

◎　svn+ssh://

然后在每个开发人员的开发机上，将这台服务器中共享的 D 盘映射到本地的 S 盘，开发人员就可以通过 svn 命令将代码同步到本机来进行开发工作了。

实例代码：

```
$ cd j2me-local
$ svn checkout "file:///S:/repos/j2me/trunk" ./
```

方案 2：基于 Subversion 原生命令搭建代码服务器

随着项目的进行，项目组稍微有了一些空闲时间，于是决定搭建基于 Subversion 和 SSH 协议的代码服务器。在 Subversion 官方软件包里有一个命令是 svnserve，使用它搭建的代码服务器可以支持 svn://或者 svn+ssh://格式的 URLs，这样开发人员使用的客户机就不需要再映射服务器的共享目录了。项目组决定继续使用 Windows 作为代码服务器，所以将 svnserve 作为 Windows Service 来搭建。

实例代码：

```
$ sc create svn
       binpath= "C:\subversion\bin\svnserve.exe --service -r D:/repos/"
       displayname= "Subversion Server"
```

```
depend= Tcpip
start= auto
```

方案 3：基于 Apache Http Server 搭建代码服务器

随着项目规模变大，人数增多，为了代码的安全，该团队决定加入简单的权限管理，并且使用更为方便的 HTTP 作为 Subversion 服务器的传输协议。

首先需要安装一个Apache HTTP 2.0 或者以上版本的Web服务器，然后安装mod_dav和mod_dav_svn，最后配置httpd.conf就可以了。其中mod_dav提供了简单的权限认证系统，而mod_dav_svn提供了连接Subversion代码库的能力。如果团队有技术能力重新编译Apache HTTP，则还可以将mod_dav和mod_dav_svn预编译到httpd中，这样使用起来更为简单。Subversion的官方文档详细描述了Apache HTTP Server的搭建步骤 [1]。

技巧：如果使用 Linux 系统来搭建以上三种方案，则可以使用 NFS 来代替 Share Folder，用 Daemon 来替代 Windows Service，其中某些参数稍有区别。

可以使用Linux常见的发行版如Debian[2]、Ubuntu[3]和RedHat[4]来创建Subversion服务器。

2.2.2 定制代码库结构

由于该团队属于小型团队，人数不多，项目单一，代码量也不大，所以选择使用标准的 Subversion 代码结构，其中包括项目主目录及其下的主代码目录（trunk）、分支代码目录（branches）和标签代码目录（tags）。

◎ 项目主目录

主目录一般使用项目名称，这样方便记忆及区别于其他项目，而且代码目录具有一定

1 http://svnbook.red-bean.com/en/1.7/svn.serverconfig.httpd.html

2 https://wiki.debian.org/Subversion

3 https://help.ubuntu.com/community/Subversion

4 https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Developer_Guide/ collaborating.svn.html

的扩展性。

◎　主代码目录（trunk）

Subversion 官方使用 trunk 这个名字来表示主代码目录。这个目录对于一个项目有且只有一个，是项目的主干代码。由于小型团队开发的软件系统一般都不是很复杂，所以开发人员应该保证 trunk 中有最新的可用代码，并且对于新开发的功能代码的集成和测试也应该在这个主干代码上进行，最好不要使用分支（branches）来对代码进行管理，尽量保证代码管理的简单性。但是随着项目越来越复杂，开发的功能越来越多，创建分支并进行分支管理是无法避免的。

◎　分支代码目录（branches）

Subversion 官方使用 branches 来表示分支代码目录。这个目录主要用来存储分支代码库，而分支的用途一般是开发相对独立的功能。对于一般的小型团队来说，不需要创建和使用分支，所以可以只创建一个空目录以备以后使用。

◎　标签代码目录（tags）

Subversion 官方使用 tags 来表示标签代码目录。这个目录主要用于存储有特定版本的代码库，比如稳定的发布版本、特定的测试版本或者特定的备份版本等。tags 目录是一个标准的目录，小团队一般可以用来做发布版本的备份，或用于管理发布版本及 Hotfix 等紧急修复。

```
/
|
|
->j2me
     |
     |
     ->trunk （用于主干代码开发）
     ->branches （用于开发时间较长且相对独立的功能开发）
     ->tags （用于发布版本的备份等）
          |
```

```
        |
        ->release1.0
        ->release2.0
```

2.2.3 分支策略

分支策略是代码管理中最为重要的一个环节，它的主要用途是帮助开发人员对代码进行协作开发，所以它的模型和用法也是多种多样的，如果使用不合理，则将会使代码开发协作变得混乱。小型团队应尽量少或者不使用分支，避免带来过高的复杂度。但是对于中大型或者分布式团队来说，对分支的使用是无法避免的，我们在后面几章中将对其进行详细阐述。

不同的代码管理工具实现分支的方法是不同的，一般分为三类：全复制、轻量复制、内部引用。

◎ 全复制：指将整个代码库复制到另外一个目录下，其特点是创建非常慢，切换分支时需要改变本地的工程目录，更换时间也比较长，比如 CVS。

◎ 轻量复制：指将整个代码库的目录复制到另一个目录下，但是不复制任何文件，只是创建文件的链接，其特点是创建比较慢，切换分支时需要改变本地的工程目录，切换相对较快，比如 Subversion。

◎ 内部引用：指使用整个代码库的内部引用的方式创建分支，其特点是创建速度非常快，切换分支时不需要改变本地的工程目录，比如 Git。

当前软件行业中对于分支的理解一般指第 3 种，而第 1 种和第 2 种只是传统的分支概念，利用路径的不同来区分，本身就是一种不合理的分支方式。第 3 种才是真正意义上的合理分支方式，因为它解决了第 1 种和第 2 种中存在的很多问题，比如分支创建慢、空间占用大等。

2.2.3.1　Subversion 分支的本质

首先Subversion没有内部分支的概念，其本质是文件和目录的复制，所以分支一般都是trunk的复制，区别是分支和trunk存在于不同的目录中。Subversion为了尽量减少分支创建的时间和存储空间，使用了一种轻量复制（Cheap Copies）的技术，采用了类UNIX的hard link方式来创建分支代码库里的文件，然后在代码文件被修改后，Subversion才会真正地创建新的代码文件。这仅仅针对服务器端的代码库，当将代码同步到本地工作目录时，分支代码库将被创建成为独立的代码文件，所以如果分支过多，就会导致本地的工作目录十分庞大。这种方式也仅局限于使用Subversion命令的方式创建的分支，通过手动创建时则不适用。Subversion被官方称为“Bubble-Up Method”[1]。

所以在 Subversion 中创建分支的方法有两种：使用 Subversion 的命令创建分支（svn copy *URL1 URL2*，默认）；手动复制需要创建分支的代码库（不建议使用）。

2.2.3.2　经典分支

对于不同的项目，其分支策略可能会有所不同，但是为了减少复杂度，如果要使用分支来协作开发，则应该选用经典的分支策略，如图 2-1 所示。

在需要做一个较大的功能时，一般的做法是开发人员在本地持续开发几天甚至几周，并且在此期间不提交代码，因为开发人员不想把一个半成品提交上去，在代码开发完毕后才提交。这样的做法存在几个问题：首先，这样做不安全，有可能在整个编辑过程中丢失代码；其次，不够灵活，无法进行部分提交；最后，如果一次性提交所有代码遇到冲突，则将非常难以合并。

所以更好的解决方案是开发人员创建自己或者某个功能的分支，这样就可以在不影响其他人的情况下提交并保存没有完成的代码，而且可以很容易地共享代码并和其他开发人员协作。

1　https://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#server.fs.struct.bubble-up

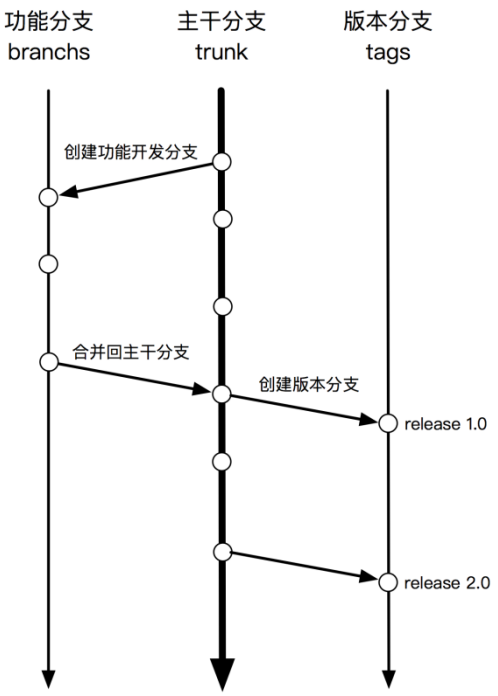

图 2-1　Subversion 经典的分支策略图

2.2.4　日常工作模式

由于团队规模还比较小，所以日常工作的模式需要尽量简单，团队应尽量保证大部分工作都在主干代码库中进行，只有极少的特定功能会被创建分支。

2.2.4.1　主干代码库的工作

在主干代码库中团队使用的是经典的 Subversion 代码管理流程。

1. 初始化代码

首先开发人员需要从代码服务器端获取项目代码，并保存到本地工作目录。

实例代码：

```
$ svn checkout file:///S:/repos/j2me/trunk
```

2. 同步代码

开发人员需要定期从服务器端同步最新版本代码到本地工作目录，从而集成其他开发人员的代码，实现协作开发。

实例代码：

```
$ svn update
```

3. 开发代码

这个是开发人员的主要工作，根据需求编写代码。在需要对代码文件进行添加、删除、复制和移动时需要使用 Subversion 的命令（svn add、svn delete、svn copy 和 svn move）来完成，并且常需要使用 Subversion 的命令来查看文件的状态（svn status），从而帮助开发人员管理代码。如果需要审查代码，则还需要使用 Subversion 的 diff 命令进行文件内容对比（svn diff）。

4. 修复冲突

开发人员修改代码后，在提交之前都需要再次同步最新的代码，以保证自己的代码与最新的版本没有冲突。但现实情况是经常会出现冲突，在冲突出现后就需要人工对冲突的代码逐行审查，然后选择正确的代码，删除冲突的标志，最后才可以提交代码。修复冲突是一门细致且复杂的工作和流程，所以Subversion提供了强大的功能来协助开发人员修复冲突 [1]。

5. 提交代码

提交代码一般是最后一步，将完成好的代码通过 Subversion 的命令提交到代码服务器上。

实例代码：

```
$ svn commit
```

1　http://svnbook.red-bean.com/en/1.7/svn.tour.cycle.html#svn.tour.cycle.resolve

2.2.4.2 分支代码库的工作

在 Subversion 里，分支的创建和删除相对比较费时，所以在小团队里进行分支相关的工作，也是尽量保持简单的分支管理流程，其中有两个关键的原则。

◎ 保证分支代码同步了主干中的最新代码。

◎ 尽量少创建分支或者重用分支。

注意：在 Subversion 1.5 中增加了 Merge Tracking 这个功能，它可以帮助开发人员跟踪代码合并的历史。

2.2.5 备份策略

在小型团队中，备份代码十分重要。对于 Subversion 而言，一般有两种代码备份方式：客户端备份和服务器端备份。

2.2.5.1 客户端备份

客户端备份指通过 Subversion 的命令定时将代码库的代码同步到备份服务器上，然后自动将其打包并备份。这种方式的优点就是不用停机备份，并且可以自定义备份的范围；缺点是无法备份代码提交日志及服务器端的配置，比如权限配置等。

实例代码：

```
$ svn checkout file:///S:/repos/j2me
```

或者

```
$ svn export file:///S:/repos/j2me
```

2.2.5.2 服务器端备份

服务器端备份是对服务器端的代码库进行备份，主要有两种方法。

（1）使用服务器端提供的命令进行备份，比如 Subversion 的 svnadmin hotcopy 和 svnsync。

（2）直接备份文件。首先需要找到一个合适的时间关闭服务器，比如凌晨；然后通过服务器提供的功能或者直接编写脚本的方式打包整个服务器端的代码库及服务器上的相关文件，比如配置文件等。这种方式的缺点是需要停机才能备份。

服务器端备份的优点包括可以备份所有提交日志、服务器配置文件等。

2.3　阿里云 Code

随着开发进度逐渐紧张，某些员工在出差或者在家办公时仍然需要编写代码，需要通过互联网访问代码库。但是由于服务器维护人员的缺乏，导致无法快速地将内部的 Subversion 服务在互联网上进行提供，所以团队决定选用一款基于云的代码管理系统。而且为了解决出差时没有互联网的情况下仍然可以工作的问题，决定将代码管理工具从 Subversion 转换成 Git。又因为团队成员的工作主要是在国内进行的，所以决定选用服务器在国内的厂商。

通过对多家国内基于 Git 的云代码管理系统进行比较，发现由于它们的核心功能都是相似的（比如在线代码浏览、权限管理等），只是周边的增值服务有所区别。所以根据公司的技术能力和售后服务等情况进行综合考虑，团队最终选择阿里云 Code。由于本地开发机上的 Git 代码库包含所有的提交日志，所以以后就算需要从阿里云 Code 迁移到其他云代码库（比如码云、Coding 等）也是非常容易的。但是现在团队需要解决的问题首先是将代码库从 Subversion 迁移到 Git，然后放到阿里云 Code 上并在互联网上提供代码管理和同步服务。阿里巴巴在阿里云 Code 上还构建了一个阿里云持续交付和项目管理平台 CRP，用于让客户在阿里云 Code 上针对代码进行在线协作需求、测试、缺陷等管理及对代码进行持续构建和发布。

注意：Coding 和码云的收费企业版也支持类似的项目管理功能。

2.3.1 将内网 Subversion 迁移到阿里云 Code

由于项目已经开发了几个月，代码的提交数量也较多，所以迁移 Subversion 代码库时也需要迁移代码提交历史。在团队不大且在同一个城市的情况下，可选择周末没有人提交代码时对 Subversion 服务器进行隔离迁移。

注意：对于随时都有代码提交的大型分布式团队，应该使用 SubGit 这样的商业工具进行不隔离在线迁移，详见 https://subgit.com/。

由于项目代码体量较小，所以直接采用最简单的方案进行迁移：使用工具 git-svn。

1. 代码导出

首先使用以下命令从 Subversion 代码库中导出代码：

```
$ git svn clone -s file:///S:/repos/j2me
```

这样可以在本地生成一个包含原 Subversion 代码库中的提交记录及分支、标签、日志等的 Git 代码库。

技巧：如果需要迁移的 Subversion 代码库是按照标准的 trunk、branches、tags 结构划分的，那么使用-s 这个参数对 Subversion 代码库进行迁移时会将对应的 trunk、branches、tags 存放到本地 Git 代码库的结构中。但是如果 Subversion 代码库不是按照标准的 trunk、branches、tags 结构划分的，则不能使用-s，而是用-T、-b 和-t。其中-T 指定 trunk，-b 指定 branches，-t 则指定 tags，例如：

```
$git svn clone file:///S:/repos/j2me -T trunk -b branches -t release。
```

2. 建立云端代码库

导入代码前首先需要在阿里云Code上建立一个远程的Git仓库。首先访问阿里云 [1]并使用阿里云账号登录该系统，然后在左侧导航栏里选择Projects选项，单击NEW PROJECT，如图 2-2 所示。

1 https://code.aliyun.com

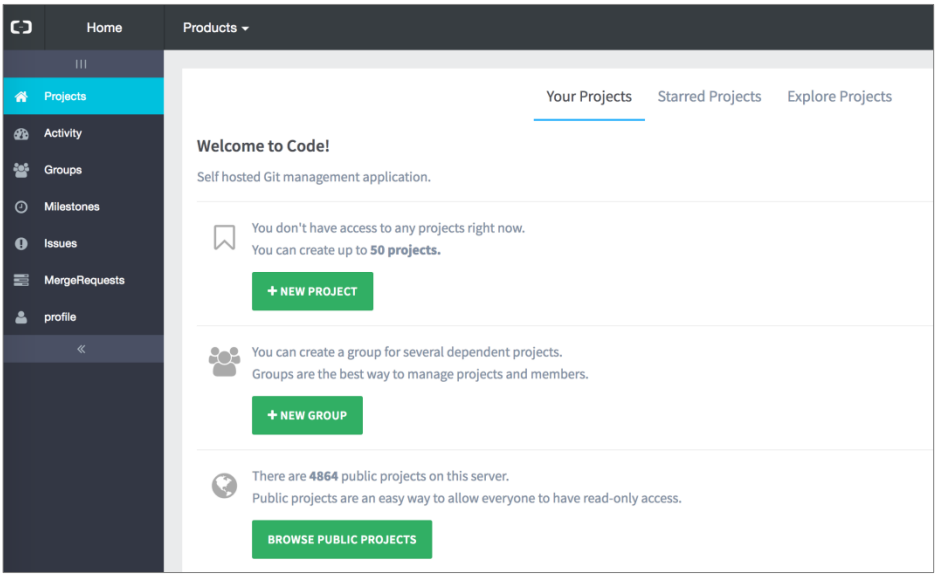

图 2-2　阿里云 Code

在 New Project 页面中的 Project path 中输入项目名 j2me，并在选择 Private 后单击 CREATE PROJECT 来创建新的 Git 云端仓库。由于当前阿里云 Code 还处于公测阶段，所以 Private 库暂时免费，正式版的收费策略还在制订中，并且单个库容量限制为 1GB，单个账户不超过 50 个库，如图 2-3 所示。

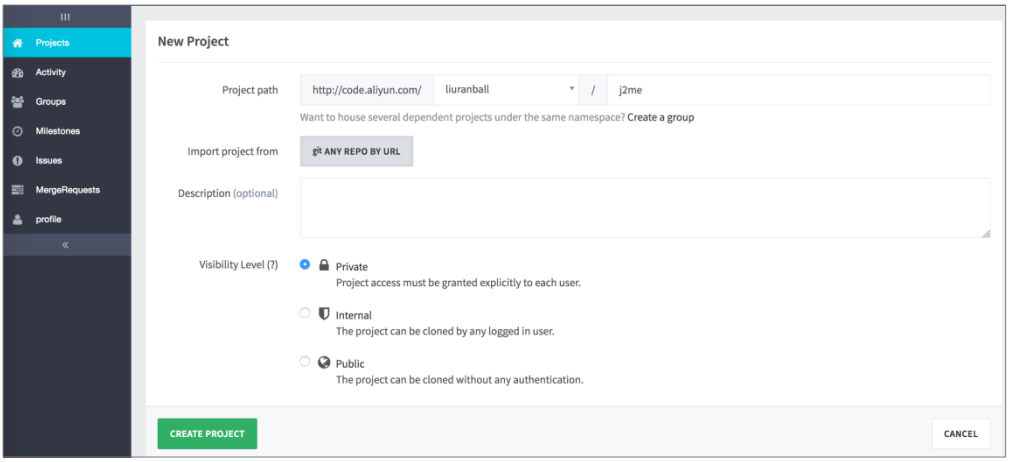

图 2-3　阿里云 Code

单击 CREATE PROJECT 就可以成功地在阿里云 Code 上创建一个代码库，并提供 SSH 和 HTTPS 两种访问模式，如图 2-4 所示。

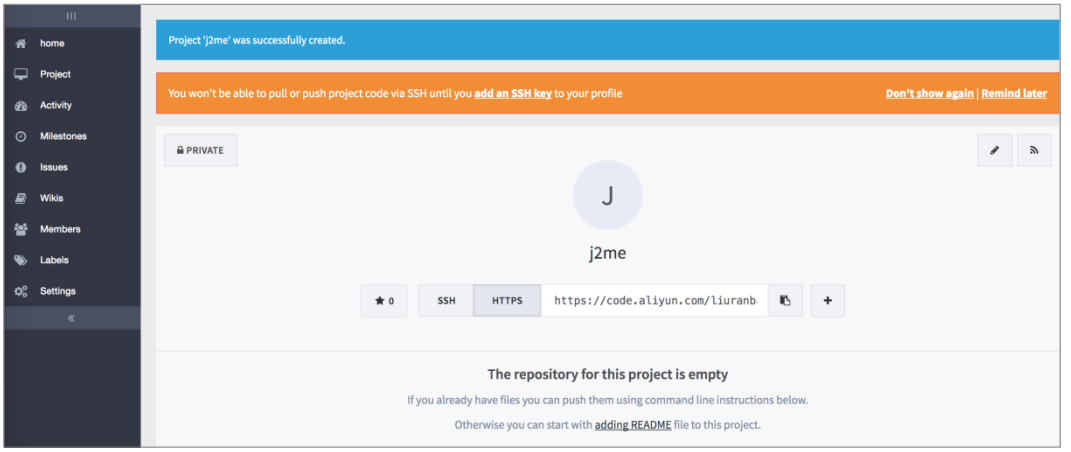

图 2-4　阿里云 Code

SSH和HTTPS都是使用基于SSL的PKI“公钥基础设施”进行认证，其基本流程是首先在客户端生成一个证书对，然后将公钥复制到服务器端的对应的用户SSH证书配置里，客户端就能使该用户在不使用密码的情况下访问这个服务器。基本上每个基于云的代码管理系统都有这部分教程，包括如何生成证书对、如何添加公钥等[1]。

单击如图 2-4 所示的 add an SSH key，并添加公钥就可以完成用户认证配置，然后此用户就可以配合私钥访问阿里云 Code 上的 Git 代码库了。

最后需要在本地的 Git 配置中加入开发人员的用户名和邮件地址，在以后的代码提交里面都包含开发人员的名字和邮件地址，方便查询日志。

```
$ git config --global user.name "username"
$ git config --global user.email "username@email.com"
```

注意：不同的用户需要把 username 和 username@email.com 改成自己的用户名。

1　https://code.aliyun.com/help/ssh/README.md

3. 代码导入

由于前面的 git-svn 命令已经将 Subversion 的代码库迁移到本地并生成了一个本地的 Git 代码库，现在只需要更改 Git 代码库中的配置，使其链接到在阿里云 Code 上创建的新 Git 代码库，运行以下命令就可以实现这个链接：

```
$ git remote add origin https://code.aliyun.com/liuranball/j2me.git
```

然后运行 git push 命令将本地的 Git 代码上传到阿里云 Code 的远端代码仓库里：

```
$ git push origin --all
```

最后查看代码提交页面，可以看到 Subversion 以前的分支和提交历史记录，最终迁移完成。

其他项目相关的设置请参见其官方帮助文档 [1]。

注意：如果团队并不需要保留 Subversion 库中的提交历史，并且没有 tag 和 branch，就不需要使用 git-svn 这个命令，可省去这个命令相关的所有步骤，直接从 Subversion 将代码导出为一个干净的代码库，并直接复制到本地新的 Git 代码库中后再运行提交并上传代码，就完成了迁移工作。

2.3.2　权限管理

权限管理是基于云端的代码管理系统中最为重要的一部分特性，因为代码服务器是直接部署在互联网上的，如果权限没有配置正确，则很有可能产生安全问题，导致代码泄露。

阿里云Code的权限管理系统使用了和Gerrit类似的基于用户组的权限管理方式 [2]，默认将权限分为多个组，分别是Guest、Reporter、Developer、Master和Owner，并给每个组分

1　https://code.aliyun.com/help

2　https://code.aliyun.com/help/permissions/permissions.md

配特定的权限。每一个用户需要属于某个组，然后获得这个组拥有的权限，这样就可以让管理人员比较容易地对每个用户进行权限管理。

2.3.3 日常工作模式

由于仍然是一个小团队，所以日常的工作模式和 Subversion 时类似，只是使用的命令和代码管理的步骤有所不同，其中包括依然使用一个开发主干模型，而没有采用 Gitflow 这种复杂模型等。

首先，注册阿里云用户并通过设置自己机器的 ssh public key 及管理员添加访问用户，从而获得代码库的访问权限，然后使用 git clone 命令初始化代码库。

其次开发与提交代码，如果修改的代码很少，并且能快速完成，就直接在主干代码上进行，并使用以下命令完成代码修改：

```
$ git add ./
$ git commit -m “message of commint”
$ git push
```

如果代码修改得比较多，一天内无法完成，就需要创建一个开发分支来进行开发，开发完毕后再合并至开发主干（比如开发主干是 master）：

```
$ git branch DevBranch
$ git checkout DevBranch
```

修改代码：

```
git add ./
git commit -m “message of commint”
git checkout master
git merge DevBranch 或者 git rebase DevBranch （如果有冲突，则这个命令会提示有冲突，然后解决冲突并完成合并。）
git push
```

git rebase 和 git merge 的比较如表 2-1 所示。

表 2-1　git rebase 和 git merge 的比较

	git rebase	git merge
主要用处	合并代码并修改当前的分支历史	合并代码并不能修改当前的分支历史
优点	历史记录漂亮，方便查看	分支合并时操作简单，代码回归方便
缺点	某些情况下操作复杂，比如分支合并、代码回滚	生成额外的提交记录，历史线多条，不方便查看
注意事项	不要为了追求漂亮的历史记录而滥用 rebase	有洁癖者勿用
最佳实践	git pull 和修改多个本地提交时	分支合并时

最后，如果遇到代码提交错误，则可以使用git revert进行回滚；如果代码提交需要更改，则可以使用git commit --amend；如果需要管理单个代码提交，比如获取某个代码提交的代码更改，则可以使用git cherry-pick等 [1]。

2.3.4　备份方案

由于阿里云 Code 是阿里云的官方服务，基于阿里云服务器，所以稳定性较好，不用特别对服务器端的系统进行备份，备份涉及的重点是代码及代码提交日志。对于小型团队，在人力成本有限的情况下可以使用以下两个备份方案。

方案 1：开发人员自己备份

由于每个开发人员的本地代码库包含了所有代码日志，所以每个开发人员都有一个完整的代码及日志备份。方法是每个开发人员最好每天上班和下班时都通过 fetch 或者 pull 同步一下本地代码。这个方案的优点是不需要额外的工作，简单、速度快；缺点是不易控制，而且以后还原时需要对每个开发人员的代码版本进行检查以确定最新的版本。

1　本书不是 Git 的入门书籍，关于 Git 的基础知识请参考书籍《Git 版本控制管理》 和 《Pro Git》

方案 2：通过特定的服务器备份

如果担心开发人员忘记同步代码或者破坏本地代码，则可以使用一台特定的服务器对代码库进行定时备份（比如每 10 分钟备份一次）。方法是首先找一台服务器（可以是本地的一台专用机器，或者是一台云服务器），然后在这台机器上首先对代码进行 clone，接着定时进行 pull 操作。定时实现方式可以是用操作系统的默认定时器如 Linux 的 cron，或者使用第三方任务系统如 Jenkins。这个方案的优点是稳定、控制简单，缺点是需要额外的人力和硬件投入。

2.4 小团队代码管理的经典模型

1. 选择免费或者低成本的代码管理系统

小型团队没有必要选用成本较高的 ClearCase、Perforce 等商用代码管理系统，选用免费的 Subversion 和 Git 就可以了。如果没有非常严格的代码限制要求，比如代码必须在企业内部或者代码必须在国内，那么就可以选用 GitHub、Coding 等云代码管理系统，这样就可以节省代码服务器的大量管理和维护成本。虽然像 GitHub 这样的代码服务器已经非常成熟了，但是仍然存在程序员泄露登录密钥而导致代码泄露的风险，因为在现实中经常发生这样的情况，所以团队实施严格的密钥使用机制是十分有必要的。

2. 代码管理系统能快速搭建

小型团队一定要选用能快速搭建的代码管理系统，这样才能在最低的成本下和最短的时间内建立好代码服务器。而且需要拥有成熟社区，遇到问题时可以快速在社区找到答案。

3. 可以不设置或者简单设置权限管理

小型团队由于人数不多（一般几个开发），所以没有必要配置复杂的权限系统。但是如果一定要设置完善的权限管理，则请参见本书第 3 章和第 4 章。

4. 使用经典的代码分支管理模型

小型团队开始时可以直接使用经典的代码分支管理，没有必要使用过于复杂的分支管理模型。但是随着软件功能的增加，开发人数增加，可以根据实际情况重新定制分支管理模型，请参见本书第 3 章和第 4 章。

5. 日常代码管理流程尽量简单

小型团队应尽量使用简单的代码流程，包括代码提交、代码审查、代码集成等。代码审查很可能不需要使用在线代码审查工具，开发人员直接面对面审查即可，代码集成也可以尽量在一个主干中完成。

6. 本地代码库快速定时备份

对于小型团队，备份方法的第 1 个选择是直接在某台机器上通过定时脚本直接同步服务器端代码，然后打包备份；第 2 个选择是使用服务器端的工具或者编写脚本直接在服务器端进行代码库和服务器配置文件的备份，在能力和资源允许的情况下，最好还是定时在服务端进行备份。

第 3 章

传统中大型团队

3.1 传统大型团队的特点

21 世纪初，随着软件规模的快速增长，软件开发团队的规模也在持续扩大，在一些复杂的业务领域如通信行业，为连接两个移动设备所涉及的软件开发团队就达上千人。这样的人员规模化很快带来了下面两方面的挑战。

（1）代码量非线性激增。

（2）模块依赖关系复杂。

如图 3-1 所示，代码量的增长和依赖关系的复杂是互相促进的，最终造成整个代码库复杂度的失控。

团队规模和代码量的相互作用产生了恶性的增长循环。团队规模的增长与代码量的增长并非线性关系，很多逻辑上的重复在多人并行开发的情况下演变成了一个巨大的问题，很难想象打通一个电话需要上亿行代码来支撑。而代码量的增加又让任何后续的修改和新

功能的实现变得越来越困难，在市场压力下增加人数成了唯一的选择。

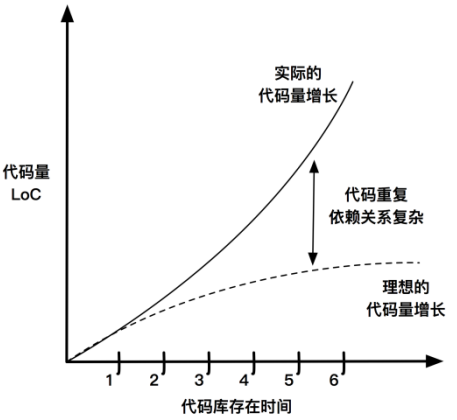

图 3-1　代码库存时间图

另一方面，当软件规模增大后，更多的人开始注意模块化的设计，这和很多行业类似，通过模块化封装降低软件的复杂度，最后通过搭积木的方式进行组装。软件模块化设计在复杂应用如通信和金融领域是十分常见的，高内聚、低耦合成了软件基本的设计原则。但模块化的设计必然造成一个应用内部的多模块依赖，依赖关系的梳理往往成了一个大型软件团队的痛点。在很多存在了十年以上的应用里仍然可以看到混乱的依赖关系，似乎已经没人能够梳理清楚哪些代码是必需的，成功编译或打包的“最佳实践”就是把所有相关代码复制到一起。

值得一提的是，很多大型团队在进行模块化设计时过度追求技术上的解耦，前期一般都能够画出一张漂亮的系统模块图。但随着业务的持续变化和演进，这样的模块化就成了累赘，不但没有起到降低复杂度的作用，最后反而成了代码在各个模块重复的根源。康威定律在长期的持续开发过程中产生了作用，这样的系统造成人员和代码量上的持续增长。现在微服务架构的诞生主要就是针对这样的问题，希望能够正确地对业务需求产生合理的组件化来降低整体复杂度，保证持续演进时成本可控。我们将在第 5 章中介绍具体的相关代码管理实践。

如图 3-2 所示，按照技术模块“解耦”的架构在响应新业务需求时必然带来很高的修改成本，这时模块的内聚性其实是非常低的。在图 3-2 的场景下模块 A 和 B 针对新的两个业务需求的修改就需要较重的流程机制来协调和同步。

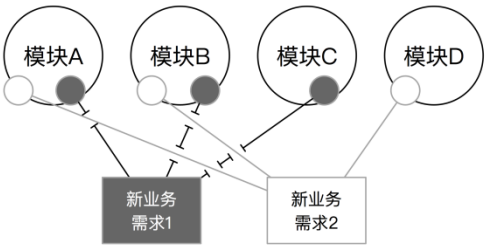

图 3-2　新业务需求与代码模块

3.2　独立大型团队在代码管理上的痛点与需求

独立大型团队在代码管理上存在以下痛点。

（1）频繁的代码冲突及合并。

（2）多文件原子提交及回滚。

（3）跨团队代码共享及依赖管理。

由于存在以上痛点，很多大型开发团队一般都会采用强流程控制的方法来避免冲突，不少团队还存在专门的版本管理员来负责每个模块代码库的最终合并及发布。在很多采用 ClearCase 的开发团队中，每个开发小组（或开发人员）都有一个 stream（代码分支），开发功能测试通过后才由专人提交到相应的特性分支上供集成或系统测试人员使用。在这个过程中 ClearCase 的文件锁也被采用为避免代码冲突的工具，即每个代码文件只能由一个开发人员检出，在该开发人员没有检入此文件之前，其他开发人员是不可能对此文件进行修改的。

这样的重流程管理看似解决了代码冲突合并的问题，也规避了多文件原子提交和跨团队代码共享的挑战，事实上却导致了两个严重的后果。

第 1 个后果是从代码到发布的生命周期拉长，特性分支成为了解决“长”开发周期和“快”业务需求的唯一方法，一个 2～3 年的产品最多可以有上百个特性分支。一个分支的

创建成本很低，但随着时间的推移，维护成本呈几何级数增加，到最后很多分支因为年久失修成为了无源的代码复制，开发人员在增加新功能时需要把代码“复制”到不同的分支上。很多大型团队羡慕能够坚持单主干开发的团队，对比后发现自己的开发效率很低。这个问题的核心就在于这种长期累积出来的开发模式，开发人员已经习惯于把代码集成工作推到最后去做，甚至在开发过程中希望锁定自己需要的程序文件从而使别人无法打扰。

第 2 个后果是代码的质量腐化严重，由于大家都在规避代码的集成工作，久而久之便形成了每个人的代码“领地”。当一个开发人员需要开发一个新功能时，即使发现了类似的已有设计（模块或函数），其第一选择也是重新写自己的，避免引起“不必要”的合并。在前面 ClearCase 的文件锁模式下，当需求方催促很紧时，很可能的解决方法就是复制一个程序文件让新功能走新的一个分支。这样的后果是显而易见的：大量的代码重复及内部的依赖变得千丝万缕。当团队尝试引入重构去提升代码的质量时，就会发现不是人员的能力跟不上，而是这个机制根本不支持重构。看似最简单的重命名都因为涉及很多代码文件而无法执行，因为代码复制甚至自动化的重命名也无法进行。

有以上痛苦经历的大型团队最终都意识到良好的模块化及代码实时集成的重要性，在敏捷核心开发实践持续集成中提倡：把最后集成的长痛变成时刻集成的短痛，直到最后让集成成为每次提交代码的一部分，而不痛的原则在大型团队里是至关重要的。下面为大家回顾一个大型团队将代码库迁移到 Subversion 的历程，希望能够从代码管理的视角帮助大家理解怎样才能建立支撑良好模块化和代码实时集成的开发环境。

3.3　大型团队代码管理案例

这是一个非常典型的大型团队，基于 Java 平台开发了近十年，积累了两千多万行代码，基于 ClearCase 建立了一整套完整的代码管理机制，通过版本经理管理着不同的特性分支。遇到的痛点自然与前文叙述的一致：开发效率越来越低，测试的时间越来越长，代码的质量只能通过每年一次的集中“整治”来修补。

团队技术管理者希望能够采用敏捷的小步迭代开发模式，来提升整个开发生命周期的

响应力。然而产品需求是拆分成小颗粒度的 Story，开发人员却无法按照这样的 Story 来实现。每个 Story 都会涉及多个模块的修改（见图 3-3），多个 Story 在 ClearCase 传统文件锁的模式下无法并行开展，开发人员突然感到互相牵制，无法按照这种方式展开协作。

如图 3-3 所示为需求在用户故事（Story）的管理模式下的多模块并行修改。由于每个 Story 都希望是端到端的需求定义，所以会要求我们同时修改各个架构层的模块。这就很容易出现如图 3-3 所示的场景，刚开始 Story #1 和#2 已经占据了 Story #3 所要修改的所有模块的代码文件，Story #3 的开发只能被迫等待。

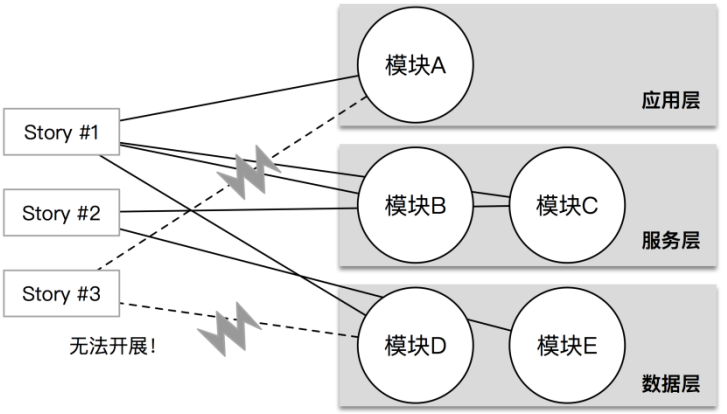

图 3-3　用户 Story 与代码模块

很快大家发现，现有的代码管理平台和机制是无法支撑敏捷开发模式的，要想做到敏捷开发提倡的小步快跑的高响应模式，就必须转变代码管理机制。经过前期的分析，考虑到核心代码权限控制的要求，决定采用 Subversion 作为代码管理平台。基于分布式思想的 Git 在权限控制方面的设计决定了控制核心代码（如算法实现）是比较困难的，当然这也不是有开源基因的代码管理平台的设计初衷。

随着 GitHub 的流行，很多团队都会把 Git 作为默认的下一代代码管理平台，但对于一些有高安全性要求，而且需要自主管理代码的大型系统产品，Subversion 仍然是一个兼顾了灵活性和安全性的高性价比选择。在现实开放过程中，我们也发现很多团队把 Git 用成了 Subversion，把一个 remote 作为“中央”来对待，在这样的情况下应该考虑自身的客

观需求，考虑是否需要分布式的管理机制。分布式在带来灵活性的同时，也带来了其他方面的额外管理成本的增加，最显著的就是代码的权限控制。

3.3.1　代码模块依赖管理

当团队热情高涨地搭建好了 Subversion 服务器并准备检出 ClearCase 代码，然后解放开发生产力时，其遇到的第 1 个巨大挑战却是模块及程序间复杂错综的依赖关系。满怀信心的开发人员发现，如果直接从 ClearCase 切换到 Subversion，则不但没有预期的收益，反而会增加代码被间接修改造成系统功能失败的风险。特别是一些现代的 IDE（如 IntelliJ）提供了很强大的自动化重构支持，Subversion 基于原子操作的设计很容易因为一个重构动作就修改了一系列文件而被作为一个原子提交。过去基于代码文件的模式则需要开发人员逐个确认需要提交的文件。

团队很快就意识到了这不是一次简单的代码从 A 库到 B 库的迁移，而是从根本上改变了开发习惯和架构的转型。为了能够享受 SVN 提交原子管理的思想，就必须有明确的模块架构及依赖关系。其中一个标志就是能够让各个模块在编译过程中依赖必须引用的二进制包，而不是外部模块的源代码。

在模块化开发已经形成共识的情况下，模块依赖管理也有了标准实践，在 Java 平台下我们采用了最成熟的 Maven 体系，用 Maven 定义的软件包生命周期（见表 3-1）来形成代码模块的约束，根据 Maven 的要求对每个模块进行显式的依赖申明。

如表 3-1 所示，Maven 默认的软件包的生命周期一般保持以上顺序执行，若上一阶段失败则不进入下一阶段。

这时挑战来了，我们面对一个有两千万行代码的代码库，各个模块在最近两年已经丢失了严格意义上的边界，大家在编译时基本上也是囫囵吞枣，拉入尽可能多的模块源代码一起编译。如何确定各个模块的依赖关系本身就是一个大问题。经过一段时间的搜寻，我们决定效仿 C++项目中大家比较熟悉的头文件整理方式，通过扫描 Java 类文件的 import，绘制出目前各个模块的依赖现状。一个简单的文件扫描程序很快在 1 个小时内给出了结果，

大家看到依赖关系图还是比较惊讶的，居然有两处循环依赖！也有很多依赖让大家摸不着头脑，不知道其是如何产生的。在这个过程中我们一次性找到了 20 多处需要整改的依赖关系。

表 3-1　用 Maven 定义的软件包生命周期

阶　段	解 释 说 明
process-resources	复制和移动代码库到指定的目标目录，为打包做准备
compile	编译项目对应的代码库
process-test-resources	复制和移动测试代码库到指定的目标目录
test-compile	编译测试对应的代码库
test	采用适配的框架来执行相关测试
package	将代码库打包成指定的格式，例如 JAR、WAR 或者 EAR
install	在本地库安装软件包，作为其他本地项目的依赖
deploy	将软件包复制到远端库中，作为其他开发人员和项目的依赖

第 2 个挑战更为艰巨，系统有 200 多个模块，每个模块都需要植入 Maven 的显式依赖申明机制，即需要有一个 pom.xml 文件来包含所有被引用的外部模块，同时需要规范自己的命名空间。如果让负责每个模块的团队自行构建，那么预计会有一个星期以上的混乱，这是团队不能接受的。当然这样的“重复”工作显然是要自动化的，在之前分析依赖关系程序的基础上利用强大的正则表达式，我们快速实现了一个自动扫描和依赖 pom 文件自动生成的程序。

这个过程中最复杂的是如何确定命名空间和版本号的使用，由于当时团队仍然有不少特性分支，完全遵循 Maven 的标准规范不足以应对，所以在这个过程中我们仍然沿袭了团队用大小版本号来区分新发布和新特性的习惯。

如图 3-4 所示，在 Maven 的版本管理中，一般将 SNAPSHOT 作为开发版本，发布版

本则去掉了 SNAPSHOT 的标识。很多团队都采用了 major.minor.patch 的模式，例如 1.1.0。第 1 个 major 数字代表一次大版本，minor 代表大版本下不同的小版本，往往包含了不同的特性、功能，最后一个数字 patch 则当作问题修复标识。时下流行的 GitFlow 模式也采用了这样的管理办法。

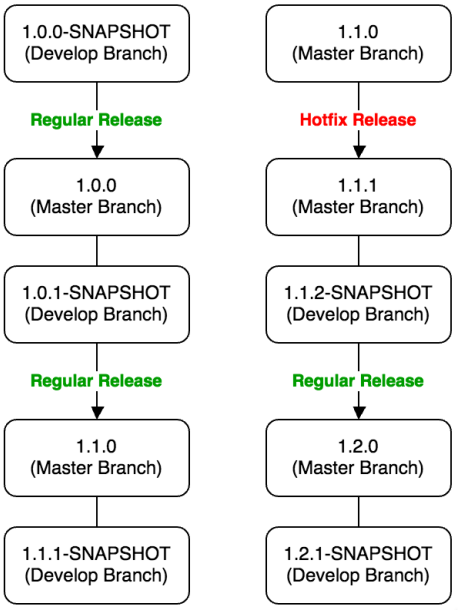

图 3-4　Maven 的版本管理

经过测试，我们发现新程序面对这两千多万行代码需要超过 30 个小时的执行时间，在快速调整了程序结构后性能仍然没有特别大的提升。由于这是一次性的“安装”工作，所以我们没有再花更多的时间优化程序，而是把主要精力投入到了评审各个模块团队针对前期发现的依赖问题的整改方案及改进。两个星期后的周末我们成功完成了这次安装，使这两千多万行代码真正有了代码依赖管理的体系。

这令团队欢欣鼓舞，这种对依赖的显式申明让开发人员和测试人员都觉得质量更有保障了，编译的问题能够被很容易地锁定到一个比较小的范围，整个代码库的管理复杂度也由此降低了很多。几乎同一时间，代码被从 ClearCase 迁移到了 Subversion，接下来的周一，所有开发人员上班后的第 1 件事是从新的 Subversion 库检出自己需要的模块代码。在

接下来的一周里，不少团队开始有了真正意义上的代码走查，大家开始讨论如何小步重构来提升局部的代码质量。

3.3.2 建立相关运作机制

由于 Subversion 本身的权限管理比较完备，这次迁移又接入了已有的 LDAP 认证机制，所以开发团队在这方面基本没有任何不适应。我们通过 Subversion 服务器的控制面板很快配置好了团队和模块之间的权限对应关系，也为一些超级用户开通了更大的代码库访问权限。当然在开源时代，我们认为让所有人看到所有代码其实是利大于弊的，我们也看到了开源软件的一些核心算法是完全开放的。

在这个案例里我们没有完全开放所有代码给所有团队成员，但我们坚持在一个模块内将所有代码开放给对应的团队，因为只有这样才能够使代码及架构的质量持续提升。这样的改变在前期还是引发了小小的风波：有些开发人员觉得失去了对自己修改的文件的“控制”，同一时间不知道有多少人在修改同一个文件。为什么要关心同一时间有多少人修改了一个程序呢？其原因无非是害怕之后需要代码合并。其解决方案是改变对代码集成的认知，让它成为自己开发过程中的一个步骤，小步提交、快速合并。

这时另外一个制约因素也起到了作用：团队规模。迁移到 SVN 后每个模块团队仍然有 30 人左右，其中开发人员有 20 人左右。不少开发人员逐渐发现只要自己提交得快就可以避免代码合并的工作量，所以“聪明”者写一两个函数就提交一次，导致代码片段化提交。其结果当然是团队的其他同事们合并代码时完全看不清逻辑，以至于合并时间拉长。在当时的模块结构下，进一步拆分团队并不现实，所以我们建立了提交的“令牌”机制，希望通过显式的声明让大家增强协作。具体运作其实十分简单，我们在团队办公区的中央挂了一个毛绒吉祥物（见图 3-5），每次需要提交的同事必须拿到这个吉祥物才能够提交。提交完毕需保证编译成功后才能归还吉祥物，如果提交出现问题，15 分钟内无法修复就必须回滚。

如图 3-5 所示为提交令牌机制现场，团队采用了兔耳朵作为“吉祥物”，能够显式地告诉团队的其他成员谁正在做提交。

图 3-5　提交“令牌”

令牌的方法只是一个临时方案，在一两个星期里却让我们看到了更多的团队沟通。可见代码管理最重要的一点是如何让负责这些代码的团队能够有更多的透明信息和更顺畅的沟通。虽然团队协作不是本书的话题，但我们也从这里看到了协作这个话题在软件开发方面的重要性。

最后一个难点是解耦后各个模块之间的版本关系，一个模块出了新版本后如何能够通知到依赖它的其他各个模块？这时 Maven 仓库起到了至关重要的作用，每个模块成功通过测试的包会被测试人员推送到 Maven 仓库里，其他模块编译时会去 Maven 仓库检查自己依赖的库是否有更新。当然，在这样一个复杂的代码库里其实有很多小问题，比如最初我们认为在开发过程中用简单的 SNAPSHOT 版本来标识当下最新的测试通过版本就可以了，但在实际运作中还是出现了需要插入紧急版本而手动修改依赖版本的情况。虽然 Maven 提供了自动的脚本化升级版本机制，但这种设计本身的理念是单一主干开发，在存在大量特别性分支的系统里使用还是会让人感觉到各种别扭。这里我们的经验是不要先去定制复杂的版本号规则来“适配”各种分支，出发点应该是如何减少分支。我们发现换一个视角思考会带来更长远的效益。

这里对运作机制做一个小总结，有以下三个方面是需要重点考虑的。

◎　代码库的权限管理。如果客观情况下还不能开放所有代码给所有人，那么中央的代码管理平台如 Subversion 可能就是一个更好的选择（较之于 Git）。

◎ 代码提交流程。帮助每个开发人员从全局考虑，而不是仅仅为了自己方便影响到整个团队的效率。

◎ 代码模块依赖。一定要明确代码的模块化结构，保证模块间不存在源码的依赖，而仅仅是接口上的依赖。多模块同时开发需要有明确的依赖管理，不管是采用流程上类似每天一个新版本更新，还是自动化的版本刷新，最重要的是在团队里达成共识。

3.3.3 建立原子提交的纪律

在基本的运作机制建立起来后，我们需要进一步规范开发人员的提交习惯，基于Subversion有规范的七步提交法则，如图3-6所示。

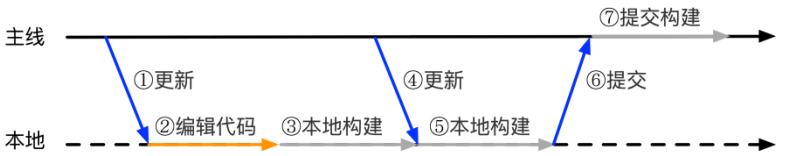

图 3-6 七步提交法则

这七步看似烦琐，其实一个熟练的开发人员每次提交几条命令执行也只需一两分钟。本地构建一般包括了在本地执行的自动化单元测试和一些回归性的端到端测试，这样能够尽量减少错误的扩散成本，即提交了不能执行的代码在后期被测试返工。

值得注意的是，如果第4步更新有代码改动，那么应该再次执行第5步的代码构建。Subversion（和很多代码管理平台如Git）有自动合并代码文件的功能，大多数时候没有问题，然而一旦合并有问题但没有被及时发现并提交到中央代码库，带来的修复成本会很高。曾有好几次由于开发人员的侥幸心理造成了代码文件错误地被自动合并且提交，等大家发现时已更新了版本且无法本地编译，这时错误的成本就是整个团队的了。

其实最难的是让开发人员理解并实践原子这个概念，即提交的最小单元。Subversion可以把多个程序文件的修改打包成原子一并提交或者回滚，利用好这个特性才能真正建立

代码层面的知识共享。一个好的开发人员在给团队解释自己的实现时能够结合业务场景和每次提交的 code diff 清晰地展现自己的实现思路，整个过程是赏心悦目的，听众能够在这个过程中逐步理解业务的需求及对应的代码实现。很多团队发现代码走查很难有效果的核心症结也在于此，讲解已经完全实现的代码是很难让大家理解其实现过程的，也很难让大家评判当下实现方式的优劣。

当然我们说的原子提交，追求的是每次提交都能够实现一个小功能或场景（也包括优化现有实现的代码重构）。对 Story 方式需求管理熟悉的开发团队会发现原子提交的方式对于他们来说是天然友好的，每一个故事、每一条验收条件都可以作为一次提交的原子目标。当然，如果再结合 TDD 的开发模式，用每个单元测试来描述一个小场再加以实现就让这种方式更加流畅了。这也是我们推荐给这个团队的方法，通过 TDD 实践来学会如何“拆分”出一个个小场景并加以实现和原子化提交。

值得一提的是，建立这样的原子提交习惯是让每个开发人员都有团队集体意识的一个过程。如果完全从每个人自己的效率出发，则这样的要求会显然引入额外的成本，我们可以想象如果没有这样的开发纪律，则开发人员各自为战只会走上老路，回到过去的低响应力开发模式。

3.3.4　建立持续集成守护机制

从一开始我们就强调了将持续集成作为一个团队协作实践的重要性。随着代码依赖关系梳理完毕且迁移到 Subversion，能够真正实施持续集成的条件也逐步满足了。我们总结了基本的持续集成纪律，如下所述。

◎　构建失败后不要提交新代码。

◎　提交前在本地运行所有的提交测试。

◎　等持续集成测试通过后再继续工作。

◎　回家之前，构建必须处于成功状态。

◎ 时刻准备着回滚到前一个版本。

◎ 在回滚之前要规定一个修复时间。

◎ 为自己导致的问题负责。

由于这样庞大的代码量和对应的百人开发团队，我们不可能让所有人共享一条持续集成流水线。前期的依赖关系梳理和解耦给我们提供了很好的分级搭建流水线的可能。以每个模块为单位，我们尝试划分流水线，一些耦合比较紧密的模块被放到了一个流水线上，我们关注的还是尽量让一个开发小组（20 人左右）有一条独立的流水线，这样才能较为彻底地执行以上持续集成的纪律。

反过来，如果某一个团队的流水线有对其他团队的代码的依赖，那么持续集成的纪律是建立不起来的。当被依赖团队出现提交错误时，持续集成的失败就很难在这个团队解决了。在遇到这样的问题时，其实是一个反思目前代码依赖关系的契机，我们往往应该思考模块架构的设计是否需要进行重构优化。

在为十多个团队都完成了基于 Jenkins 的流水线搭建后，我们也搭建了上一层基于完整版本的主流水线，构建了如图 3-7 所示的分层流水线结构。由于一代 Jenkins 并没有内建流水线（pipeline）的概念，所以两级流水线的“组装”工作都是由自动化脚本实现的，主要功能就是在支持底层流水线完成后自动触发上层流水线。在一些稳定发布分支上几个重要模块的修改需要自动触发上级流水线的构建。而一般情况下上级流水线的触发交给了系统测试团队，来决定进行一个新版本的构建是否有价值。

在如图 3-7 所示的分层持续集成流水线架构中，模块 A、B 和 C 分别有一条流水线，“逻辑决策”是否执行第二级流水线由测试团队决定。在最后一个手动决策点主要考虑是否需要进入用户的体验测试环境中。

目前二代 Jenkins 已经内建了流水线的概念，但要支持这样的自动“扇入”（Fan-In）还是需要额外用脚本定制的。值得一提的是，由于可能的脚本漂移（即在不同的环境中执行同一脚本得到不同的结果），我们还是选择在下级和上级流水线中分别从代码开始构建，而没有直接在上级流水线中引用下级构建的产出物（Maven 仓库）。上级流水线的编译和打

包环境被隔离，不安装任何其他软件包（如安全检查），从而能够尽量减少和生产环境的差异。

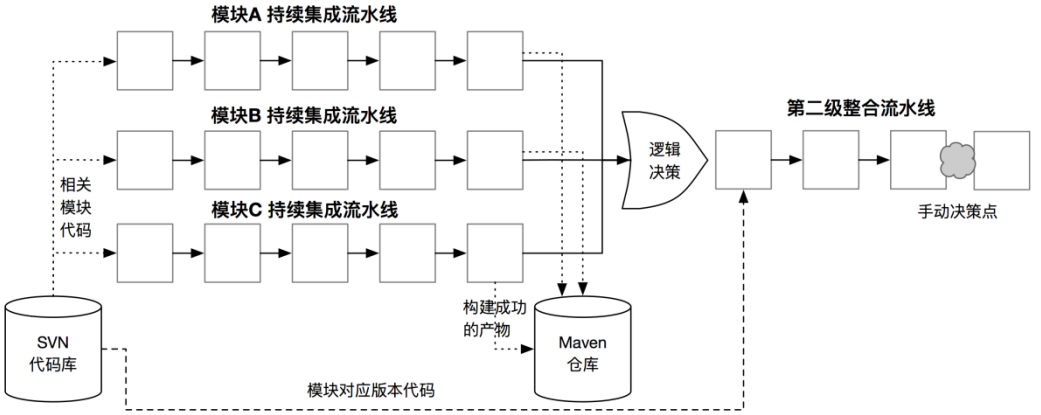

图 3-7　分层持续集成流水线架构

在一切就绪后，我们开始在各个团队安装信息辐射看板，这是很有意思的事情，一般我们会在团队的区域安装流水线的展示屏。在一些有条件的团队中我们还会安装各种提示灯，如图 3-8 所示是一个团队的持续集成警报灯，通过红绿灯的方式可更直观地将结果通知给团队。由于架构的约束，每个团队的规模仍然较大，所以前面的“令牌”机制仍然在执行。当然，具体规则变成了拿到令牌的开发人员在保证自己团队的持续集成流水线完全通过后才能够释放令牌，如果流水线失败则有 15 分钟来修复或者回滚。在如图 3-8 所示的案例中我们增加了小小的惩罚，负责修复流水线的开发人员需要戴着图中那顶小红帽。

在如图 3-8 所示的持续集成团队现场，我们除了采用了小红帽的警示机制，还采用了红绿灯来更直观地展示持续集成的当前状态。

经过一个多月的磨合，团队逐渐开始形成了新的开发节奏，每个开发人员每天提交代码 3～5 次，合并代码成了平常开发活动的一部分，在有争议的地方两个开发人员会自动讨论解决。技术债在很多小团队里开始被公布并进入团队的质量管理任务里。

限于本书的目的，我们不再进一步讨论持续集成系统的优化，下面提炼几条内容仅供有类似需求的团队参考。

◎ 持续集成流水线的执行效率：考虑编译及测试工作的并发执行，并关注自动化测试的数量。

◎ 持续集成任务执行服务器管理：考虑各个服务器的工作负载及数据的隔离，并考虑容器技术的应用。

◎ 代码结构的质量管理：考虑结合良好的代码实践设计检查和告警，比如发现新的模块依赖被建立时告警、发现测试覆盖率下降时告警等。

图 3-8　持续集成流水线失败

在《大规模敏捷开发实践：HP LaserJet 产品线敏捷转型的成功经验》一书中 HP 的三位主要转型负责人针对 HP LaserJet 固件 400 多人的开发团队描述了其 4 年的转型历程，其中类似的分层持续集成流水线在迁移代码到 Subversion 后起到了相当重要的作用，而达到的效果也非常惊人。

◎ 开发人员的开发效率提升了 8 倍，开发新特性的时间从过去用 5%减少到 4%。

◎ 产品问题的支持经费从过去的 25%减少到了 5%。

◎　从 10 多个的长期特性分支转换到了单一主干开发模式，并构建了完整的自动化测试集。

◎　每次产品的构建周期从过去的 1 周缩短到了 1 小时，一天可以有 10～15 次完整构建。

如何从一个有 10 多个长期特性分支，每次产品需要 1 周构建和 6 周回归，

3.3.5　大型团队代码管理小结

上面的案例为我们展现了较为完整的大型团队进行代码管理实践的变革过程。很多核心软件产品由于经年累月的开发，目前都面临着如该案例所示的团队之前的困难和挑战。在回顾的过程中其实很难把更多的具体矛盾一一呈现，这个团队在整个转型过程中的坚定态度是保证最后成功的基础。这里我们也希望通过这个案例让更多处于同样困境中的团队坚定转型的信心。

如果浓缩到一个关键点，那么这个过程是代码库的解耦及围绕代码的协作纪律进行的变革。最终从 ClearCase 迁移到 Subversion 平台仅仅是这个过程的副产物。在面对大型团队时我们所选择的开发实践及方法决定了对代码管理平台的选择，而我们所选的代码管理平台又反向约束了我们的开发实践和方法。

当前有不少大型互联网产品团队正在经历代码平台的迁移，比如从 Subversion 到 Git。同样，这样的选择来源于开源时代的开发实践和方法，我们的代码管理平台需要支撑更加分布式的协作方式，需要提供更大的灵活性。接下来将介绍一些从 Subversion 到 Git 的迁移方案。

3.4　大型团队的代码服务器迁移

对于想从集中式代码管理系统迁移到分布式代码管理系统的团队来讲（典型的从 Subersion

到 Git），如果团队规模小，那么问题不大，但是这对于大型分布式团队来讲困难重重。主要有以下两个困难。

（1）代码量太大，很难一次性将所有代码和日志都在短时间内迁移成功。

（2）下属团队太多，很难在同一时间让所有团队都切换至新的代码管理工具上。

为了解决这些难题，一般会首先选择一个团队来使用新的代码版本管理工具。如果这个团队转换成功，则再将其作为标杆向其他团队推广，从而逐步将所有团队切换到新的工具平台上。

在从Subversion到Git的迁移方案中主要会用到两种工具：开源、免费的git-svn[1]；商业、收费的SubGit。整体迁移方案如图 3-9 所示。

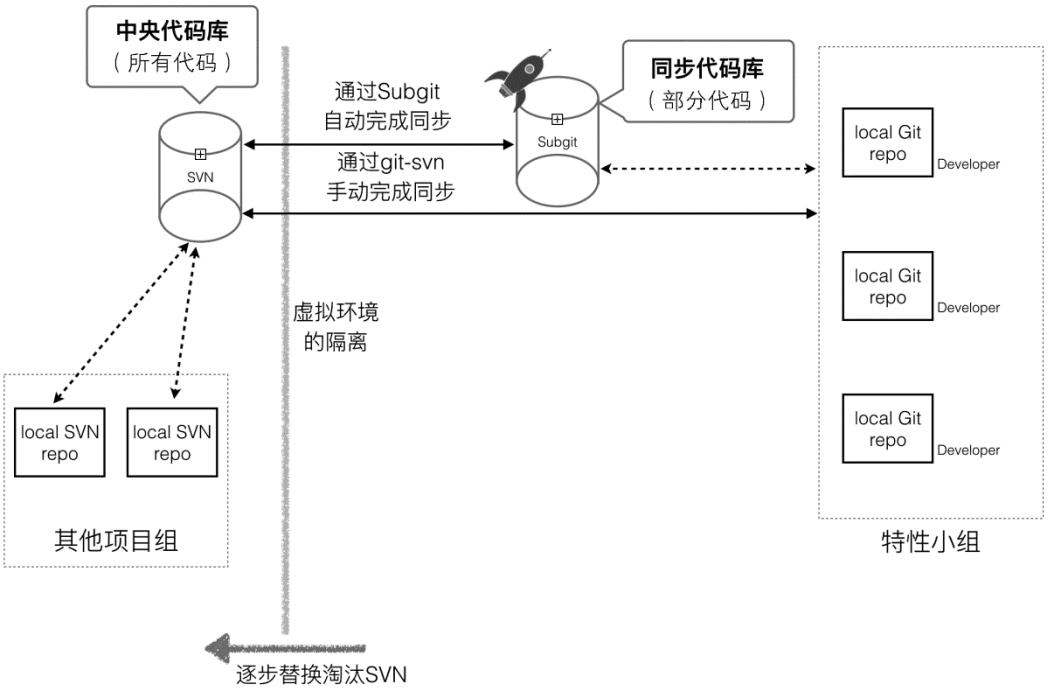

图 3-9　Subversion 迁移方案 1

1　https://git-scm.com/docs/git-svn

图 3-9 中包含了两种不同的方案，这两种方案各自有不同的优缺点，我们需要在深入了解这些方案后做出合理选择。

1. SubGit

SubGit 是一种比较好的选择，但是若大型团队的人数众多，其价格就会比较高昂。

首先收集所有需要迁移的人员的配置信息，包括人员的名字和邮箱地址，以及他们需要的代码目录信息，并使用这些信息配置 SubGit。

其次选择部分人员或者某个团队来尝试先迁移到 Git 客户端并通过 SubGit 进行代码同步。

最后，在这部分人员或团队没有使用障碍的时候再逐步推广到其他团队。在所有团队都逐步完成迁移后，就可以把中心端的 Subversion 代码库迁移到 Git 代码库中了。对于迁移的步骤这里不再赘述，与第 2 章中的迁移步骤类似，但仍然存在一些不同之处，其中最大的不同就是权限管理。由于默认的 Git 服务器基本上都不支持在同一个代码库中对不同的目录进行不同的权限管理，所以如果一定要对不同的目录进行不同的权限管理，则只有对新的 Git 代码库进行多库拆分，这部分工作比较复杂，因为涉及代码的目录结构，甚至需要对构建系统或者软件架构进行重构，所以决策前需要进行仔细的思考与验证。

2. git-svn

git-svn 是 Git 官方提供的迁移工具，简单、易用，是低成本迁移的首选。

首先，直接选择部分人员或者某个团队，安装标准的 Git 客户端，然后使用 git-svn 直接检测 Subversion 的代码到本地，git-svn 会将其转换成一个本地的 Git 代码库并保留 Subversion 服务器的信息。以后在程序需要同步代码或者上传代码时直接使用 git-svn 的命令就可以了。

其次，在这部分人或者团队成功使用 Git 本地库后，再逐步推广到其他项目组中。在全部人员都使用 Git 本地代码库后就可以开始服务器端的迁移了。

SubGit 与 git-svn 的最大区别就是 SubGit 用钱省去了工程师的一些工作，而 git-svn 增

加了工程师的工作成本但节省了投入成本。

对于一般的中大型团队而言，有上面的两种解决方案已经足够了，但是对于一些特定的大型分布式团队，如果团队的资源充足，并且希望将同一个类模块或者同一地区的团队一起直接迁移到 Git 下，则还可以使用 Gerrit/GitLab 等系统搭建一个独立的 Git 服务器，从而以分布式的方式进行代码迁移，如图 3-10 所示。

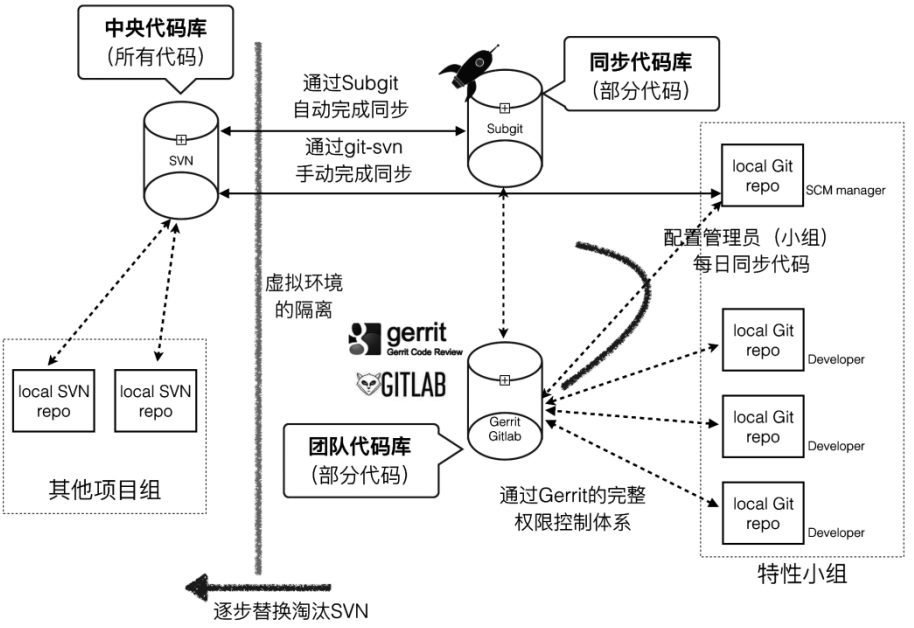

图 3-10　Subversion 迁移方案 2

实施步骤如下。

首先，按照迁移方案 1 的方式选择工具并开始搭建，与此同时安装并配置一台 Gerrit/GitLab 服务器。

其次，使用SubGit提供的Gerrit/GitLab插件 [1]来同步Subversion与Gerrit/GitLab之间的代码库。如果选用git-svn，则需要通过代码管理人员来同步Subversion与Gerrit/GitLab之间的代

1　https://subgit.com/gerrit.html 和 https://subgit.com/gitlab.html

码库。其实git-svn也可以将同步的步骤全部自动化，问题是后续会出现代码冲突并可能导致代码无法同步，所以需要配置代码管理人员，以解决代码冲突等问题。

前面的基础设施建好后，需要迁移的团队就可以直接从 Gerrit/GitLab 中进行代码检出和提交了。

这种方案的特点如下。

（1）隔离了团队开发人员与 Subversion 中心服务器的联系，让其完全感觉不到中心服务器的存在，不受其影响。

（2）可以基于 Gerrit/GitLab 方便地建立独立的 CI 系统，避免与主 CI 系统冲突。

（3）如果选用SubGit方案，则可以有效地解决SubGit的版权费用 [1]高的问题，因为仅使用一个或者几个统一的用户来通过SubGit与Subversion进行同步。

（4）可以对 Subversion 的代码库进行分库操作，以建立多代码库的方式同步到 Gerrit/GitLab 中，从而实现一定程度的代码权限管理，比如某些人或者某些小组只能访问特定的代码库。

当然，它也存在一些缺点：需要专门的代码管理人员来管理 Gerrit/Gitlab 与 Subversion 中心服务器之间的同步，以及管理 Gerrit/GitLab 服务器，管理成本较大。

上面两种经典方案来自于对项目的总结，但从 Subversion 到 Git 的迁移没有一种统一的最佳方案，因为不同的团队遇到的问题是不同的，所以需要的方案也是不同的。参考上面两种经典方案，不同的团队也许需要根据自己的实际情况进行一些定制，这样才能更有效地进行代码迁移。

1　SubGit10 个用户以下是免费的，而需要 25 个 Git 用户的时候就需要 990 欧元。参见 https://subgit.com/pricing.html

第 2 部分

当前与流行

第 4 章　分布式中大型团队

第 4 章

分布式中大型团队

4.1 分布式中大型团队的特点

随着软件开发团队规模的高速增长，有两个亟待解决的问题。

（1）开发成本不断增加。

（2）单一地区的人才数量无法满足增长的需求。

为了解决开发成本和人才数量的问题，我们尝试在不同的地区和城市招聘或者外包开发人员来组建分布式开发团队，从而降低人员成本和获得更多的人才来开发更为大型的软件系统。

对于软件开发来讲，分布式团队的存在带来了更多的问题，其中包括如何进行分布式敏捷协作，如何进行分布式代码开发和管理，如何进行分布式持续集成，如何进行分布式测试，等等。这些问题的本质就是组织架构的分离与集成管理，而解决这些问题的根本思路就是高效、快速、正确地集成和分离团队及团队的产出物，并通过化零为整的方式来形

成一个统一的开发团队。如何进行软件分布式代码管理是这众多问题中的一个基础问题，只有很好地解决了这个问题，才能很好地支持团队去解决其他问题，比如降低开发成本，提高产品的质量，加快产品开发速度及促进团队的协作。

4.2　分布式中大型团队在代码管理上的痛点与需求

对于分布式中大型团队，由于人数和团队众多，并且不同的团队分布在不同的地区，所以对于代码管理存在不少痛点，如下所述。

（1）互联网的不确定性。有时某个团队通过互联网无法连接中心代码库的服务器或者连接很慢，导致和代码中心服务器不在同一局域网的异地团队无法和代码中心服务器进行通信来完成代码更新、提交和日志查询等操作。

（2）无法一起审查代码。由于代码审查人员和代码提交人员可能不在同一个办公室，甚至相差多个时区，导致他们基本上很难在一起共同审查代码。

（3）团队交流和代码集成困难。由于不同的团队分布在不同的地区，交流困难，使得不同的团队之间很难对代码的细节进行高效沟通和集成，导致集成效率低下，错误频出，开发时间增加。

（4）很难找到备份代码中心仓库的时间段。由于在不同时区的团队随时都有可能提交代码，所以很难找到一个时间段来停机及备份代码。

为了解决以上痛点，很多团队提出了一些相应的要求，如下所述。

（1）能进行离线代码管理。为了解决互联网的不确定性带来的与中心代码库的连接问题，开发人员需要在不能和中心代码库通信的情况下在自己的开发机上对代码进行管理，或者在本团队内搭建一个独立的分布式代码库服务器来进行管理，然后在网络恢复时将本机或者本团队修改后的代码与中心代码库进行同步。代码管理系统需要支持本地代码管理，包括离线提交代码、离线查看代码日志、离线比较不同的代码版本、离线回滚代码等。

（2）能进行在线代码审查。为了让不同区域的员工在代码提交者离开办公室之后还可以审查代码，并把意见方便地记录下来反馈给提交者，且提交者在第 2 天上班时可以方便地阅读意见并对代码进行修改，代码管理系统中的代码审查功能需要记录所有的意见和操作作为日志，需要支持人工审查和自动审查，并支持持续集成。

（3）能对代码进行分布式权限管理。由于分布式团队中存在各个团队的地域和级别的区别，比如核心代码不能出境或者雇佣了一些外包团队，对于这种国外或者外包团队，只能公开特定的代码库给他们读取，并且能容易地控制他们的代码提交权限。所以代码管理系统需要能够对中心代码库方便地进行代码权限管理。

（4）能对代码进行分布式提交和集成。由于代码提交和集成是软件开发中最为重要的步骤之一，因此代码管理系统需要一个功能强大并且可以定制化的代码提交和集成功能，比如各个团队可以有自己独立的分布式代码库，并且这些分布式代码库可以与中心代码库进行同步和集成。

（5）能对代码库进行热备份。不同地区的分布式团队可能由于时区的不同，会全天 24 小时随时有可能与中心代码库进行代码同步。为了减少管理成本，不影响或者中断团队的开发和集成工作，代码管理系统需要支持对中心代码库和各个团队自己的分布式代码库进行不停机热备份。

（6）易用的分支管理及灵活的分支策略。由于很多情况下分布式团队中不同团队的地域、文化、工作方式及能力有所不同，所以他们可能会有独立的持续集成系统、独立的开发流程，以及灵活的分支策略。对于不同分支策略的需求，代码管理工具需要提供支持，比如每个团队可以自定义和实施不同的代码分支策略，中心团队可以集成或者进行统一管理。

4.2.1 离线代码管理

分布式代码管理系统很好地满足了这个需求。在分布式代码管理系统中提交代码一般有如下几个步骤。

（1）从远程代码仓库上拉取（pull）代码副本到开发者本地的开发环境中（包括所有历史提交记录）。

（2）检出（checkout）一个分支，在此分支代码的基础上进行开发、调试和测试。

（3）将修改提交（commit）到本地仓库的副本中。

（4）继续开发完成下一次提交。

（5）选择合适的时机，将代码更改推送（push）至远程代码仓库上。

在整个代码开发流程中，把代码拉取到本地之后，在新的提交被推送到远程仓库之前，开发者完全可以在本地访问到完整的历史记录，也可以持续进行代码提交。

如果在任意时间内网络故障造成远程代码仓库不能访问，则开发团队的每个开发者都可以在本地机器上进行代码提交。在网络故障解除并且远程代码仓库可以再次使用之后，开发者再将本地的提交推送到远程代码仓库。在这种情况下，要注意以下两点。

（1）在远程代码仓库恢复访问之后，要尽快把本地的代码更改推送到代码仓库中。

（2）在恢复访问的短时间内，开发者会集中推送代码，有极大可能会出现冲突，所以开发者上传代码前需要特别注意处理代码冲突。

借助分布式代码管理工具，我们还可以在不同的地域建立多个远程代码仓库，减少中央代码仓库的单点故障带来的全局影响，提供更健壮的服务。

4.2.2　在线代码审查

代码审查是指对开发人员的代码修改进行审阅和查看，在代码被构建成交付件发布之前找到代码中的问题，并反馈给提交者进行修改和改进。代码审查还给开发团队带来了非常多的好处。

◎ **提升产品质量**。发现因为粗心而导致的实现错误；找到更高质量的解决方案，比如精通算法的审查者会发现比原来算法更优的实现；具有丰富业务知识的审查者则会发现业务理解的偏差等。

◎ **统一代码风格**。使代码中的命名规范、代码的模块化结构更加一目了然，使开发者更容易理解代码，减少发生冲突的可能性。

◎ **提前获得反馈**。通过结对编程或走查的方式，在代码还未推送前就发现问题，尽早发现并解决 bug。对于一些特殊的软件产品（如 SDK），在正式发布之前，通过开放源代码和开放代码审查的方式，尽早收集使用者的反馈和建议，从而改进产品。

◎ **促进知识传递**。每行代码、每次提交，除了提交者，还有一个或多个审查者阅读和理解。审查者只有在充分理解代码的含义和前因后果后，才能给出评价和建议。这样审查者也了解了提交者的上下文和意图，也更容易在此基础上继续进行后续开发。

◎ **培养团队新人**。新人通过阅读和审查代码，或者阅读有经验的开发者给出的评价，来学习团队的代码风格及代码中的设计思想。团队的其他成员通过阅读新人的代码，可以了解新人的能力成长，根据新人的能力来给他们分配更合适的任务。

◎ **增强合作氛围**。有领导力的开发者会更主动地承担审查者和知识分享者的角色，带领团队共同成长。其他团队成员则自然形成追求“好评”的氛围，追求更高质量的代码匠艺。

越来越多的团队把代码审查和其他实践一起运用，来改进产品的交付。代码审查变成开发流程中不可或缺的一部分，**任何代码在没有通过代码审查的情况下，是不允许被提交到代码“主线”上的**。在持续进行的代码审查活动中，我们发现了一些重复的机械步骤。

◎ 通过执行编译和构建脚本发现编译期的错误。

◎ 通过自动化测试发现代码执行的逻辑错误。

◎ 通过静态代码扫描工具发现不符合规范的地方。

聪明的开发者使用构建脚本、单元测试框架、代码静态扫描来实现这些检查，通过持续集成服务不断地执行这些检查，从而将开发者从枯燥无味的机械劳动中释放出来，使得他们在审查过程中有更多的时间来发现更深层次的问题。

代码审查的实践要做到以下三点。

◎　审查要在代码被提交到“主线”之前进行。

◎　对提交“主线”的权限进行控制。

◎　要支持将代码的扫描和验证工具进行持续集成。

代码审查的形式很多，结对编程或者结对代码走查都是非常有效的方式。这些方法要求开发者坐在一起，在代码编写的过程中阅读代码及进行讨论，最终完成代码改进，在本地进行验证，达成一致后再提交到“主线”。但分布式开发团队因为地域和时区的差异，不具备“坐在一起”的物理条件，也产生了不同的审查实践。

早期一些开源团队采用邮件进行代码走查来克服时间和空间的差异。提交者把代码提交制成 patch 并将其作为邮件的附件发给维护者；维护者应用 patch 到本地代码上进行验证，在邮件中给出评价；提交者根据评价做出修改，重新把 patch 再次发送给维护者；反复几次之后，维护者觉得 patch 可以提交时，把代码 patch 提交到代码仓库。

当然，在邮件中阅读代码的体验非常差，比如没有代码上下文、评论文本和代码文本穿插在一起、多个邮件消息之间的代码和评论不连续，等等。于是以Gerrit为代表的在线代码审查工具应运而生 [1]。如图 4-1 所示是Google Android的Gerrit代码审查功能的首页 [2]，上面包含所有需要审查的代码提交，其中包括需要或正在进行审查的所有代码提交列表（open），已经通过审查并且已经合并的代码提交列表（Merged），以及所有没有通过审查并且已经被拒绝的代码提交列表（Abandoned）。

引入 Gerrit 作为代码审查工具之后，代码提交的流程如下。

Gerrit 把参与代码审查的角色（人或机器）分成以下三类。

◎　提交者。被审查的代码来自提交者，提交者会根据其他两个角色给出的反馈进行修改及提交，直到最终通过审查，才能正式提交代码。

1　https://en.wikipedia.org/wiki/List_of_tools_for_code_review

2　https://android-review.googlesource.com

◎ 验证者。对提交的代码进行验证，判断代码的正确性，一般由 CI 自动完成，包括构建、测试和静态代码扫描。

◎ 审查者。有更多领域的上下文，技术经验相对丰富，能对代码的逻辑正确性做出判断，能对代码的设计给出建议。

图 4-1 Gerrit审查提交列表 [1]

提交者从代码仓库中检出代码（master），在本地进行代码的编写与测试，完成提交后，使用下面这条命令推送代码（与普通推送不一样）：

```
git push origin HEAD:refs/for/master
```

通过这条命令推送的代码不会直接进入 Git 仓库中的任何用户创建的分支，而是形成一个变更出现在 Gerrit 审查界面上。实际上，在 Gerrit 中提交者无法直接向 refs/head/*推送任何代码提交，这避免了没有经过审查的代码影响任何开发分支。

Gerrit 收到提交者的变更之后，会发送通知给验证者和审查者，他们可以使用通知中的链接直接打开 Gerrit 中的代码审查界面，如图 4-2 所示。

1 https://android-review.googlesource.com

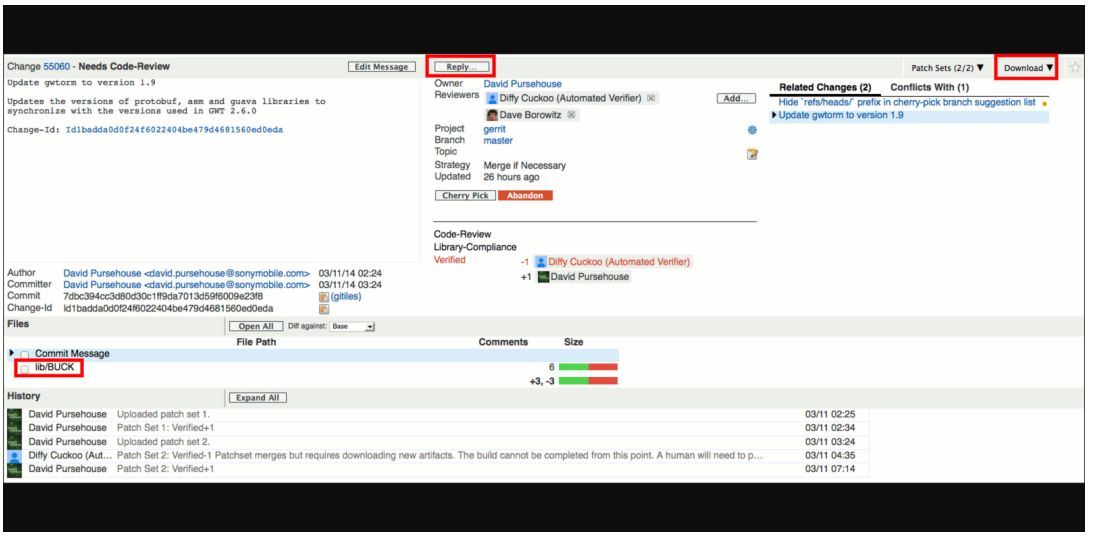

图 4-2　Gerrit 提交审查界面

验证者和审查者可以单击文件并打开代码对比界面来审查代码的提交，并可以对特定的代码行添加评语，然后单击“Reply”按钮对这个代码提交添加评语并就验证（Verified）和审查（Code-Review）进行打分，如图 4-3 所示。

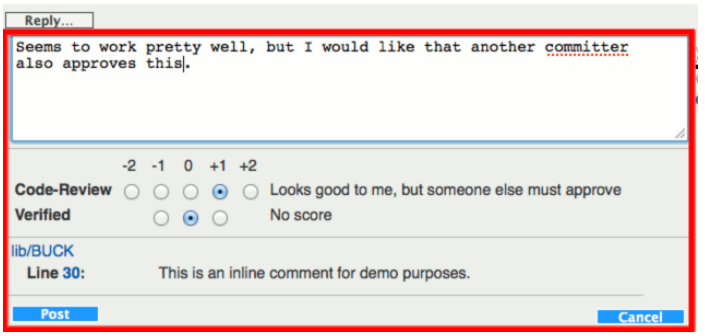

图 4-3　Gerrit 提交评分和意见

还可以单击“Downalod”按钮后选择一种适合自己的方式下载代码到本地，通过本地构建和测试来对代码进行审查，如图 4-4 所示。

一般的审查人员可以在通过审查后对代码提交加 1 分，如果没有通过，则减 1 分。有高级权限的审查者可以直接加 2 分通过或者减 2 分拒绝提交。

图 4-4　Gerrit 下载提交

同时，可以配置自动触发持续集成服务器（比如 Jenkins），让其自动获取 merge 提交后的代码库来自动构建和测试，如果通过持续集成的构建和测试，则自动在审查结果上加 1 分。

最后当审查结果为 2 分时，拥有代码合并权限的审查者就可以直接单击“Submit”按钮完成代码提交到主干的合并。

Gerrit代码审查的更多细节请参考其官方文档 [1]。

除了这些独立部署的在线代码审查工具，还有很多基于云的代码托管服务如 GitHub、GitLab、Bitbucket 等，也把代码审查作为服务的一部分。它们都使用一种 Pull Request 或者类似的实践来进行代码审查，关于具体的 Pull Request 的流程，请参考第 6 章中关于 GitHub 的介绍。

4.2.3　对代码进行分布式权限管理

很多传统的中心端代码管理服务器只提供了基于个人的中心端代码权限管理。这种权限管理的方式相对简单，易于配置与实施，但是这种方式不太适合大型分布式团队中的代码权限管理，因为这种方式不能很好地对分布式代码管理服务器进行管理，而且对于分布式团队中不同级别和区域的成员很难进行差异化管理。所以在分布式团队中进行代码管理需要代码管理服务器提供灵活的配置和种类多样的权限管理机制和功能，其要求如下所述。

◎　能对每个用户进行权限管理。

1　https://gerrit-review.googlesource.com/Documentation/user-review-ui.html

◎　能提供权限组的方式对同一类用户进行权限管理。

◎　能提供多种权限认证方式，比如 SSH、LDAP 等。

◎　能对多个不同的代码库和分支设置不同的权限。

◎　能提供服务器端代码合并功能并将其用于不同权限用户之间的协作，其中包括同一个分支内的合并、不同分支之间的合并及库与库之间的合并。

现在基本上所有商用和大部分开发免费的集成代码管理服务器都支持基于个人、权限组和多重权限认证，以及对多库的权限设置，但是很少会提供服务器端代码合并功能。目前提供服务器端代码合并功能的都是第三代分布式代码管理服务器系统，比如 GitLab、Gerrit 及 GitHub 等。

正是由于提供了这样的功能，才使得分布式代码管理系统更加适应分布式团队的分布式软件开发，因为它解决了分布式团队中最重要的一个问题：分布式协作开发中的权限问题，即不同的团队或者个人之间由于时间或者地域等问题无法协作开发并提交代码，以及如何在有代码提交并审查后控制代码合并或者回滚等。

4.2.3.1　代码管理工作流模型与权限管理

权限管理与工作流模型密切相关，因为权限管理模型往往体现在工作流模型里，所以通过工作流模型来理解和管理权限更为容易。下面是几个常见的代码管理工作流模型。

1. 唯一中心式模型

唯一中心式模型是最为常见的模型，如图 4-5 所示，其对应的权限管理也最为简单，只要在中心端针对每一个独立的开发者设置权限就可以了。很多中小型团队在只有一个中心端服务器的项目时都使用这种模型，少量大型分布式团队在只有一个中心端服务器的项目时也可以使用这个模型，但可能会带来一些挑战，比如提交管理时比较混乱，没有强制的标准化审核流程导致审核混乱以至于权限管理混乱。对于小型项目，在团队人少的情况下没有强制的标准化审核流程，出错的概率一般不大，修复成本也较低。但是对于中大型项目分布式团队而言，代码提交人员较多，提交频率很高，人员情况复杂，比如技术水平、工程意识等参差不齐等，出现错误的概率就会大大增加。所以建议中大型分布式团队使用

集成经理式模型和多项目分布式模型来应对这些挑战。当然这两种模型也不完美，也存在其他方面的问题。

唯一中心式模型如图 4-5 所示。

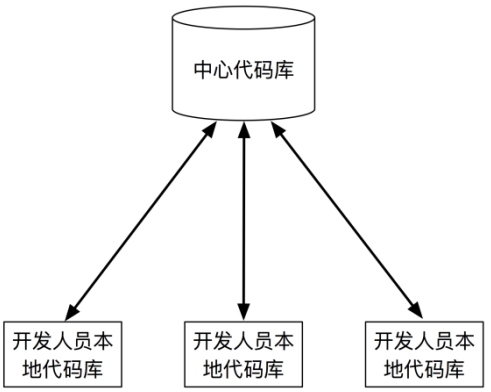

图 4-5　唯一中心式模型

2. 集成经理式模型

为了解决唯一中心式模型中提交管理可能出现的混乱问题，并应对没有强制的标准化审核流程带来的挑战，集成经理式模型应运而生。这个模型的主要特点就是开发人员在将代码提交到中心库之前，必须将代码提交给一个专门的集成经理（可以是一个人，也可以是一个小组）去审查，审查通过后才能由集成经理将代码提交到中心代码库，开发人员不允许对服务器端的中心代码库直接进行代码提交，只允许从中心代码库中同步代码。通过这样的强制化标准流程，团队的开发人员就不能随便将自己的代码提交到中心代码库，从而可以解决唯一中心式模型存在的提交混乱问题。但是集成经理式模型也存在对于集成经理的依赖问题，可能造成集成经理的工作量巨大。所以在真实的项目中集成经理只是一个角色的名称，代表团队中经验丰富、能力出众且足以审查其他成员所提交代码的一组人。

对于这种模型，在真实的项目里存在着三种稍微不同的实践。

实践一如图 4-6 所示。这种实践在开源社区和商业公司中最为常见，GitHub 和 GitLab 也默认支持并推荐这种模型。其步骤如下。

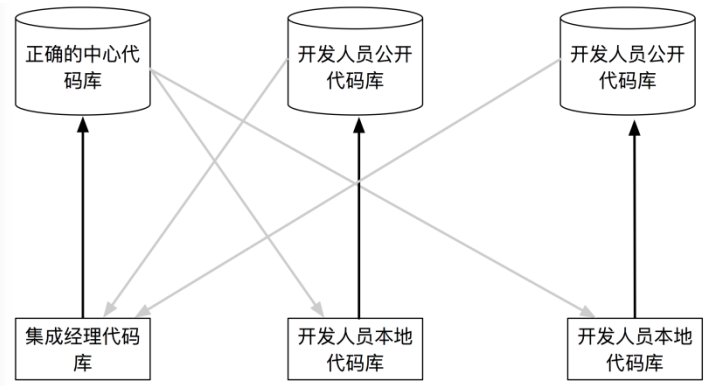

图 4-6　集成经理式模型一

第 1 步，开发人员从中心代码库中将代码同步到本地，并创建一个自己的分支。

第 2 步，开发人员将自己的分支上传到服务器端，从而在服务器端创建一个自己的分支。为了节省管理配置成本，一般情况下服务器端不会对这个由这个开发人员创建的分支设置任何权限。这种方式仅针对于部署在企业内部的代码管理系统（注意，如果代码管理系统部署在互联网上，则还是建议针对这些分支设置权限管理）。

第 3 步，在开发人员编写完代码后将代码提交到本地代码库中，再将代码推送到服务器端自己的公开代码库分支中。

第 4 步，通过服务器端代码管理系统的代码审查功能，将所有需要集成到中心端代码库的修改提交给集成经理，在集成经理对这些提交的代码审查通过后由其将代码合并入中心端代码库。

现实中 GitHub 的 Pull Request 和 GitLab 的 Merge Request 等都是这种模型里最为经典和常用的代码提交和审查的方式，它们有效地解决了开发人员与中心代码库之间的写权限管理问题。

但是使用这种模型需要开发人员对中心代码库有创建分支等分支相关的权限，这会导致中心代码库仍然存在一定的权限管理成本和风险，比如错误配置分支的权限及操作了其他人的分支。为了解决这个问题，实践二出现了。

实践二如图 4-7 所示。

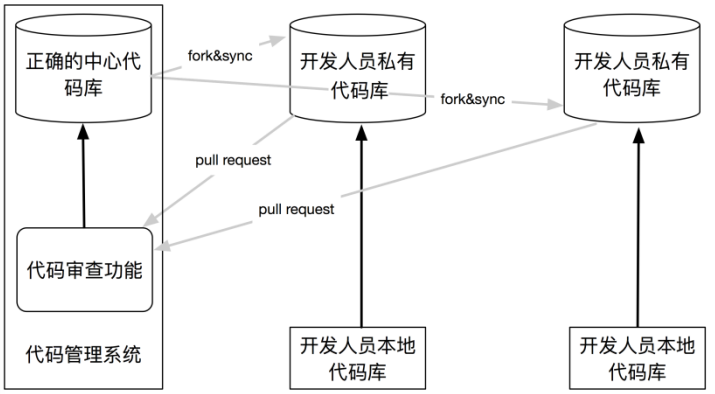

图 4-7　集成经理式模型二

这种实践在开源社区里较为常见，GitHub 也默认支持并推荐这种模型。其步骤如下。

第 1 步，开发人员直接通过代码管理系统中的复制功能将中心端代码库中的主分支复制一份到一个独立代码库中，这样开发人员针对这个复制的独立代码库拥有所有管理权限，包括设置权限、创建分支等。比如GitHub中的代码库复刻（fork[1]）技术。

第 2 步，开发人员从自己创建的这个独立代码库中获取代码进行开发，但是需要通过特定的技术和方法将中心端的代码库同步到自己本地的代码库中，比如在GitHub中可以对代码库复刻后的代码库进行同步 [2]。

第 3 步，开发人员在编写完代码之后将代码提交到本地代码库中，再将代码推送到自己的独立代码库中。

第 4 步，通过服务器端代码管理系统的代码审查功能将所有需要集成到中心端代码库的修改提交给集成经理，代码集成经理在对这些提交的代码审查通过之后将其修改合并到中心端代码库。

1　https://help.github.com/articles/fork-a-repo/

2　https://help.github.com/articles/syncing-a-fork/

这种实践的优势就是减少了中心端代码管理系统中对于每个单独的开发人员的权限配置管理工作，基本上完全隔离了开发人员和中心端代码库之间除读外的所有权限，减少了管理风险。但面临的挑战是开发人员需要自己维护在管理系统中的独立代码库，增加了开发人员的工作成本，而且开发人员需要学习更多的代码管理系统的技能和知识才能完成管理工作，所以也增加了开发人员的学习成本。如果团队里有大量的开发人员对代码管理的经验并不丰富，或者学习意愿不高，则不建议使用这种实践方法，这时实践三就是一个不错的选择。

实践三如图 4-8 所示。

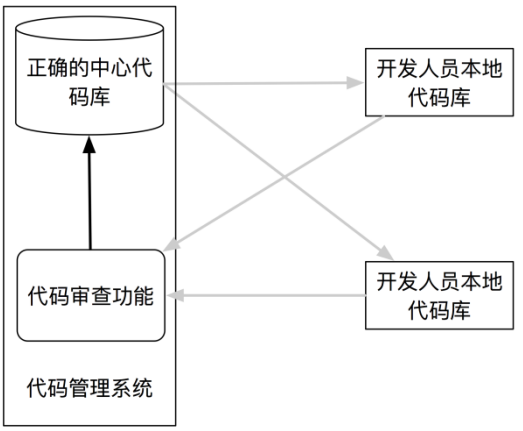

图 4-8　集成经理式模型三

实践三在特定的商业公司和大型开源项目里较为常见，比如 Google 的 Android 项目及 OpenStack 项目都是使用这种实践方法。其步骤如下。

第 1 步，开发人员从中心端代码库中直接同步代码到本地。

第 2 步，开发人员在编写完代码之后将代码提交到本地代码库中，再将代码推送到中心端代码服务器，代码管理系统会自动创建一个代码审查记录并通知代码审查人员进行审查。在代码审查人员对这些代码审查通过后由其将代码合并到中心端代码库。

实践三其实结合了分布式代码管理和中心端代码管理两种方式的特点。使用中心端代码管理的理念隔离了分布式代码管理的复杂度，让开发人员能使用传统的基于中心端代码

管理的方式进行工作，从而减少了工作和学习的成本。但是这种实践也有不足之处，那就是相对于实践二，它把开发人员的管理和学习成本转移到了服务器端，增加了代码服务器端的管理成本。

这三种集成经理式模型的实践没有一个是完美的银弹，各自有优缺点，所以要根据团队各自的特点和需求进行选择，比如技术能力和学习能力强的团队则可以选择实践一和实践二；如果想让团队人员工作起来更为容易，则可以选择实践三；而如果希望减少中心端服务器的管理成本，则可以选择实践二。

3. 多项目分布式模型

如果一个大型分布式项目拥有大量相对独立的子系统或者子模块，而这些子系统和子模块都有相应的团队或者负责人来维护，那么就需要针对这些子模块和子系统设置独立的分布式代码库，并且需要有独立的团队或者负责人来对这些独立的代码库进行维护，比如设置权限、审查代码及将审查后的代码合并到中心代码库等。这个模型有效地隔离了子系统和子模块之间的权限，将大量的与权限相关的工作转移给了各个下级团队，比如 Linux Kernel 项目就是用这种模型进行代码权限管理的。

这种模型的经典流程如图 4-9 所示。

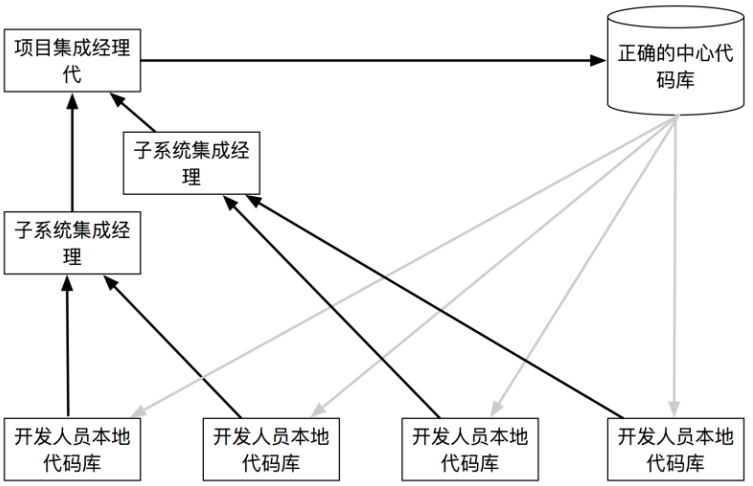

图 4-9　多项目分布式模型

这个模型的经典实践如下。

第 1 步，中心端代码库给所有项目开发人员只读权限，然后每个子系统的负责人自己创建一个独立的分支或者代码库，并给团队成员配置代码访问权限。

第 2 步，开发人员从中心端代码库中同步代码到本地代码库并进行开发。

第 3 步，开发人员在编写完代码后将代码提交到本地代码库中。

第 4 步，通过服务器端代码管理系统的代码审查功能，将所有需要集成到子系统代码的修改提交给子系统代码审查人员，在审查人员对这些代码审查通过后由其将代码合并到对应的子系统的代码分支或者独立代码仓库中。

第 5 步，由一名专职的中心端代码库管理人员主动地定时从子系统分支或者代码库中拉取代码，审查后再合并到中心端代码库。

这只是一个经典实践过程，现实中不同的团队根据自身项目的限制和需求，需要对其中某些步骤进行修改和定制化，比如第 5 步中中心端代码库管理人员主动地定时去子系统中同步代码的这个步骤，有可能改为由子系统管理人员主动推送代码给中心端代码库管理人员去审查等。其缺点就是整个流程较为复杂，但为了更好地管理拥有大量子系统的分布式系统，包括处理相应的代码权限和代码提交流程等问题，这样的成本是值得的。Gerrit、GitHub 和 GitLab 等代码管理系统都可以实现这种模型。

4.2.4　对代码进行分布式提交和集成

企业发展到一定规模的时候，其业务或者内部的 IT 支撑系统会越来越复杂，相应的产品或项目的代码也会随之变得越来越复杂。比如企业的业务可能扩展到多个领域，会建立不同的团队来开发相应的产品，而这些产品在开发演进过程中可能逐渐呈现出某些共性，团队成员可能会将它们提炼出来，形成企业内部的工具框架或者 SDK 来复用。在这些框架的基础上，又会有更新的产品和项目被开发出来。随着团队的扩张，企业的开发人员和研发中心可能会分布在全国乃至全球各地。对于大型企业来说，代码管理系统及相应的实践需要能够支撑这样的工作场景，这就要求我们采取分仓库来管理代码，提高集成模块化代码的能力，以及相应的代码仓库的分布式管理能力。

4.3 代码仓库拆分与集成

每个软件产品都是一样的，只要能给使用它的用户带来便利且带来价值，就会一直发展演进下去。对一名在下午六点饥肠辘辘的程序员来说，眼前的编辑器、正在编写的代码中的程序框架、刚刚用来下单叫外卖的 App 都是能为其带来价值的软件产品，而其正在创造的一个线上商城又何尝不是呢？在和这些软件产品的交互中，我们也感受到它们随着时间推进而发生的变化：编辑器自带的插件越来越多；新的程序框架又增加了好多酷炫功能；外卖 App 升级了，可以买药了。它们直接或者间接地提醒项目成员这样一个事实：软件产品在不断地迭代更新，会带来不断变化（绝大部分是增加）的代码和不断增加的历史记录。不断增加的代码量和复杂度逐步挑战着开发团队的认知，开发团队则不断地思考怎样应对这高复杂度的挑战：新的开发框架被引入来提升效率、单模块系统被拆分成多组件来分解复杂度。但现实是残酷的，产品要应对的市场变化得更快；开发团队疲于应付代码和复杂度的快速膨胀，直到到达极限无法承受，最终不得不在架构演进的同时，让新开发者加入，把复杂的系统拆分成不同的模块交给新的团队来处理，问题会暂时得到缓解。增加人手却是把双刃剑，不断增加的开发者同时会催化代码复杂度的膨胀（比如复制、粘贴代码），最初的雪中送炭慢慢变成了火上浇油。

让我们一起来看看两个著名开源项目的真实情况，体会代码量和代码复杂度在较长一段时间内的持续增长。第 1 个开源项目是 Java 服务器应用开发者耳熟能详的 Spring 框架；第 2 个开源项目是 Chrome，它是前端开发者每天都会用到的浏览器。它们在最近数年的代码仓库的变化趋势如表 4-1 所示。

另外，可以在Open Hub上找到这两个项目关于代码的其他数据：Spring-Framework[1]和Chrome[2]。

1 https://www.openhub.net/p/spring-framework

2 https://www.openhub.net/p/chrome

表 4-1　Chrome 与 Spring Framework 代码仓库的变化趋势

项　　目	2010 年 2 月的代码行数	2016 年 6 月的代码行数	2010 年 2 月的提交数	2016 年 6 月的提交数	贡献者总数
spring-framework	434 337	648 604	144	206	187
Chromium	2 681 480	14 482 117	2952	6373	5448

一个产品到底都有多少行代码？项目团队每天有多少次提交？如果对这些问题没有准确的答案，则可以对比自己的产品和这两个开源产品的规模，做个大概的估算。请相信，这些数字只高不低，很少有项目能做到像这些知名开源项目一样的自律和整洁。

现在可以计算一下：有一个 100 人的开发团队，每人每天提交两次代码（把本地工作空间中的代码推送到服务器），每次代码提交都会触发流水线执行 1 次（需要增量地检出代码库中的代码）；假设每次检出需要 30 秒，提交需要 1 分钟，这样的消耗加起来达到了每天 5 小时，如果再算上流水线因为失败而出现的反复提交次数、每次提交代码前花在合并上的时间，以及其他诸如数据统计、代码浏览等一些只读操作，则耗费的时间会更多。如果不幸选择了笨重的管理工具或者分支策略，则会造成更多的浪费甚至成为开发活动的瓶颈。如果团队变得更多呢？提交更多呢？文件更多呢？

笔者曾在某个产品开发团队看到这样的情况：开发者早上打开 IDE，第 1 次拉取代码之前的登录操作就反复尝试了数次，花了接近 10 分钟；本地修改构建脚本验证之后，想让 CI 也能被触发进行验证，这时提交单个修改文件也要花上几分钟，每次提交代码都变成了痛苦的煎熬。本身浪费的时间暂且不说，开发者本能的反应就是要避开这个痛苦的操作，不知不觉减少提交的频率。减少提交的频率意味着本地的代码没有及时集成，问题暴露和修复的时间变长，反馈变慢，不利于实现快速交付软件的价值并进行验证。同时，每个开发者本地的未提交代码积攒得越来越多，每个开发者的代码之间的差异也越来越大，在一段时间后再集中提交时，解决冲突的痛苦会被放大数倍，形成恶性循环。那么如何应对这些困难呢？首先需要看看软件架构是如何应对复杂度的。

如果一个项目的业务逻辑复杂，则我们最先想到的就是引入适合自己的软件架构来分而治之。无论产品采用什么风格的架构，都一定可以拆分成不同的模块。各模块按照它们

之间的特定关系集成在一起，形成了最终交付的产品（这种模块间的关系在软件架构中被称为依赖）。模块化几乎是所有软件产品开发中的必然实践，小到代码类、文件和包的划分，大到前后端代码的分离、微服务的拆分，等等。因此几乎所有项目在建立伊始就已经包含了模块划分和模块之间的依赖管理机制；重用已经造好的模块“轮子”（第三方库）和工具来提高效率。很少有产品会重走一遍从单体到模块的拆分过程，因为所有的编程语言、框架和工具都一定具备模块化和依赖管理的能力，而产品从一开始选择语言框架或工具时就确定了模块化的具体实施方法，在开发过程中依照规范实施即可。稍后笔者将介绍各种常见的依赖管理机制。

在项目使用一种特定风格的软件架构把产品拆分成不同的组件后，原本一个代码仓库就具备了被拆分的可能性。如果单一代码仓库的体量达到了影响工作效率的地步，就需要把不同的组件拆分到不同的独立仓库中，并利用当前技术栈的模块化和依赖管理机制在交付前把各个模块集成起来。

模块化并不意味着每个模块都必须使用独立仓库存放代码，把整个公司的代码放在同一个仓库中和模块化并不冲突。而且把所有代码放在同一个仓库中有如下好处。

（1）代码重用。一旦发现重复的代码，就可以立即提取出来；提取之后就可以在自己的代码中直接用到。

（2）代码重构。像重命名这样的重构操作可以在所有用到的地方立即得到更新，不会有遗漏。

（3）统一的工具链和规范。所有开发团队使用同一套工具可便于沟通理解，形成一致的规范。

同时，代码开放与全员贡献的团队文化更是给工程师创造了自由的环境。以 Google 和 Facebook 为代表的大型互联网公司会投入大量的资源来推行这种管理方式，如下所述。

（1）基础设施。有足够多的仓库镜像甚至数据中心来加速各地的访问速度。Google 就在全球多个数据中心有多个代码仓库镜像。

（2）客户端。常见的如 Subversion/Git 等工具无法支撑如此体量的代码仓库，需要基

于一些代码管理工具做二次开发。Google 和 Facebook 分别基于 Perforce 和 Mercurial 开发了内部使用的工具。

（3）持续集成。持续集成服务要能够灵活地配置、构建不同的组件或组件集合。同时，构建工具也要支持增量构建和构建缓存来加快构建速度。Google 和 Facebook 基础设施的能力不在话下，它们也分别开发了 Bazel 和 Buck 这两种构建工具来解决大型多技术栈混合项目工程的效率问题（稍后将简单介绍这两种工具的优势）。

（4）代码浏览审查。有专门的在线应用浏览、检索所有的代码，也有专门的在线系统对所有的提交进行审查和讨论，所有这些系统都是对所有开发者开放的。

不是所有团队都有资源能做到这样的管理方式，也没有必要追求如此的极致，而且多代码仓库和单一代码仓库各有优势，大部分企业和团队会选择拆分代码仓库来规避日益增加的复杂度。

4.3.1　优化单代码仓库

对于拆分的选择必须十分慎重，有些问题并不需要通过拆分代码仓库来解决，有更好的处理方法。在决定拆分代码仓库之前应该首先尝试这些解决方案，有可能就不再需要拆分代码仓库，即使拆分不可避免，也可能在拆分之后遇到同样的问题，这时也可以考虑这些解决方案。下面分别介绍几种常见问题的应对方式。

4.3.1.1　如何处理“大”文件

第 1 种会严重影响代码管理工具的效率的问题是：工程文件中包含了大量非文本文件资源，例如多媒体资源、二进制包、文档、虚拟机镜像，等等。它们和源代码文件相比体积巨大，小的有几百 KB，大的甚至有几个 GB。如果它们被放在代码仓库中，则首先会严重降低代码检入、检出及提交的速度；其次，我们不能将它们直观地进行对比（需要专门的对比工具），出现冲突之后解决起来很困难且耗费时间；最后，这些资源往往由非技术背景的团队成员来开发和维护，比如设计师、业务分析师，等等，代码管理工具对他们来说实在算不上友好（别说使用命令行工具，就算给他们解释清楚分支提交冲突等概念也是

一件困难的事情）。代码管理工具多被用于处理文本文件，并不适合管理二进制文件，仓库中包含的二进制文件应该越少越好。我们把这些二进制大文件进一步分为三类，对每类文件的正确处理方式也有所不同。

第 1 类是和产品完全不相干的资源或文档，它们根本就不应该出现在项目的代码仓库中。这种类型的常见资源有 PSD 文件（并不会被直接用到，被用到的是它导出的图片文件）、Microsoft Word 编写的项目需求文档（它们的内容应该早已被拆分成用户故事、测试用例等，其中属于代码的部分存在仓库中，而其余部分应该存放在项目管理系统中，比如 Jira 或者项目 Wiki 中）。如果需要类似代码管理工具的历史记录功能，则也有其他服务和工具更有针对性地完成。例如，Adobe 的专门存放资源的服务 Creative Cloud 及像 Dropbox 这样的网络存储服务都可以满足需要。如果出于对安全的考虑，这些资源不能被放到企业外的互联网上，那么内部的文件服务（比如 FTP）会是比代码仓库更好的选择。当然，需要为这些资源设计一些备份的目录结构和策略，使用脚本实现一些定制化的工具来弥补缺失的历史记录功能。

第 2 类是二进制格式的库和工具，例如动态链接库文件或者 Jar 包、构建用到的可执行程序或者 SDK、部署用的镜像，等等。它们可能是由代码生成的中间交付件，也有可能是从其他地方复制过来的，但它们都必须作为产品最终交付的一部分。如果是在代码编译过程中生成的二进制文件，则可以借助编译工具提供的缓存功能，缓存在开发者本地的工作环境中。只要代码没有变化，在构建的过程中就可以优先使用本地缓存中“命中”的二进制文件。这种缓存没有变化的中间构建结果的功能叫作增量编译，现在基本上被所有的编译工具支持。这些二进制文件如果没有被存放在代码仓库中，就会在第 1 次检出代码或者代码有变化时被重新编译生成二进制文件，这种较低频率的构建所耗费的时间也是可以接受的。而对于第三方的二进制文件，把它们放在代码仓库中除了影响性能，还缺少对这些文件的版本规范管理。和其他类型的大文件一样，它们也有更适合的特殊文件服务来管理；稍后会介绍二进制文件的依赖管理机制。

第 3 类是项目中必不可少的二进制文件，它们必须存在，是整个软件不可分割的一部分，比如项目必须使用的资源文件（图片及各种格式的音视频文件等）。有一些代码管理工具对这种文件类型的支持不算好。例如，早期的Git版本对这些大文件并不友好，需要

借助一些工具来解决这种问题。最新的解决这种问题的方法就是使用Git LFS（Large File Storage）[1]，它把这些类型的文件单独存放到了另外的存储空间中，而在代码仓库中只记录了它们的引用。目前主流的代码仓库托管服务GitHub、BitBucket、Coding和用于自建代码仓库服务器的GitLab、Gerrit（通过插件）等都已支持Git LFS。

1. Git LFS 的使用

Git LFS 的使用非常简单。首先要在开发机器上安装必需的软件：一是支持 LFS 的 Git 客户端（版本 1.8.5 以上）；二是 Git LFS（可以在其官网上找到不同操作系统的下载链接）。安装完软件之后别忘了一定要执行命令：

```
$ git install lfs
```

这条命令会开启对 Git LFS 的支持，必须执行但只用执行一次。安装、配置之后就可以配置、使用 Git LFS 项目了。假设需要把项目中的所有 PNG 格式的文件使用 LFS 来存储，则应进入到项目的文件目录中，执行如下命令：

```
# 项目中用到的所有.png 文件都使用 LFS 来存储
$ git lfs track "*.png"
# 把.gitattributes 添加到代码仓库，这样其他人拉取代码之后才可以同样使用 Git LFS 拉取到.png 文件
$ git add .gitattributes
```

接下来就和正常使用 Git 没什么区别了，把想提交的文件提交到代码仓库并推送就可以了。

通过借助这样的工具，可以把代码中的大部分占据磁盘和带宽资源的超大文件转移到代码仓库外，剩下的大文件即使留在仓库内也可以利用代码管理软件的高级功能，避免影响日常使用的效率。对于大文件的处理原则就是：无关文件不要放在仓库中。如果源文件尺寸太大，就想办法把它放在仓库外，但必须利用一种机制记录对这些文件的引用，并能在拉取或检出代码时取得这些文件。那么，现在有一个问题：一个移动应用构建产生的安

1　https://git-lfs.github.com/

装包应该被放在代码仓库中吗？相信你已经有答案了。

如何移除已经提交的大文件

可以使用filter-branch命令来把错误地提交到仓库中的文件从历史记录中删除。filter-branch命令号称Git的“核弹级”功能，它可以批量地修改大量的提交记录。Git的官方手册中介绍了这条命令的多种不同用法[1]，其中就包括如何从历史提交记录中删除一个文件。

4.3.1.2 超大工程检出

即使我们把大文件移出了代码仓库，单仓库中的代码随着时间的推移，其文件一定会越来越多，而提交的历史记录一定会越来越多。由于分布式代码管理工具默认复制整个代码仓库，而检出时是切换整个工作空间的所有文件树，过多的文件和历史记录也会影响我们复制检出、查看代码的速度，影响日常工作的效率。那么有没有办法解决这种问题呢？

首先，如果开发者还没有使用固态硬盘，就可以先换上固态硬盘。固态硬盘的随机读写性能远远超过传统机械硬盘这方面的性能，使用后整个开发工作的体验会变得完全不一样。现在，固态硬盘的成本劣势与它带来的效率提升相比完全可以忽略不计。Joel Spolsky[2]在 2009 年的一篇博客 [3]中就描述了固态硬盘给程序员带来的效率提升，而现在笔者在国内许多团队中仍然看到无数开发者在机械硬盘上浪费时间。如果要提升开发者的效率，则这绝对是一笔首先要考虑的回报最丰厚的投资。

回到产品代码仓库上，开发者当前的工作基本上限定在最近几天的历史记录和某个或某几个目录的代码中。他们不需要关心去年的提交记录，也不需要改动其他不相关的目录中的代码。我们可以利用代码管理工具的“浅”（Shallow）复制和“疏”（Sparse）检出功能，来优化本地代码仓库的体积和操作的效率。下面我们以 Git 为例来说明这两种方法。

所谓“浅”复制，是指只检出仓库中特定范围的提交记录。例如，可以使用下面的命

1 https://git-scm.com/book/zh/v1/Git-工具-重写历史

2 《软件随想录》作者，Stackoverflow 与 Trello 创始人

3 https://www.joelonsoftware.com/2009/03/27/solid-state-disks/x

令检出 spring-framework 仓库中最新的一次提交：

```
$ git clone --depth=1 git@github.com:spring-projects/spring-framework.git
```

仅检出当前最新的提交对持续集成服务来说相当重要。持续集成服务关注的就是某个分支上最新的一次提交，对它进行构建和验证可更快地返回结果。

持续集成服务 Git Clone 的默认行为

常见的持续集成服务几乎都使用了“浅”复制的规则，在构建时只检出部分提交记录，来减少代码检出对整个构建过程的影响。比如，Travis的默认设置是只检出最近最多 50 次的提交记录，也可以在配置 [1]中把这个次数限制设置得更小。而Jenkins的Git插件在 2012 年的 1.1.23 版本中已经支持了“浅”复制 [2]，在这篇博客 [3]里提到了更多的Jenkins Git 插件的其他高级设置。

在搭建自己的持续集成环境时，如果决定不使用持续集成服务提供的这些默认的插件或配置，而是自己写脚本来检出代码，则请仔细决定检出代码时使用的参数，避免不必要的浪费。

在 Git 1.9 版本之前，“浅”复制的功能有非常大的限制：“浅”复制到本地的代码仓库基本上只能当作一个只读的库来浏览代码，不能执行 git fetch，也不能执行 git push。而 Git 1.9 版本解除了这些限制，现在可以在“浅”复制的仓库上进行 git pull 和 git push 操作了，但需要注意：必须保证运行 git pull 和 git push 时提交记录的范围是足够的。

下面这条命令可以检出从 2017 年 6 月开始的整个 spring-framework 中的提交，假设要在最新的 spring-framework 源码基础上做开发，则可以这样做。

```
$ git clone --shallow-since=2017-6
git@github.com:spring-projects/spring-framework.git
```

1　https://docs.travis-ci.com/user/customizing-the-build#Git-Clone-Depth

2　https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

3　https://www.cloudbees.com/blog/advanced-git-jenkins

如表 4-2 所示是 2017 年 6 月 25 日对这两条命令做的测试数据（很慢），由此可以看出“浅”复制的性能优势明显。

表 4-2 “浅”复制与“疏”检出的性能比较

对 比 项	--depth=1	--shallow-since=2017-6	完 整 克 隆[1]
速度	8KB 每秒	20KB 每秒	约 10KB 每秒
耗时	17 分 18 秒	13 分 21 秒	大于 1 小时 53 分 54 秒
尺寸	13.23MB	13.64MB	大于 36.08 MB
对象的数量	9541	13477	373924

所谓“疏”检出，是指只检出整个代码仓库文件中的部分目录。和 SVN 可以单独检出每个子目录不一样，Git 的“疏”检出依然需要先复制整个仓库，只是在本地检出时指定需要检出到工作空间中的子目录。除了第 1 次的复制耗费时间，剩下的检出操作速度可以得到提升。可以通过如下命令实现（还是以 spring-framework 为例）：

```
# 创建一个新的空仓库
$ cd spring-framework
$ git init
# 设置远程仓库
$ git remote add -f origin
git@github.com:spring-projects/spring-framework.git
# 开启“疏”检出功能的支持
$ git config core.sparsecheckout true
# 设置需要“疏”检出的目录
$ echo "spring-orm" >> .git/info/sparse-checkout
# 现在检出的文件中就只有 spring-orm 这个目录了
$ git pull origin master
```

1 在接近两小时的等待之后，完整复制的测试不得不终止，但测试的结果显而易见

此外还有微软最新开源的 GVFS（Git Virtual File System）也可以提高本地代码仓库操作的效率。首先，它在底层使用虚拟文件系统，只有在文件被第一次用到时才会下载真正的文件到磁盘上；其次，它使用“疏”检出的方式只检出必要的对象，保证代码仓库可以正常地进行开发；而剩下的对象只会在需要时才下载。这样它就可以支持超大型的单体代码仓库，这一切对开发者而言都是透明的。目前，GVFS 还在开发阶段中，但是我们已经可以根据其主页中的指引进行使用了。

无论使用哪种代码管理工具，都应该先仔细阅读其文档，找到它们支持的“浅”或者“疏”检出的特性，利用这些特性只检出部分历史记录或子目录，来提升日常工作的效率。除了本身的读写速度，还有另外一些工作也会因为大量的代码文件而变得特别缓慢，其中最频繁的工作就属构建了，例如每次提交代码前需要本地构建，拉取代码后需要本地构建，推送代码后触发持续集成服务需要构建，等等。这些操作有没有办法优化呢？

4.3.1.3　超大工程构建

现在我们把代码检出到了本地工作空间，并修改了代码，然后就可以构建了。开发者不仅需要读写源文件，还需要把它们编译测试之后再集成起来变成可以交付的软件产品，这个过程叫作构建。如果我们每次都从仓库的每一个源文件中完全从头开始构建，则在文件数量特别大时，构建花费的时间会很长，就像如图 4-10 所示的来自 xkcd.com 的著名漫画描述的场景。

构建的过程都是依靠构建工具来完成的，这些工具都提供了一些优化方法来提升构建的速度。这里有两种思路：一种是并行执行，充分挖掘 CPU 的潜力，同时构建多个目标；另一种是想办法把中间构建结果缓存起来，不用每次构建都从头来过。下面我们来看看 AOSP（Android Open Source Project）构建工具和 Android 应用的构建工具 Gradle 是如何运用这两种思路进行优化的。

AOSP的项目被拆分成了数百个模块，有不少模块之间没有依赖关系，它们可以同时构建来加快构建速度。AOSP的构建系统可以充分发挥CPU的能力，通过指定make命令的jN参数来执行并行编译。参数中的*N*是一个数字，代表同时执行的编译线程的数量，一般

是计算机硬件线程数的两倍，具体说明可以参考AOSP构建系统的文档[1]。下面这条命令意味着会同时有 16 个线程并行完成编译（意味着需要有一台拥有 8 核CPU的计算机）：

```
$ make -j16
```

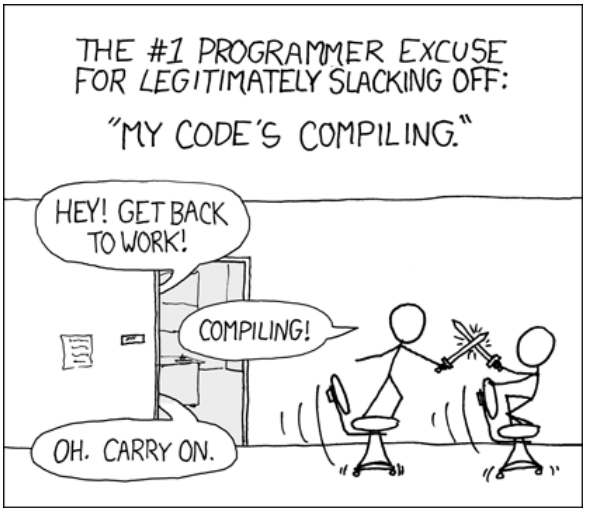

图 4-10　我的代码正在编译中[2]

AOSP是一个相当庞大的项目，代码量有几十GB，一般性能不错的开发机器（4 核 CPU、16GB内存再加上固态硬盘）即使使用并行构建，一次干净的编译也要花上数个小时，这对于持续集成来说是无法让人接受的。好在还有其他方法来加速整个编译过程，比如ccache[3]。ccache是适用于C和C++的编译器缓存，在编译过程中产生的文件可以缓存在磁盘上以供未来编译使用，因为在开发过程中大部分C或C++文件都不会发生变化，所以使用ccache缓存能大幅提高构建速度（可以缩短到原来的三分之一[4]）。

而普通的Android应用项目使用的构建工具是Gradle，Gradle一直以来令人诟病的就是它的构建速度，尤其是Android应用的构建速度。这中间有一部分是DEX性能的原因，也

1　https://source.android.com/source/building

2　https://xkcd.com/303/

3　https://source.android.com/source/initializing?hl=zh-cn#setting-up-ccache

4　http://blog.udinic.com/2014/06/04/aosp-part-2-build-variants/

有一部分是Gradle自身的原因。而Gradle也在不断地优化，现在这个问题终于得到了解决[1]：使用最新的Android Gradle Plugin 3.0 之后，Android应用的构建终于摆脱了缓慢的毛病，可以参考图 4-11 中的测试数据。

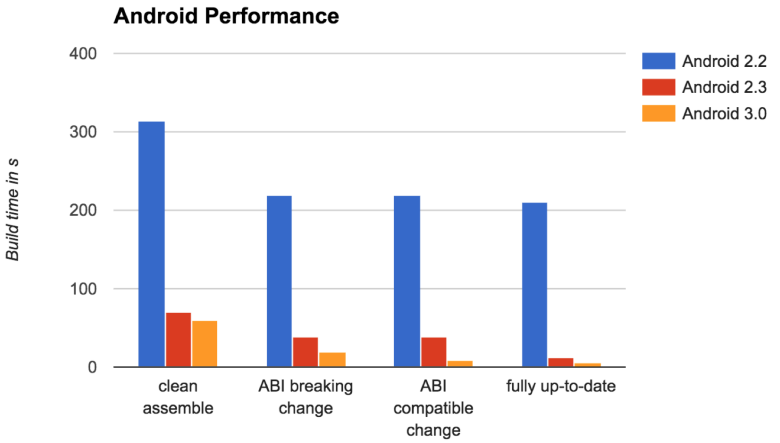

图 4-11　Android 构建的性能[2]

这除了得益于使用之前版本就已支持的Gradle配置，比如开启常驻的守护进程和开启并行编译，还得益于Gradle 3.5 引入的重要更新：Build Cache[3]。Build Cache会根据每个可以缓存的Gradle Task的所有输入文件计算出一个独一无二的哈希值，用来代表这个Task在缓存中的输出文件，只要输入没有变化，这个任务的输出文件就可以缓存起来被反复使用；而且它还支持一个本地网络中的缓存服务器，可以供团队中的所有成员构建时访问。据称，这个特性可以带来平均 25%的性能提升（构建时间平均减少 25%），在某些情况下可以带来最高 80%的性能提升。可以参考Android官方文档[4]来进行上述设置。包含这些配置的gradle.properties文件如下：

```
# 如果项目中的模块之间没有关系，则它们可以并行执行来加快构建速度
```

1　https://blog.gradle.org/blazing-fast-android-builds

2　https://blog.gradle.org/blazing-fast-android-builds

3　https://blog.gradle.org/images/android-performance.png

4　https://developer.android.com/studio/build/build-cache.html

```
org.gradle.parallel=true
# 守护进程存在，使得数据和代码在下一次构建前已经载入内存，会显著地提高后续构建的性能
org.gradle.daemon=true
# 开启构建缓存，可以把中间构建结果存储在缓存中供后续构建使用
org.gradle.caching=true
```

Build Cache功能不是由Gradle 4.0 首创的。它来自两位前辈的启发：Google的Bazel（由Google内部叫作blaze的构建工具开源而来）和Facebook的Buck（其实和Blaze有着千丝万缕的关系，是由前Google员工在Facebook开发出来的）。Buck和Bazel被称为终极构建工具。网上还有一份关于使用各种构建工具构建Java项目的速度对比的报告[1]（见表 4-3），其中包括第 1 次构建、没有文件变化时重新构建和有文件变化时增量构建的速度对比。可以看出，现代的Java构建工具在增量构建的速度上优势明显，它们背后都使用了缓存技术。

表 4-3　构建工具速度对比

对　比　项	sbt	Bazel	Buck	Gradle	Maven	Ant
第 1 次构建	82.4 秒	131.6 秒	148.3 秒	143.5 秒	141.7 秒	280.3 秒
没有文件变化时重新构建	65.6 秒	0.2 秒	1.5 秒	5.8 秒	136.3 秒	274.0 秒
有文件变化时增量构建	61.0 秒	126.7 秒	139.1 秒	146.7 秒	142.6 秒	281.6 秒

如果对该对比方法的细节感兴趣，则可以参考其测试方法[2]。

Buck的文档[3]详细地解释了它的构建速度如此之快的原因。

◎　它的每个构建规则（模块）都列出了影响该模块交付件的依赖，保证了构建是可重现的（若输入不变，输出的交付件就不变）。

1　https://tkruse.github.io/build-bench/2017_03_commons-math.html

2　https://tkruse.github.io/build-bench/README.html

3　https://buckbuild.com/concept/what_makes_buck_so_fast.html

◎ 构建规则形成有向无环图，依赖关系十分清晰，保证构建时可以尽可能并行。

◎ 可将构建规则的结果（交付件）缓存起来，供未来构建时重用，或者通过持续集成服务共享给其他环境在构建时进一步重用。

这两个工具还有一个优势就是可以支持多种技术栈，而不是像 sbt 或者 Gradle 那样只支持 JVM 语言的项目。如果项目中包含了使用不同技术栈的模块，还想进一步提升构建的速度，则不妨考虑 Buck、Bazel 或者其他类似的“终极”构建工具。

通过上面这些手段，我们可以优化单仓库代码的使用体验。但是随着时间的不断推进，在代码积累到超出单个仓库经过优化后的处理极限时，该来的始终会来，开发团队不得不做出拆分代码仓库的选择。首先要做的就是把单个代码仓库变成多个仓库，即拆分；然后把拆分出来的仓库组合在一起，即集成。那我们又需要什么样的工具和实践呢？变成多仓库之后，原来团队内的开发者之间的协作，很有可能变成不同团队之间的协作，那么我们应该如何协作呢？

4.3.2　代码仓库的拆分

如前所述，任何软件架构最终将导致代码库被拆分成不同的模块，各模块之间的关系是用依赖机制去定义和管理的。所以要拆分一个单一的代码仓库，首先应该引入合适的架构风格和依赖管理机制。常见的软件架构包括：分层架构、事件驱动架构、插件架构、微服务架构等。而特定的语言框架和工具都会有自己的依赖管理机制。对软件架构风格的讨论不在本书的写作范围内，但引入模块化之后一定会遵循当前技术栈的依赖管理机制。原有的代码按特定的依赖管理机制模块化之后，接下来就好办了，仓库中的代码模块可以独立拆分出去，或者引入新的仓库存放为新的模块。

对任何代码管理服务来说，都可以在服务器上建立一个新的仓库，只需要给它一个有意义的名字，然后就可以使用它（URL 中会包含新的仓库名称）来访问新的代码仓库了。如果要把新的仓库和原有的仓库集成，则只需修改原有仓库的依赖配置就可以了，这些我们会在下一节详细介绍。

但如果要把原本属于代码仓库的部分代码拆分到另外一个仓库，就要复杂得多。首先，有可能需要对原来的项目进行重构，把要拆分的代码先进行模块化，放到独立的目录中；其次，把这个独立的目录从原来的仓库中分离出去，变成一个新仓库，而这个新仓库最好还保留着原来的提交记录和标签，因为这对于开发团队来说是一笔重要的财富，它记录了该模块的来龙去脉；最后，把新仓库集成到原来的仓库中，集成的手段有很多种，也将在下一节具体介绍。

下面依然以 spring-framework 为例，来展示把文件目录从仓库中拆分出来的 Git 命令和参数。这一系列命令会把 spring-orm 这个目录从 spring-framework 中拆分成一个独立的仓库。

（1）复制完整的 spring-framework 仓库：

```
# 先把整个代码仓库复制到一个新的目录中，这里以 spring-orm 作为目录名，一会儿这里将只剩下这个目录中的内容
$ git clone git@github.com:spring-projects/spring-framework.git
spring-orm
# 进入这个新的目录下查看，这里有完整的代码仓库文件树，其中还有一个子目录 spring-orm，这是我们要拆分出来的目录
$ cd spring-orm
$ ls
```

（2）把所有和 spring-orm 子目录相关的提交记录从完整的提交记录中分离出来，并更新分支和标签：

```
# 接着，我们使用 Git 的 filter-branch 命令来修改提交历史记录，只留下和子目录 spring-orm 有关的提交记录，这些会变成新仓库的提交记录
$ git filter-branch --subdirectory-filter spring-orm \ # 只保留子目录 spring-orm
    --prune-empty \ # 不保留空的提交
    --tag-name-filter cat \ # 让原来的标签指向新的提交
    -- --all # 处理所有的分支
# 等一会儿再检查目录（当前目录的内容已经变成了 spring-orm 子目录中的内容），提交记录和
```

标签

```
$ ls
$ git log --oneline
$ git branch
$ git tag
```

（3）检查确认无误之后，把旧的提交记录去掉，准备推送到新的代码仓库中：

```
# 把和 spring-rom 无关的旧提交记录清除
$ git reflog expire --verbose --expire=0 --all
$ git gc --prune=0
```

（4）建立新的远程代码仓库用于存放拆分出来的 spring-rom，把代码推送上去：

```
# 新建一个代码仓库，并把它设置成 origin
$ git remote set-url origin git@github.com:qinyu/spring-orm.git
# 把所有分支和 tag 都推送到新的代码仓库，当然你也可以指定需要推送的分支和标签
$ git push --mirror origin
# 大功告成
```

Git 的 filter-branch 和 reflog 命令的详细解释如下。

我们之前见过filter-branch，它可以批量地修改大量的提交记录。这里使用它来选定一个子目录，并保留它的提交记录，并最后作为新的仓库根目录。reflog命令则用于维护和恢复提交记录，其官方也提供了一些例子 [1]。这里我们用这条命令来清除和新仓库无关的旧提交记录。

注意，上面的拆分过程可能会对原来仓库的提交记录造成破坏性的修改，应该在执行之前反复演练并做好备份。使用上面这一系列命令拆分出来的代码仓库包括了所有和它有关的分支和 tag。接下来我们看看另外一种更简单的拆分方法：使用 subtree 命令。

（1）创建一个裸仓库（没有工作空间），用来临时存放被拆分出来的代码。

1　https://git-scm.com/book/zh/v1/Git-内部原理-维护及数据恢复

```
# 创建一个裸仓库
$ mkdir spring-orm
$ cd spring-orm
$ git init --bare
```

（2）把 spring-orm 的代码作为一个分支分离出来：

```
# 复制完整的仓库到 spring-framework 目录
$ cd ..
$ git clone git@github.com:spring-projects/spring-framework.git
# 进入这个新的目录查看，这里有完整的代码仓库目录树，其中还有一个子目录 spring-orm，这是我们要保留的目录
$ cd spring-framework
$ ls
```

（3）接下来我们把 master 分支上的内容和 spring-orm 目录有关的所有提交记录拆分到一个新的单独的 split 分支：

```
# 默认拆分的是 master 分支上的内容和提交记录，如果需要拆分其他分支，则请指定其他分支名
$ git subtree split --prefix=spring-orm -b split
```

（4）检查一下存放分离内容的 split 分支，并把它推送到前面第 1 步新建的临时裸仓库：

```
# 检查一下这个新分支的提交记录
$ git log --oneline split
# 把这个分支推送到刚才我们创建的裸仓库
# 如果不是 master 分支，则请修改目标分支名
$ git push ../spring-orm/ split:master
```

（5）建立新的远程代码仓库，把代码推送上去：

```
# 进入裸仓库中
$ cd ../spring-orm
# 新建一个代码仓库，并把它设置成 origin
$ git remote add origin git@github.com:qinyu/spring-orm.git
# 把 master 分支推送到新的远程代码仓库
```

```
$ git push origin master
# 大功告成
```

这种方式拆分出来的仓库只有一个分支，也不会有任何标签，但是使用起来更简单。如果只是对子目录的某个分支上的提交有兴趣，对其他分支及标签不那么关心，则可以使用这种方式（这就是使用 subtree 集成依赖的场景，稍后我们就会看到）。而如果读者是一个完美主义者，则前面那套命令更适合；拆分出来的仓库与原来的仓库耦合更低，更适用于把模块作为二进制文件依赖或者服务依赖。

注意，被拆分出来的单个模块最好能编译成一个独立的交付件，可以用于其他模块的构建，或者作为服务独立部署。这会带来两个好处。

◎ 独立的二进制交付件可以缓存起来，构建工具可以利用缓存加快构建速度。部署系统可以直接用这些现成的二进制文件加快部署的速度。

◎ 独立交付意味着可以独立地测试和验证，可以在整个产品集成之前发现模块的错误，缩短问题发现的时间。

这也要求每个被拆分出去的模块都应该有自己完整的持续交付流水线，交付的是单独的二进制交付件或者独立部署的服务。所以在代码拆分之后还有工作要做，就是为这个独立的代码仓库创建自己的构建脚本。

4.3.3　代码仓库的集成

代码仓库的拆分只是第一步，单独拆出去的代码仓库一般不能直接作为面向最终用户的产品。它们还是得按照依赖管理机制的规则集成起来，组成一个完整的产品。在不同的依赖管理机制下，每个代码仓库（模块）上的工作流程有所不同，它们之间的集成方式也有所不同。有些模块在编译时会和其他模块的代码一起编译；有些模块则在编译时会使用到其他模块的二进制交付件。我们把组件之间的依赖关系分为两类：代码依赖和二进制文件依赖。

什么是依赖“地狱”？

模块化是如此诱人，开发者一旦习惯之后就会忍不住想要运用它。但如果产品中的依赖无节制地扩散开，就会产生依赖“地狱”的问题。比如，一篇博客里[1]提到的left-pad这个NPM库（NPM是JavaScript事实上的依赖管理工具，这个库用来完成在长度不够的字符串的左边填充空格），总共有 11 行代码，完全可以自己花上两分钟来实现，却做成了一个依赖库，而那些依赖它的知名库（如React、Babel）都因为它受到了影响而出现了问题。依赖地狱形成的原因如下。

- ◎ 依赖层次过多。要解析及获取全部的依赖非常耗时，尤其是这些依赖需要在远程服务器上下载时。
- ◎ 循环依赖。在这种情况下，依赖会形成死循环，根本无法解析。
- ◎ 依赖冲突。在层层依赖中，难免会出现同一个依赖的不同版本，而在解析过程中某个版本的依赖会被放弃，这可能会影响依赖它的功能。
- ◎ 向前兼容性。在升级依赖的时候，依赖的变化有可能会对产品造成破坏性的修改，产品被迫做出修改时会带来不可预期的风险。

所以解决依赖“地狱”问题成了依赖管理工具首当其冲要解决的问题。

依赖是两个主体之间的关系，它们之间需要形成某种约定，以确定双方的“权责”，我们赋予它一个时髦的新名字：契约。这份契约由提供方实现，它的内容以某种“标识”标记起来，并存放在一个公共的地方，以方便使用这份契约的消费方找到它。同时，一份契约还会随着时间的变化发生改变，所以它还需要一个版本号，表示这份契约在不同时间的内容。这样，根据契约的“标识”和版本，就可以定位一份契约的内容。这背后还有一个潜规则：某一版本的契约的内容必须固化下来，契约的内容若发生变化，则契约的版本号也要发生变化。这样，基于这份契约的产品功能才会得到保障，不会因为契约内容的变化受到影响甚至被完全破坏。否则契约的内容一旦改变而版本不变，就破坏了“可重现”的构建原则：基于相同的输入（契约的“标识”和版本号没有变），产生了不同的结果（契

1 http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/

约的内容变了，间接影响了产品的功能）。

一旦发生变化，版本号就必须改变，这样的规则看起来过于严苛，会阻碍契约的正常演进。但事实上这些变化还可以继续分类：一类是破坏性的修改，或者说不兼容的修改，这种变化发生时依赖该变化的消费方也要跟着改变；另一类是非破坏性的修改，或者说可兼容的修改，这种变化发生时，依赖它的消费方可以不用修改，依然信任它。那么版本号可以体现出这些不同类型的变化吗？当然可以，业界通用的版本号规范已经由Tom Preston-Werner（GitHub联合创始人）归纳总结，称之为语义化版本 [1]。

语义化版本顾名思义，就是在版本规则中隐藏着和依赖变化有关的含义。它提议用一组简单的规则及条件来约束版本号的配置和增长，解决依赖“地狱”的问题。

语义化版本的格式是：主版本号.次版本号.修订号，版本号递增的规则及版本编译信息如下。

◎ **主版本号：**当代码做了不兼容的对外接口修改时递增。

◎ **次版本号：**当代码做了向下兼容的新增功能时递增。

◎ **修订号：**当代码做了向下兼容的问题修正时递增。

◎ **先行版本号及版本编译信息：**可以加到“主版本号.次版本号.修订号”的后面，作为扩展。

语义化版本并不是全新的概念，它是根据已有的被广泛使用的规则制定的，现在已经被几乎所有依赖管理工具采纳。

根据语义化版本的规则，我们就可以推断出某次契约变化可能带来的影响，但这次影响的内容到底有哪些，对契约的使用方也非常重要。这里也有各种五花八门的工具来告诉使用者契约究竟发生了什么变化。最常规的手段就是随着版本的发布，契约的提供方会同时附加一份变化清单（Changelog）来描述发生的变化。如果使用Git来管理代码，而且遵循了提交消息格式的一定规范，那么就可以用工具来自动生成变化清单。可供选择的工具

1　http://semver.org/lang/zh-CN/

有很多：用Python写的gitchangelog[1]，它拥有自己的提交消息规范[2]；用JavaScript写的git-changelog[3]，它采用的是AngularJS提交消息的格式规范[4]，等等。除了附上这种文本，还可以借助平台把各版本契约的变化呈现出来，例如Android API用专门的网页[5]来记录各版本API之间的差异。如果项目采用的是REST风格的契约并用Swagger作为展现API（契约）的平台，则还可以试试Swagger::Diff[6]。

眼尖的读者已经发现问题了，我们看到契约的内容都是没有实际约束力的，即使提供方的文档描述得非常清楚，但是契约的内容真如文档所述吗？经验告诉我们，只读的“死”文档是不可靠的，只有可以执行的“活”文档才能给我们信心。“活”文档是什么？它就是项目中各种层次的测试代码及其在持续集成服务上不断验证的结果。如果提供给我们的是源码或者二进制库，则我们需要它的单元测试来验证它提供的方法调用；如果提供的是服务，则我们需要它的接口测试来验证它暴露出来的 API。只有看到了测试执行的结果是绿色的，我们才能相信契约的内容是真实可靠的。当然，如果消费方在契约内容的制定上是主动角色，则还可以尝试消费方驱动的契约测试方法。

契约测试是一种针对外部服务的接口进行的测试，它能够验证服务是否满足消费方期待的契约。这个契约包含了对输入和输出的数据结构的期望、性能及并发的要求。各个服务消费方的团队编写的契约测试套件打包存放在一起，并且在服务提供方的构建流水线上执行及验证。通过这种方式，服务提供方的维护者能够很容易地发现其修复对消费方的不利影响。因此，契约测试不仅能够为服务的消费方提供信心，对服务提供方的维护者来说其实价值更高。主流的契约测试工具是Pact[7]，如图 4-12 所示是它的具体调用过程。

1 https://pypi.python.org/pypi/gitchangelog

2 https://github.com/vaab/gitchangelog/blob/master/src/gitchangelog/gitchangelog.rc.reference

3 https://github.com/rafinskipg/git-changelog

4 https://github.com/rafinskipg/git-changelog#git-commit-guidelines---source--angular-js

5 https://developer.android.com/sdk/api_diff/24/changes.html

6 https://github.com/civisanalytics/swagger-diff

7 https://github.com/realestate-com-au/pact

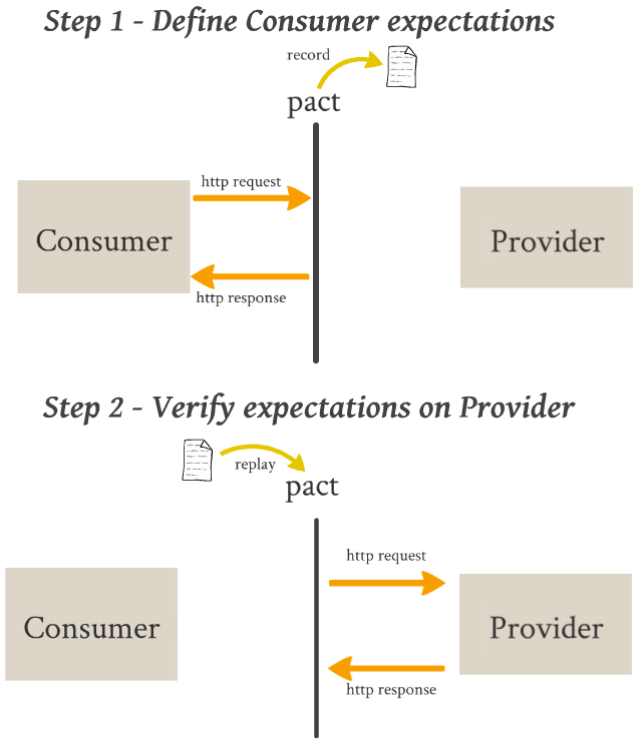

图 4-12　契约测试的调用过程 [1]

契约测试非常适合微服务架构，这种测试方法变得越来越流行。我们也可以在其他架构风格中采用同样的思路。

通过语义化版本、变更清单和消费者驱动的契约测试，可以解决依赖兼容性的问题，但还得依赖管理工具和实践来解决剩下的依赖“地狱”问题（依赖冲突、循环依赖和依赖层次过多）。不同的工具采取了不同的思路，比如Gradle若发现依赖中出现了同一个依赖的不同版本（依赖冲突），则默认 [2]会使用“最近”的那个版本；也可以选择让Gradle直接抛出冲突的错误，自己来选择要使用的版本。NPM的解析方式则允许同时使用同一个模块的

1　https://github.com/realestate-com-au/pact/raw/master/documentation/pact_two_parts.png

2　https://docs.gradle.org/4.0/userguide/dependency_management.html#sec:dependency_resolution

多个不兼容版本[1]，它可以将模块的这些不同版本放置在合适的地方，从而做到在想要的地方加载想要的版本。它们的解决方式和这些依赖在产品中的使用方式息息相关，对于Java 来说，最终类会被加载到JVM里，这就要求只有一个依赖能被使用，这时必须做出选择。

但我们不能把这些问题直接扔给工具解决，作为契约的提供方和消费方，在使用依赖管理机制时，我们也要遵循最佳实践来规避一些问题，同时让项目的依赖关系更清晰、更容易维护。契约提供方的实践如下。

（1）遵循语义化版本的规则，通过版本号告知破坏性的变化。

（2）通过持续交付自动流水线发布交付件，并把交付件上传到消费方可以获取的地方。

（3）发布之前必须通过完整的测试，保证实践和契约描述一致。

而契约消费方的实践如下。

（1）动态版本号只能在正式发布前的开发过程中使用，正式发布时必须使用固定的版本号，避免产生不可重现的构建结果。

（2）及时更新依赖的版本，避免变化积累太多，导致迁移到新版本会变得越来越难。

我们在了解了契约或依赖管理背后的原理和它们如何解决依赖“地狱”的问题后，才能更好地理解模块之间的关系。而我们在没有现成的依赖管理工具可用时，也能遵循这样的思路，设计并实施符合自己要求的管理机制。接下来我们再看看基于源码的依赖管理方式。

4.3.3.1 源代码依赖

源代码依赖是指模块在编译（编译型语言）或执行（脚本语言）时必须使用其他模块的代码。也就是说在每次编译或执行时，必须把被依赖的模块代码也拉取下来，按照一定的规则放在当前的目录结构中，最后形成一个完整的树状目录结构，当作“一个”完整的产品进行构建。采用这种方式后，模块之间的关系非常紧密，可以在工作空间中同时重构

1 https://npm.github.io/how-npm-works-docs/theory-and-design/dependency-hell.html

多个模块的代码，还可以随时使用模块的最新代码。如果拆分出来的模块代码能够做到和其余代码紧密协作甚至同步开发，而团队特别在意能够做产品级别的代码重构，则这会是一种非常合适的依赖管理机制。

很多代码管理工具都支持把在同一个目录树中的单个目录设置成不同的仓库，我们把整个产品的代码仓库叫作父仓库，其中包含的作为子目录存在的仓库，我们称之为子仓库，它们就是父仓库的依赖。首先，我们来看看这种机制是否包含了依赖管理的几个要素。

◎ 依赖的“标识”，对应子仓库的名称（或者完整 URL），通过它就可以在代码管理服务器上找到这个仓库。

◎ 版本号，对应子仓库中的提交号、分支或者标签，标签和分支也应该按照语义化版本的规则命名。

◎ 契约内容，对应子仓库中提交记录的消息，还有可以执行的测试代码。

而管理这些契约就需要利用代码管理工具了，比如 Subversion 就可通过配置子目录的 svn:externals 属性来指定这个子目录指向的代码仓库。而 Git 更提供了两种不同的选择：submodule 和 subtree。我们还是以前面拆分出去的 spring-orm 为例，看看它将怎样集成到原来的 spring-framework 仓库中。

1. 使用 subtree 来管理子仓库依赖

我们先看到的Git的“子命令”是subtree，其实它并不是Git自带的子命令，而是 2012 年第三方开发者为Git贡献的一个脚本 [1]，它利用Git的子树合并和子树拆分功能来管理多个代码仓库。它诞生的日期比submodule子命令要晚，但它适用的场景范围是最极端的，使用起来也是最简单的（如果使用方法正确）。

subtree 子命令的最大特点是，父仓库中包含了整个代码仓库的所有内容，也包含那些属于子仓库的内容，即子仓库的内容同时存在于两个仓库中。这意味着在拆分出子仓库的内容后，我们不用把子仓库的内容从父仓库中删除。

1　https://git.kernel.org/pub/scm/git/git.git/tree/contrib/subtree/git-subtree.sh

接着看前面拆分的例子，我们已经用 git subtree split 命令把子目录从仓库中拆分到独立的仓库中。下面我们来看看使用 subtree 命令如何把它添加到父仓库中：

```
# 首先，我们回到父仓库中，需要把子仓库添加到 remote 配置中，能使用 remote 配置的名字方便地引用（这里我们用了本地目录中的一个仓库 orm-repo 来模拟远程仓库）
$ git remote add orm ../orm-repo
# 接着我们通过 subtree 命令把子仓库添加到父仓库中
$ git subtree add \ # 第一次添加子树
    --prefix=spring-orm \ # 子目录的相对路径
    orm \ # 从哪个远程仓库拉取代码
    master \ # 把哪个分支的代码检出到子目录，可以指定分支、标签或者某次提交
    --squash # 我们不希望子仓库的提交记录“污染”主仓库的提交记录，所以把子仓库的所有提交记录压缩成一次提交
```

很遗憾，这次命令不会成功，我们会得到一条错误提示“prefix 'spring-orm' already exists.”，这是因为当前的工作空间中已经存在 spring-orm 目录。你也不能使用 subtree pull 命令来拉取子仓库，这将会得到这条错误提示“Can't squash-merge: 'spring-orm' was never added.”，告诉你子仓库还没有添加进来。所以我们这里使用 subtree split 命令来让主仓库达到使用 subtree add 命令后的状态。我们先学习命令，再来解释原理：

```
# 我们使用 split 命令把主仓库“缺失”的记录生成出来
$ git subtree split \ # 把子目录的提交分离出来变成一个新的分支
    --prefix=spring-orm \ # 子目录的相对路径
    --rejoin \ # 在主仓库的提交记录里增加一次提交，记录了符合 subtree 命令要求的子仓库状态（实际拆分出来的分支和当前分支的合并）
    -- master # 只把 master 分支上和子目录有关的提交分离出来
# 根据你的代码仓库的大小，这里需要一定的等待时间
...
# 先用 git show 查看最后一次提交的内容
$ git show
# 你看到的应该是类似这样的结果
commit 8904084508a960e8034cd6f31a5b689cde96cddd （HEAD -> master）
Merge: 15cf9c1d78 e87532a6ce
```

```
    Author: qinyu <*****@gmail.com>
    Date:   Sun Jul 2 09:43:45 2017 +0800
        Split 'spring-orm/' into commit 'e87532a6ceaaf63481331ce9c4e686401200
caec'
        git-subtree-dir: spring-orm
        git-subtree-mainline: 15cf9c1d78039b8b94793a4585032fbfd307c841
        git-subtree-split: e87532a6ceaaf63481331ce9c4e686401200caec
```

这次提交是 subtree 命令的关键，首先它是一次 merge，是父仓库 master（15cf9c1d78）和 spring-orm 拆分出来的分支（e87532a6ce）的合并，在上面的例子中使用 git show 进行了确认。这次合并提交的消息内容也符合 subtree 约定的格式，包含非常重要的字段：git-subtree-dir、git-subtree-mainline 和 git-subtree-split。从这三个字段的命名和内容中不难猜出它们的意义：它们分别记录了子仓库对应的子目录路径、主仓库的提交记录 SHA1 和子仓库的提交记录 SHA1。subtree 的 add、merge 和 pull 命令就是在提交记录中查找符合格式的提交记录，并使用这些字段的值来判断是否应该做新的合并。

你还可以查看整个仓库的提交记录来验证上面的描述：

```
    $ git log --graph --oneline
    # 你看到的应该是这样的结果，可以看到出现了一个新的分支，它的最新提交记录 SHA1 是
`e87532a6ce`，这个分支包含的就是从主仓库中分离出去的和 spring-orm 有关的历史记录，它就
是子仓库的 master 分支
    *    8904084508 （HEAD -> master, origin/master, origin/HEAD） Split
'spring-orm/' into commit 'e87532a6ceaaf63481331ce9c4e686401200caec'
    |\
    | * e87532a6ce （orm/master） @Nullable all the way: null-safety at field level
    | * e7d699423d Deprecate adapter classes for async interceptors
    | * 3c42c830f7 Add onError callback to DeferredResult
    | * 8837bd01af Refactor iterator of Map with Java8's Map.forEach
    | * 274e6b3369 Polish
    ...
```

现在你可以再使用 git subtree pull --prefix=spring-orm orm master --squash 命令来拉取

子仓库的代码，只不过现在主仓库不会发生任何变化，spring-orm 子目录的内容已经是最新的了，而且主仓库也正确地记录了子仓库的状态：

```
    $ git subtree pull \ # 拉取子仓库的变化
        --prefix=spring-orm \ # 子目录的相对路径
        orm master \ # 从哪个仓库的哪个分支拉取
        --squash # 把子仓库的提交记录压缩成主仓库中的记录，避免子仓库的提交记录“污染”
    # 现在你会得到类似下面的提示，告诉你 spring-orm 已经是最新的了
    ...
    Subtree is already at commit e87532a6ceaaf63481331ce9c4e686401200caec.
    # 现在没有问题了，可以推送主仓库了，当然记得推送代码之前先拉取，如果出现了冲突，则还要解决冲突
    $ git pull
    ...
    $ git push
    # 大功告成
```

如果你是在一个全新的父仓库中使用 subtree 添加子仓库，则直接使用 git subtree pull --prefix=spring-orm orm master --squash 命令就可以了。

```
    # 首先，我们回到父仓库的工作目录中，如果还没有把子仓库添加到 remote，则先添加进来
    $ git remote add orm ../orm-repo
    # 然后直接使用 subtree pull 命令
    $ git subtree pull --prefix=spring-orm orm master --squash
    # 你可以查看提交记录
    $ git log
    # 结果类似这样
    commit 72877ab034a77ef6ce6600c4c0ebe37210daa9e8 (HEAD -> master)
    Merge: f1566df 4a2f9fd
    Author: qinyu <*****@gmail.com>
    Date:   Sun Jul 2 12:53:36 2017 +0800
        Merge commit '4a2f9fde962ade4ddc9ca2d02c63a9301ac7f7cd' as 'spring-orm'
    commit 4a2f9fde962ade4ddc9ca2d02c63a9301ac7f7cd
```

```
    Author: qinyu <*****@gmail.com>
    Date:   Sun Jul 2 12:53:36 2017 +0800
        Squashed 'spring-orm/' content from commit e87532a
        git-subtree-dir: spring-orm
        git-subtree-split: e87532a6ceaaf63481331ce9c4e686401200caec
    commit f1566df7ecfe00471bf34ca1884679603add3a22
    ...
    # 现在没有问题了，可以推送主仓库了，当然记得在推送代码之前先拉取，如果出现了冲突，则还要解决冲突
    $ git pull
    ...
    $ git push
    # 大功告成
```

和刚才在 spring-framework 里看到的有些区别，但提交记录消息中的关键字段都还在。现在另一位开发者只要拉取主仓库的内容就可以了。如果不需要向子仓库贡献代码，则甚至连子仓库都不用添加，因为主仓库已经完整包括了所有代码。但现实的场景不会这么简单，来看看下面两个常见场景。

（1）子仓库的代码有更新，需要在主仓库里使用它。子仓库分离出去的一个好处就是可以和主仓库的开发分开，甚至可以由其他团队独立进行。而作为主仓库的开发者当然需要第一时间利用这些子仓库的开发成果。

（2）在主仓库里直接修改属于子仓库的代码，再提交到子仓库。子仓库虽然分离了出去，但在主仓库的开发者本地的工作空间中，代码仍然是完整的，包含了所有父仓库和子仓库的内容。在其本地环境里，开发者可以自由地修改代码。在属于子仓库的代码被修改后，开发者需要把代码推送回去，供其他开发者使用。

这两种场景也是代码依赖与其他依赖管理机制相比的最大优势：随时随地可以在最新版本的代码上工作，随时随地可以自由地修改属于产品的任意代码。

先来看第 1 个场景，子仓库的代码由其他团队做了修改并提交（不是在主仓库中直接修改），现在要在主仓库中使用这些修改。这里我们假设 spring-orm 的子仓库上已经有了

两次新的提交：

```
    # 拉取子仓库的代码并合并到主代码仓库中
    $ git subtree pull --prefix=spring-orm orm master --squash
    # 大功告成，可以通过查看提交记录确认
    $ git log
    # 你看到的提交记录应该类似这样，多了最新的两条提交记录，记录的是新的 subtree 合并的内容，也包含了 subtree 约定的那些字段
    commit ed133ed327ba06b7c951ea18e97373c38381cb58 （HEAD -> master）
    Merge: 8904084508 81c1f21de1
    Author: qinyu <*****@gmail.com>
    Date:   Sun Jul 2 14:16:58 2017 +0800
        Merge commit '81c1f21de1fc0c4c3f6986dc8c2c2c014c5833d6'
    commit 81c1f21de1fc0c4c3f6986dc8c2c2c014c5833d6
    Author: qinyu <*****@gmail.com>
    Date:   Sun Jul 2 14:16:58 2017 +0800
        Squashed 'spring-orm/' changes from e87532a6ce..c63f30fe0d
        c63f30fe0d Changes from third-party again
        f46558a27e Changes from third-party
        git-subtree-dir: spring-orm
        git-subtree-split: c63f30fe0d4cc0bd9933f51ede78b2e8ccaa36ca
    commit 8904084508a960e8034cd6f31a5b689cde96cddd （origin/master, origin/HEAD）
    Merge: 15cf9c1d78 e87532a6ce
    ...
    # 在推送代码之前一定要先拉取更新
    $ git pull
    # 如果出现冲突，则在解决冲突后再提交
    ...
    # 推送到远程仓库
    $ git push origin master
    # 大功告成
```

subtree pull --squash 命令始终会产生两条提交记录，前一条提交记录是由最新拉取的子仓库提交和上次拉取的子仓库提交(即上一条包含 git-subtree-split 约定字段的提交记录)之间所有的提交压缩而成的；后一条则是这次提交和 master 分支的合并记录。这样拉取操作是增量的，而合并操作确保了这些内容出现在主仓库中。

接下来我们看看第 2 种更复杂的场景，在主仓库里修改属于子仓库的代码并推送到子仓库：

```
# 我们已经修改了 spring-orm 子目录中的内容(当然你可以同时修改非子仓库的代码并一起提交和推送)，在本地主仓库中做了两次提交并推送到了主仓库中，确认一下
$ git log
# 看起来是这样的：
commit 1045fde798dad779906a3f480f67129a5ca8ed63
Author: qinyu <*****@gmail.com>
Date:   Sun Jul 2 17:10:30 2017 +0800
    Changes from dev A again
commit bfa85765926afaeed2797be779f9c1e6fa11f656
Author: qinyu <*****@gmail.com>
Date:   Sun Jul 2 17:10:05 2017 +0800
    Changes from dev A and modify container
...
```

现在我们把主仓库中关于 spring-orm 的这几次提交也推送到子仓库中：

```
# 在推送之前先更新，如果有冲突，则先解决冲突
$ git pull
...
# 推送修改到子仓库
$ git subtree push \
   --prefix=spring-orm \ # 要推送的代码属于哪个子目录
    orm master # 推送到哪个仓库的哪个分支
# 大功告成，可以在单独的子仓库的工作空间中拉取确认
```

现在我们已经完成了假设的两种最常见的场景，subtree 目前都能胜任。但有一点不方

便，在 subtree 特殊约定的提交记录的消息格式中，没有包含子仓库的地址；这样在每次执行 subtree 命令时都要指定仓库的地址（或者像上边这样先把子仓库加入 remote 配置中来简化输入参数）。而在代码仓库变得更复杂之后，subtree 也会出现一些问题。

首先，subtree 并没有批处理操作多个子仓库的方式，这意味着如果仓库中包含多个子仓库，则需要逐个仓库地分别执行命令（当然可以写脚本来完成批处理）。而我们注意到，subtree 拉取产生的提交记录都在主仓库的一个分支里，这意味着多个子仓库的拉取记录会展开平铺并散落在主仓库这个分支上的提交记录中。在提交记录中本来就不清晰的依赖关系变得更难以理解，这是 subtree 命令的硬伤，无法解决。

其次，若多个团队成员同时使用 subtree 来拉取同一个子仓库的代码，则在各自的本地主仓库中会产生各自的提交，虽然内容一样，但在主仓库拉取时会产生一些相同的或空的合并提交，“污染”主仓库的提交记录。就像这样：

```
    $ git log --oneline
    # 下面的 5 次提交记录，前 4 次分别来自两位开发者的 subtree pull 4db5829e5b 和 5ee653c737 中的一位，而 23a5319ebc 和 634afb872a 来自另一位，内容是一样的；最后一次 ca7b29b4e2 则是两位开发者各自提交内容的合并，它的内容是空的，因为两位开发者分别拉取的子仓库的内容实际上是一样的
    ca7b29b4e2 （HEAD -> master, origin/master, origin/HEAD） Merge branch 'master' of /Users/yqin/Github/subtree-playground/framework-repo
    4db5829e5b Merge commit '5ee653c737dc8f2884d99a7d6bb7bc2c31fb5482'
    5ee653c737 Squashed 'spring-orm/' changes from b719a9806e..3f7a209203
     README.md | 1 +
     1 file changed, 1 insertion (+)
    23a5319ebc Merge commit '634afb872a90ef912e71d0ee75bfd3ddd1cfa36a'
    634afb872a Squashed 'spring-orm/' changes from b719a9806e..3f7a209203
     README.md | 1 +
     1 file changed, 1 insertion (+)
```

要避免这种问题，可以通过在团队内部设置一个专门的角色来负责所有子仓库的拉取操作，避免多人并发操作，也可以使用“令牌”，由拿到令牌的开发者执行这次操作。

然后，切换子仓库的分支时十分烦琐，因为主仓库中的代码和子仓库完全使用合并的方式来处理，如果通过 subtree 命令直接拉取子仓库的一个新分支，则实际的效果是将子仓库的代码和现有主仓库中对应的代码进行了一次合并。这并不是我们切换子仓库分支的初衷（我们期望的是代码完全变成新分支的内容）。可行的方式是删除子仓库的目录，从子仓库的分支上一次性全新地拉取。

最后，子仓库和主仓库推送的顺序也很重要。如果子仓库的代码在主仓库的工作空间中进行修改，就会被先推送到子仓库中，而不是主仓库里；那么其他主仓库上的开发者就不能及时拉取属于子仓库的这部分修改；这可能导致其他开发者直接使用 subtree pull 来拉取子仓库的代码。这样的话，虽然两位开发者的本地工作空间中的代码是一样的，但其各自的主仓库的提交记录大不相同。在其分别推送代码之后，必然需要合并甚至解决冲突，主仓库的提交记录又会被“污染”了（同前面看到的一样）。我们也需要一些规则来避免这样的问题。

（1）在推送代码时应该先推送主仓库，再推送子仓库。

（2）在拉取代码时，也应该先拉取主仓库的代码，再考虑从子仓库中拉取代码（当然拉取代码的操作要么专人负责，要么获得令牌后执行）。

关于 subtree 的使用方式我们先介绍到这里。现在总结一下使用 subtree 时的一些最佳实践。

（1）add、merge 及 pull 时都使用--squash 参数，避免子仓库的提交记录“污染”主仓库的提交记录。

（2）在主仓库的工作空间修改子仓库的代码提交后，应该先推送到主仓库，再推送到子仓库。

（3）从子仓库拉取代码的操作应该由专人负责，或者采用“令牌”机制。

使用 subtree 的一些缺陷如下。

（1）依赖“契约”的内容，没有统一的管理，散落在文档（仓库地址）和提交记录（版本号）中。

（2）没有直观地使用语义化版本规则来存放依赖的版本号（提交 SHA1 时可以对应某个符合语义化版本规则的标签，但不直观）。

（3）不能方便地进行批量操作，也不能方便地切换子仓库的分支。

它非常适用于以下场景。

（1）团队的开发工作以主仓库为主，代码的修改主要发生在主仓库内；子仓库只用于共享代码给其他团队用。

（2）以子仓库来共享代码的各个团队之间可以无缝地协作，甚至在同一个团队中也可以；对所有的仓库都有相同的访问权限。

（3）主仓库引用的是子仓库的固定分支上的最新代码，始终在这分支上向前滚动开发。

2. 使用 submodule 管理子仓库的依赖

使用 submodule 管理子仓库时，每个子仓库里的内容和提交记录与主仓库是分开的，它们是完全独立的两个仓库，只是在本地工作空间中嵌套地放在一起。主仓库中只存有子仓库中某次提交的引用，并没有子仓库的完整内容，这一点和 subtree 大相径庭。有一种说法很形象：subtree 的子仓库里是复制的内容，而 submodule 的子仓库里是链接。我们使用和前面一样的例子来说明 submodule 的用法，把 spring-orm 作为一个子仓库加到父仓库中。

因为 submodule 的仓库是分开、独立的，所以我们要把 spring-orm 目录从原来的仓库中移除，否则无法加入 submodule。最简单的方法就是：

```
$ git rm -r spring-orm
$ git commit -m "Remove original spring-orm"
```

然后我们把子仓库作为 submodule 添加进来：

```
# 注意和 clone 一样，下面这条命令会创建一个子目录，并把子仓库的代码检出到子目录中，默认使用子仓库的 master 分支
$ git submodule add ../orm-repo spring-orm
```

```
# 可以用 git diff head 查看这次提交的内容，它看起来大概是这样的：
$ diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000000..5390c943a1
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "spring-orm"]
+   path = spring-orm
+   url = ../orm-repo
diff --git a/spring-orm b/spring-orm
new file mode 160000
index 0000000000..e87532a6ce
--- /dev/null
+++ b/spring-orm
@@ -0,0 +1 @@
+Subproject commit e87532a6ceaaf63481331ce9c4e686401200caec
```

首先我们会注意到新增的.gitmodules 文件记录了子仓库对应的子模块在工作空间中的路径和它的仓库地址。然后请注意最后 7 行，spring-orm 作为一个新的“目录”被添加到了代码仓库中，而它的“内容”是子仓库的某次提交的“链接”，这就是这份契约的“标识”和版本号。所有这些都是主仓库中这次提交包含的所有内容。这个“标识”和版本号最后会同其他提交记录一起被推送到远程仓库里：

```
# 提交主仓库中的修改
$ git commit -m "Add submodule spring-orm"
# 在推送之前记得先 pull
$ git push
```

而另一位开发者在复制代码时，使用的命令参数和普通的命令参数稍有不同：

```
# 从主仓库中检出代码，同时初始化和更新子仓库的代码
$ git clone framework-repo --recursive spring-framework-other
```

--recursive 参数的意思是，在复制主仓库的同时，递归（如果在 submodule 中嵌套了 submodule）地更新其中所有 submodule 的代码，依据的是.gitmodules 文件中记录的各 submodule 的仓库地址和具体的提交记录。

到目前为止，这一切看起来很简单，但我们不可能只检出子仓库中的代码就了事，我们还需要在分离出去的子仓库上做开发，并且希望这些修改能够在父仓库中使用。我们还是看一下 subtree 例子中出现的两种场景。

（1）子仓库的代码有更新，需要在主仓库里使用它。

（2）在主仓库里面直接修改属于子仓库的代码，要把它提交并推送回子仓库中。

在第 1 种常见场景中，主仓库的开发者在需要用到子仓库中新的提交时，需要先更新本地工作空间中子仓库的内容，再更新主仓库的.gitmodules 文件中该 submodule 引用的提交对应的 SHA1 值。在更新代码之前，我们进入 submodule 的子目录查看子目录的一些状态：

```
# 进入 submodule 对应的目录中，之后所有 Git 命令操作都应用于子仓库（即子仓库是激活的）
$ cd spring-framework-other/spring-orm
# 先查看哪个是远程仓库（这里用本地的一个仓库 /Users/yqin/Github/submodule-plyaround/orm-repo 代替）
$ git remote -v
origin  /Users/yqin/Github/submodule-plyaground/orm-repo （fetch）
origin  /Users/yqin/Github/submodule-plyaground/orm-repo （push）
# 接着看一下工作空间的状态
$ git status
HEAD detached at e87532a
nothing to commit, working tree clean
```

我们可以看到，一旦进入 submodule 对应的子目录，外层主仓库就仿佛被屏蔽了。所有 Git 的命令操作都只会在子仓库中起作用，不会影响到主仓库。这是使用 submodule 的关键，我们需要时刻牢记当前工作空间“激活”的是哪个仓库。另外需要注意的是，子仓库的代码处于游离状态（detached），并没有指向任何分支。

接着就可以更新代码了：

```
    # 获取子仓库的远程代码仓库中的新的提交
    $ git fetch
    # 检出你需要的那次提交的代码（可以是一个分支、一个标签或者一次提交）
    $ git checkout origin/master
    Previous HEAD position was e87532a... @Nullable all the way: null-safety at
field level
    HEAD is now at 9bc930f... Changes from third-party
    # 检查当前工作空间的状态，确认我们已经拿到想要的代码了
    $ git log
```

现在你可以在本地编译、构建代码，让子仓库中的新代码发挥作用了。但是注意我们在前面提到，现在的变化都发生在子仓库中，外层的主仓库并没有记录子仓库的变化：

```
    $ cd ..
    # 你可以看到，在主仓库的提交记录中并没有记录这次子仓库的变化并回到主仓库目录（现在激活
的是主仓库）
    $ git log
```

团队的其他开发者都无法得知这次变化，也就无法使用子仓库中的这些新代码。你必须明确地告知主仓库发生了这次变化，并做一次推送。

```
    # 你可以用 diff 命令看到这次变化，但它还没有被加到暂存区
    $ git diff
    diff --git a/spring-orm b/spring-orm
    index e87532a6ce..9bc930f3cb 160000
    --- a/spring-orm
    +++ b/spring-orm
    @@ -1 +1 @@
    -Subproject commit e87532a6ceaaf63481331ce9c4e686401200caec
    +Subproject commit 9bc930f3cbc77c22d001202d38967239d3aa46fd
    # 把 submodule 的变化（子仓库引用的提交记录的变化）提交到主仓库
    $ git add spring-orm
    git commit -m "Update submodule spring-orm"
```

```
# 推送到远程仓库，记得在推送之前先 pull
$ git push origin master
```

好了，现在其他成员可以拉取这次提交了：

```
# 拉取主仓库的代码
$ git pull
# 递归地更新 submodule 里面的代码，如果没有初始化，则先初始化
$ git submodule update --init --recursive
```

现在我们来看看第 2 种更加复杂的场景中（直接在主仓库中修改子仓库中的代码）又需要多少命令。

首先，我们要进入 submodule 的子目录中并检出正确的分支（因为子目录的代码处于游离状态）：

```
$ cd spring-orm
# 因为子仓库的代码处于游离状态，所以首先需要让它处于一个分支中，这里直接使用 master
$ git checkout origin master -t
```

接下来在你的 IDE 或编辑器中修改子仓库中的代码（当然可以同时修改不属于子仓库的代码），在完成测试后，需要先准备提交和推送子仓库的代码（因为父仓库和子仓库的代码是独立的，所以需要分开提交）：

```
# 注意，现在还是在 spring-orm 的子目录中提交子仓库的所有修改
$ git commit -a -m "Changes from dev A"
# 推送提交到子仓库的远程仓库
$ git push
```

和之前的场景一样，上面的修改、提交和推送全部只和子仓库有关。现在我们回到主仓库中，提交并推送主仓库的变化（submodule 现在指向了子仓库的一次新的提交）

```
$ cd ..
# 把 submodule 的变化（子仓库引用的提交记录的变化）及其他非子仓库的代码修改提交到主仓库
```

```
$ git commit -a -m "Update submodule spring-orm"
# 推送到主仓库的远程仓库，在推送代码前一定要先拉取更新
$ git push origin master
```

这样我们就完成了外层父仓库的提交和推送，现在团队中的其他开发者可以拉取主仓库的代码变化了，拉取后别忘了执行 git submodule update 来更新子仓库对应的子目录中的代码。对于 submodule 的合并，Git 处理得非常好，基本不需要人为干预。

现在我们看看在 submodule 命令使用的过程中容易出错的地方。由于主仓库和子仓库是分开、独立的两个仓库，所以它们的操作是各自独立的。可以看到上面这些例子中拉取或者推送的操作都是两段式的：先拉取主仓库再更新子仓库；或者先拉取子仓库再修改推送主仓库；又或者先修改、推送子仓库再推送主仓库。任何操作都需要在两个仓库上进行，所以在执行任何操作之前，得明白当前哪个仓库是“激活”的。

还有一点，Git 几乎没有提供任何参数让这种两段式的操作自动、先后发生，所以得时刻牢记当前的状态，不要忘记提交任何内容。最容易犯错的是拉取主仓库的变化（其中包括了子仓库的提交引用）后忘记了 submodule update；在下一次主仓库提交时不小心使用了-a 的参数，导致主仓库中的子仓库引用回滚到之前的提交状态。这些问题可以通过 git alias 或者脚本定制来解决：

```
# 设置 Git 命令的别名来合并两次操作。
$ git config alias.spull '!'"git pull && git submodule update --init
--recursive"
# 使用这条命令拉取主仓库的代码，就不怕遗漏子仓库的更新了
$ git spull
```

和 subtree 命令一样，对子仓库和主仓库的二段式操作是有顺序的，在拉取时由“外”向“内”，即先拉取主仓库，再更新子仓库。在推送时顺序要反过来。前面的例子是由外向内的。下面我们来看推送时易出错的一个例子，修改完子仓库的代码并提交到子仓库（没有推送）后，在主仓库里提交指向子仓库的新提交的变化，如果这时直接推送了主仓库，则其他开发人员拉取主仓库的代码后将无法更新子仓库的内容（submodule 的远程仓库上根本没有这次提交）。针对这种情况，git push 命令提供了--recurse-submodules 参数，在推送时检查

当前仓库的所有 submodule 里是否有提交没有推送到它的远程仓库。它有两个值：check 和 on-demand，我们不难猜到它们的效果分别是什么，有兴趣的读者可以自己尝试一下。

至于包含多个子仓库或者嵌套子仓库的父仓库，submodule 命令提供了--recursive 参数来处理。如果还需要 submodule 命令默认没有提供的批量操作，则可以试试 submodule foreach 命令。总体来说，用 submodule 命令处理多个仓库是很强大的。

subtree 切换子仓库的分支不是很方便，那么 submodule 支持的效果如何呢？可以试试在 git submodule update 命令中加上--remote 参数，它会拉取子仓库当前跟踪分支的最新代码。那么这个跟踪分支在哪里指定呢？答案是在默认的约定和.gitmodules 文件中。我们在前面添加 submodule 时没有指明分支，默认跟踪 master 分支的最新提交；如果在 git submodule 时用-b 参数指定了分支，则会跟踪指定的分支。这个指定的分支也记录在前面的.gitmodules 文件中，看起来如下：

```
[submodule "spring-orm"]
   path = spring-orm
   url = ../orm-repo
   branch = .
```

每个子仓库的 branch 字段的值就是跟踪分支。注意，上面例子中 branch 的值“.”非常特殊，它的含义是所有子仓库的跟踪分支的名字沿用主仓库当前跟踪分支的名字。这样可以方便地切换所有仓库（包括主仓库和子仓库）的跟踪分支，但前提是所有子仓库采用同样的分支策略（至少每个仓库都得拥有同样名字的分支）。现在，如果需要切换某个子仓库的分支，则修改.gitmodules 文件中相应子仓库的 branch 字段的值，再使用 git submodule update <子仓库名> --remote 就可以了。

另外，再次提醒，git submodule update --remote 只负责更新子仓库的代码，对于主仓库还得自己提交和推送，切记。

我们可以发现，现今的 submodule 命令已经很强大了，基本可以满足代码仓库依赖的大部分使用场景，但使用时仍需遵循一定的规则，如下所述。

（1）子仓库和父仓库是完全独立的两个仓库，在执行 Git 命令时需要注意当前“激活”

的是哪个仓库。

（2）所有操作都是“二段式”的，子仓库和主仓库的操作缺一不可，而且要注意仓库操作的顺序。

（3）子仓库的代码在更新之后并没有对应的分支，处于游离状态（detached），这时要继续在子仓库中工作，应该先检出一个分支。

（4）主仓库包含多个子仓库时，带上--recursive 是一个非常好的习惯。

（5）主仓库和子仓库使用相同的分支策略，可以使用“.”作为统一的跟踪分支来简化分支切换。

（6）尽量别使用嵌套的子仓库。

规则是不是太多？submodule 很强大，但命令比较复杂，操作比较多，有时还需要借助别名和脚本。我们不能指望团队中的每个开发者都是 Git 命令专家，那么有没有更简捷的解决方案呢？

3. 使用 Repo 来管理子仓库依赖

在学习完 submodule 和 subtree 之后，我们会发现它们有一些共同的问题。

（1）不能直观地看到父仓库中的所有依赖及它们的版本，它们的信息散落在.gitmodule 文件成提交记录中。

（2）批量处理多个子仓库不是很方便，命令很烦琐。

（3）切换分支不很方便。

这时，Repo就是另外一个选择。作为第 1 个大规模使用多仓库来集成产品的项目，AOSP在使用多仓库管理代码时，gitsubmodule并没有现在这样丰富的功能（特别是缺少追踪分支），Google自己创造了一个管理大规模子仓库的工具Repo[1]。AOSP的规模有多大？

1　https://source.android.com/source/developing

下面是一组来自openhub[1]的数据。

◎ 总共有 533 289 次提交。

◎ 总共有 3477 位贡献者。

◎ 总共有 13 502 957 行代码。

◎ 最近 30 天一共有 2548 次提交。

从一开始，根据架构的不同层次，Android就被划分成了 200 多个子仓库，到目前为止已经发展到 1044[2]个，包括了不同的硬件驱动、设备内核、开发工具、框架、应用，等等。

但一个单一设备的 ROM 不会使用到全部 1044 个子仓库，它往往有选择地由部分子仓库组合而成。在 AOSP 中，Google 通过一个清单文件来描述一个设备需要使用到的所有子仓库。而 Repo 的核心功能就是根据清单文件对子仓库群进行批量管理。

下面是 AOSP 主分支（master）上的清单文件的代码片段：

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote  name="aosp"
           fetch=".." />
  <default revision="master"
           remote="aosp"
           sync-j="4" />
  <project name="accessories/manifest" />
  <project name="brillo/manifest" />
  <project name="device/asus/deb" />
  <project name="device/asus/flo" />
  <project name="device/asus/flo-kernel" />
  <project name="device/asus/fugu" />
```

1 https://www.openhub.net/p/android

2 https://android.googlesource.com/mirror/manifest/+/master/default.xml

```
  <project name="device/asus/fugu-kernel" />
  <project name="device/asus/grouper" />
  <project name="device/asus/tilapia" />
  <project name="device/casio/koi-uboot" />
...
```

这份清单文件和git submodule的.gitmodules文件类似，记录了仓库的地址（remote）、各个子仓库（project）的名字（name）和它们默认（default）的分支（revision），等等。这份清单文件的格式可以参考官方文档[1]，它完整地描述了使用Repo管理代码仓库时的“契约”。

除了这份清单文件，Repo 还能对这些子仓库进行批量管理。日常使用过程大致如下。

（1）运行 repo init 命令，复制 AOSP 的一份清单文件。

（2）运行 repo sync 命令开始同步，将分别复制这份清单文件记录的多个子仓库到本地的工作空间中。

（3）使用 repo start 新建一个主题分支。

（4）修改文件。

（5）使用 git add 暂存更改。

（6）使用 git commit 提交更改。

（7）使用 repo upload 将更改上传到审查服务器中。

具体的命令请参考AOSP文档[2]，这里不再赘述。

repo upload 命令会将代码推送到代码审查服务器 Gerrit 上，Gerrit 也是 Google 开发的用于代码审查的在线工具，使用 Gerrit 代码审查是 AOSP 推荐的开发流程必经的一步。如果选择 Repo 作为源代码依赖的管理手段，则最好和 Gerrit 一起使用。

1　https://gerrit.googlesource.com/git-repo/+/master/docs/manifest-format.txt

2　https://source.android.com/source/developing

4.3.3.2 二进制依赖

二进制依赖是指模块化之后每个模块都能被构建成二进制文件并存放到某个缓存（本地或远程）中，而依赖这些模块的产品只需要用一些“坐标”来说明依赖关系，在构建时通过特定的依赖管理工具从缓存中得到这些二进制文件，在构建的过程中使用这些依赖的二进制文件。几乎每种现代编程语言都会提供这样的二进制管理工具，有的甚至还拥有多种管理工具，有的还直接和构建工具集成在一起。这些工具使用专门的配置文件记录一个项目所依赖的库（每个依赖的条目都包括“标识”和“版本号”）；在构建时根据配置文件中的依赖条目，按照一定的规则进行解析（主要处理重复和冲突的依赖）；根据解析的结果从中央制品库上下载相应依赖的二进制文件，并在构建中使用这些二进制文件。每种语言都会有官方或者第三方维护的集中式中央制品库，它是这门语言生态圈的基础，社区中的开发者利用中央制品库来存储和分发库文件。表 4-4 列举了一些主流编程语言的依赖管理工具及它们使用的制品库、配置文件及依赖条目的格式。

表 4-4 主流语言依赖管理工具

语 言	工 具	中央制品库	配 置 文 件	依赖条目“标识”和“版本”格式示例
JVM 语言	gradle	bintray.com	build.gradle	groupid:artifactid:version
JVM 语言	maven	central.sonatype.org	pom.xml	<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>[4.0</version> <type>jar</type> <scope>test</scope> <optional>true</optional> </dependency>
C#	nuget	www.nuget.org	.nuspec	<dependency id="PackageA" version="1.1.0" /> <dependency id="PackageB" version="[1,2) " />

续表

语　言	工　具	中央制品库	配 置 文 件	依赖条目“标识”和“版本”格式示例
Node.js	npm	www.npmjs.com	package.json	{ "dependencies" : { "foo" : "1.0.0 - 2.9999.9999", "bar" : ">=1.0.2 <2.1.2", "baz" : ">1.0.2 <=2.3.4", "baz" : "git+https://isaacs@github.com/npm/npm.git" } }
Ruby	bundler	rubygems.org	Gemfile	gem "coffee-rails" gem "sass-rails", github: "rails/sass-rails", branch: "5-0-stable" gem "turbolinks", "~> 5"
python	pip	www.pypi.org	requirements.txt	pkg1 pkg2>=1.0,<=2.0
Objective-C Swift	cathage	GitHub	catfile	github "ReactiveCocoa/ReactiveCocoa" >= 2.3.1 binary "https://my.domain.com/release/ MyFramework.json" ~> 2.3
	cocoapods	GitHub	podfile	pod 'Objection', '0.9' pod 'AFNetworking', :git => 'https://github.com/gowalla/AFNetworking.git', :tag => '0.7.0'
Go	内置	GitHub	在源代码文件中	import （ … "golang.org/x/crypto/ssh" "golang.org/x/crypto/ssh/agent" "github.com/pkg/sftp" ）

在企业内部使用这些工具来管理依赖，首先要解决的就是基础设施的问题，在构建的过程中要能够访问中央制品库并下载二进制文件。中央制品库位于互联网上，企业内部往往对访问互联网有所限制，所以不能很方便地直接访问中央制品库；同时，内部项目发布的一些二进制库又不希望被提交到中央制品库并暴露给企业外的开发者。所以企业必须投入成本来建设和维护企业内部交付件存储的“私服”，它既可以存储企业内部的交付件制品，也可以提供访问中央制品库的“代理”。主流的解决方案有Nexus和Artifactory，可以根据自己的需求选择相应的方案[1]。

1. 应对依赖“地狱”

在使用代码仓库管理源代码级别的依赖时，依赖“地狱”的问题并不十分明显。这个问题在二进制依赖管理机制中是无法回避的，由于传递依赖的存在（依赖的依赖），极易出现依赖冲突的状况。各种依赖管理工具为了解决依赖“地狱”各显神通，Gradle 在解析版本不一致的依赖时会采用就近原则，选择离自己最近的（传递层次最少的）版本；而 npm 遇到不同版本的依赖时，会把它们都加载起来且不会有冲突。

依赖管理工具不能彻底解决依赖“地狱”的问题，在遇到无法解析的冲突时依然得靠开发者自己来解决。对于企业内部的依赖，制定恰当的规则可以有效地避免冲突。无论是依赖开发者（提供方），还是依赖的使用者（消费方），都有义务在依赖的开发、发布、应用和维护的各个阶段遵循这些规则，这样才能解决该问题，保障整个生态圈的流畅运转。

依赖的开发和维护团队，必须严格按照语义化版本来命名版本号。如果主版本号不变，就必须保证对外暴露给其他团队使用的接口向下兼容。而这是由覆盖了公共接口的契约测试来保障的，并通过流水线不断地验证且给出反馈。如果契约测试不通过，开发团队就应该及时修复。如果依赖的新版本是破坏性的修改，则在开发过程中应该尽早发布测试版本以供使用者测试，收集反馈并根据反馈对依赖进行修改。使用者如果需要提前尝试依赖开发过程中发布的版本，则可以在依赖配置文件中使用包含这些测试版本的动态版本范围，不用频繁地修改依赖条目的版本号，从而可以不断地使用新的版本；也可以使用类似 SNAPSHOT 的版本号约定，使用滚动发布的测试版本。表 4-4 的最后一列列举了一些语言

1 Nexus 和 Artifactory 功能比较：https://www.jfrog.com/blog/artifactory-vs-nexus-integration- matrix

依赖动态版本的写法。

动态版本（范围或 SNAPSHOT）的依赖给开发过程带来了很大的便利，但在开发阶段结束并将产品发布后，动态版本的依赖反而会使产品变得脆弱；尤其是作为其他产品依赖发布的库。试想一下，产品 A 依赖库 B，而库 B 又依赖了另外一个库 C，其中 A 依赖的是 B 的确定版本 1.0.0，而 B 对 C 的依赖却是一个动态的版本范围 1.0.+。则对于 A 来说，B 的版本没有变，那么行为就不会变。而事实是，如果 C 不断发布 1.0.*的版本，则 A 的行为也可能会受到影响（要保证 C 的变化不破坏 A 的行为，几乎是不可能的）。如果 B 在发布确定版本时，其依赖的 C 的版本也能确定下来，则这个问题就不会存在，A 所依赖的 B 的行为会非常稳定，它自己的行为也就稳定了。

这就是健壮性原则（又称 Postel's Law），可概括为 10 个字："严格的发送，宽容的接收"，由 Jon Postel 在 TCP 早期的规范中提出。延伸到依赖管理领域，它意味着在闭门造车的开发阶段，其接收的依赖范围应该设置得宽松些（动态范围或者 SNAPSHOT），方便测试依赖的新特性；而到了被公开使用的发布阶段，它的依赖应该固化下来（固定版本号），避免依赖变化带来的不稳定性。这就要求产品在发布时，必须"锁定"其所有依赖（包括传递依赖）的版本号。

一定要杜绝在发布的产品中使用动态版本，这常常是由于不规范的手动发布造成的。好的实践应该是把发布的流程使用脚本实现，并由流水线来完成。一次完整的发布应该包括：锁定依赖的版本号、流水线验证通过、提升产品自己的版本号、提交代码仓库并打上标签。这里最复杂的就是如何锁定依赖的版本号，好在主流语言的依赖管理功能都提供了相应的工具来帮助开发者完成这个工作。

以 npm 为例，它提供了 shrinkwrap 功能来帮助 Node 开发者，只需执行执行 npm shrinkwrap，就可以在项目根目录得到一个 npm-shrinkwrap.json 文件，其中记录的就是锁定了版本的依赖条目。shrinkwrap 命令根据目前安装在 node_modules 上的文件情况锁定依赖的版本。在项目中执行 npm install 时，npm 会检查在根目录下有没有 npm-shrinkwrap.json 文件，如果这个 shrinkwrap 文件存在，则 npm 会使用它（而不是 package.json）来确定安装的各个包的版本号。这样一来，在安装时确定的所有版本信息会稳定地固化在 npm-shrinkwrap.json 文件里。不论是前面例子中的 A、B 还是 C 中的版本如何变化，或者它们的 package.json

文件如何修改，你始终能保证项目中执行 npm install 得到的版本号是固定的。

需要注意的是，使用版本锁定工具生成的这些锁定了版本号的配置文件，应该被放到代码仓库中管理起来，这样才能保证团队中的所有开发者使用的是一致的依赖。

新版本的功能开发完成后，就可以正式发布出去，并通知使用者版本发生了更新并提供新版本的 ChangeLog。这时开发维护团队可以使用发布分支策略，创建一个基于发布版本的分支，用于此版本的维护（修正发现的 Bug，但不增加新功能）。版本的维护周期由开发维护团队决定，超过此期限则不再维护此版本。而使用者收到新版本发布的消息后，应该在当前使用版本的维护期限到期之前及时更新依赖的新版本。后面会介绍这种发布分支策略。

使用者还可以更加主动地参与到依赖的开发中，在发现问题或者有新需要而开发、维护团队无法响应时，完全可以从依赖的版本仓库复刻一个新仓库，在新仓库上自行完成代码的修改（包括测试），并以一个专门标识（一般使用和原依赖不同的“标识”）的形式把二进制文件发布出来，供自己使用，同时向开发维护团队提出 Merge Request 或者 Pull Request，在经过开发维护团队的代码审查之后，由开发维护团队将这些修改合并到主仓库，包含在下一次发布的新版本中。待新版本发布之后，使用者再将专用依赖切换到新版本的“标识”。后面将介绍这种仓库复刻是如何鼓励团队间的协作及引导创新的。

2. 使用代码仓库存放依赖

在各种语言的依赖工具对比中还有一些有趣的依赖条目配置的写法，例如：

```
    pod 'AFNetworking', :git => 'https://github.com/gowalla/AFNetworking
.git', :tag => '0.7.0'
```

它是指这个依赖使用 Git 仓库中的一个标签对应的代码（除标签外还可以指定分支和一次具体的提交）。这种方式可以让使用者在依赖的某个版本被发布到制品库之前就可以进行集成调试，因为这些库并不会把每次提交都发布到制品库。脚本语言尤其适合这种方式，因为它们直接使用源代码，没有编译的消耗；从制品库上下载文件（源代码包）和从仓库里检出代码并没有效率上的区别。有不少的依赖管理软件（Carthage 和 Golarg）甚至没有中央制品库，直接利用分布式的代码仓库（GitHub）来代替制品库。

除了脚本语言，编译型的语言也会有相应的工具来支持这个特性。除了分布式的存储依赖带来的构建速度的提升，更重要的是使用这个特性在依赖版本正式发布之前可更早地和依赖集成。对于托管在GitHub上的Java项目来说，可以使用JitPack[1]服务来实现这个特性。下面是使用JitPack的Maven配置示例：

```
<repositories>
<repository>
    <id>jitpack.io</id>
          <url>https://jitpack.io</url>
</repository>
</repositories>
<dependency>
<groupId>com.github.User</groupId>
<artifactId>Repo</artifactId>
<version>Tag</version>
</dependency>
```

可以看到这里的配置使用了 GitHub 账号（groupId）和仓库名（artifactId）的约定来确定仓库的地址。当然，这个仓库中的源代码一定要符合 Maven 或者 Gradle 的目录结构约定，并提供了 pom.xml 或者 build.gradle 才能完成构建。

3. 流水线

二进制依赖和依赖它的产品一般是由独立的团队开发的，他们也会负责各自的流水线设计和维护。二进制依赖的开发流水线的成功以将通过验证（包括了两者之间的契约测试）的制品上传到制品库中为标志，也就是说在制品库中出现的文件都是经过验证的。如果产品使用的是依赖的二进制文件，则只要是从制品库中下载的，依赖的质量就应该有保障。而在开发过程中使用动态版本也会保证产品和依赖更紧密地集成。

如果使用了指向代码仓库中提交的依赖配置，那么应该保证其指向的提交、分支或者标签应该是经过流水线验证的。接下来将会介绍主干开发及其分支策略变种，在这些分支

1　https://jitpack.io/

策略中，在一些重要的分支上每一次提交都是经过审查和流水线验证的。在配置依赖时应该使用这些分支。

只要严格地使用流水线来验证代码提交并上传制品，在配置依赖时使用制品库中的文件或者代码仓库中经过验证的提交，并且在流水线中包括针对契约的验证，就可以保证两者无缝地集成。

4.3.4 小结

表 4-5 展示了二进制依赖管理与源代码依赖管理的对比，我们可以按照自身的需要选择合适的依赖管理方式。

可以看到，依赖和契约由一开始的代码级别，被逐步、清晰地分离并使用文件级别进行抽象。一个单块系统被拆分成了多个二进制文件组成的软件包。随着微服务的流行，单个软件包变成了跨进程的微服务集合。在这种情况下代码仓库又该如何管理呢？我们将在第 5 章中详细介绍。

表 4-5 二进制依赖管理与源代码依赖管理的对比

	二进制依赖管理	源代码代码依赖管理
构建速度	直接使用二进制文件，速度快	所有代码全量构建，速度慢
重构支持	不能同时修改依赖的代码和项目自身的代码，重构困难	可以同时修改项目自身和依赖的代码，重构简单
提交操作	不同的代码仓库在不同的工作空间下分别提交代码，管理简单，效率较高	在同一个工作空间下管理多个仓库代码，代码修改分别提交到不同的仓库，容易出错，效率较低
开发调试	调试时无法跟踪到源代码，即使发现依赖的问题，也无法修改	调试时可以看到源码，发现问题时也可以即时修改

续表

	二进制依赖管理	源代码代码依赖管理
响应变化	发现需求和问题时需要寻求依赖提供方的帮助，并放在下一次的开发计划中	可以直接修改并马上应用

4.4　分支策略

在使用代码管理工具时，使用最频繁的就是“分支”功能。比如开发一个新特性时、修改一个 Bug 时、开始一次代码重构时、一次新的发布流程开始时，等等，我们都可能会创建一个分支。创建分支的动机很简单：希望在接下来的一个时间段内，对代码仓库的所有修改都能够被分解成多个不同的任务并同时进行开发，每个任务的代码和提交记录都能聚焦在独立的任务上，互不干扰，特别是不会影响到代码仓库中已有的功能。下面是一位开发者在日常工作中使用分支的例子（依然以 Git 为例）。

假设这位开发者领到了一张故事卡，需要独立完成。这是一个有点复杂的任务，得花上几天完成。现在，代码仓库有一个主干分支（trunk/master，简称主干），团队中的每个成员各自修改的代码最后都要合并到这个分支上进行验证。首先，他（她）在本地的工作空间中同步主干上的最新提交，以此为基线开始工作。紧接着，他（她）为这次任务创建一个新的本地分支（只要没有被推送到代码仓库中，这个分支仅存在于他（她）自己的本地的工作空间中）。这个分支被命名为故事的名字，以方便理解。在接下来的几天中，他（她）每天都会增加一点和这个故事相关的代码，并提交到这个新的分支上。同时，为了避免最终整个故事完成后的新分支和主干冲突太多，他（她）还会每天拉取主干上的最新提交，把这些内容合并到自己的这个故事分支上。项目的模块化做得还不错，主干上的新提交几乎与这个故事代码没有交集，几乎没有出现任何冲突。终于，三天过去，这个故事开发完毕，所有代码都被提交到了本地的分支上，他（她）叫来 BA 和测试人员，在自己的工作空间里给他（她）们演示故事功能。大家都觉得这个故事完成得不错，他（她）再次把主干合并到这个分支上，改掉可能出现的冲突，并推送这个分支到远程仓库中。他（她）

发起一次由这个分支到主干的 Pull Request（简称 PR），邀请团队中的其他开发者来审查，同时会触发持续集成服务对这次 PR 进行自动验证。他（她）根据反馈（来自审查或者持续集成）进行重构或修改后，继续把代码提交到这个分支上，再次推送到远程仓库进行审查（推送之前还要把主干合并到这个分支上），并再次取得审查和持续集成的反馈。这次，在几位同事通过审查之后，分支被自动合并到主干上。至此，这次分支的使命完成。

4.4.1 主干开发分支策略

上面的例子描述的这位开发者的工作流程，就应用了我们经常说的主干开发分支策略（简称主干开发）。所有开发者的工作全部在一个分支上协作完成，并且努力确保这个分支上的每次提交都构建成功。我们把这个分支称为主干，代码修改可以直接提交到主干（但要进行提交前的代码审查）上，或者使用临时分支完成代码审查，最终必须合并回主干上，这些分支仅用于审查，从主干分离出来，存在时间很短。这种分支策略是达成持续集成或者更进一步的持续交付的关键要素，和这些实践相辅相成。只有保证频繁地提交开发者的代码到主干上，并保证在主干上提交构建成功，软件才能随时处于可交付状态。随着持续交付成为软件开发领域被认可的拥抱变化的方法论，主干开发也被越来越多的团队所采用。如图 4-13 所示。

主干开发的定义看起来很简单，有经验的开发者却知道这说起来容易做起来很难，在实际工作中要坚持单主干开发，需要有笃定的内心及各方面的协作，即使有不可抗拒的外力干扰，使我们不得不做出一定的让步，我们也得遵循一些原则，在这种分支策略上做出调整，并寻求平衡。

首先，我们来看看软件开发中代码管理的基础设施，包括在开发过程中使用的软硬件。选择正确的基础设施和软硬件也会为我们实施主干开发提供很多便利。

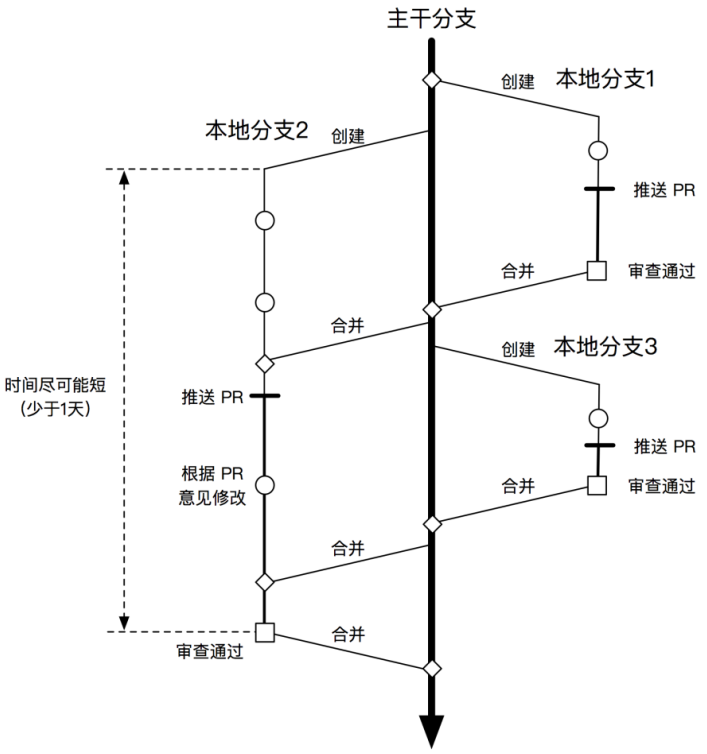

图 4-13　主干开发分支策略

4.4.1.1　主干开发对代码管理服务的要求

虽然主干开发模式对代码管理工具没有任何特殊要求，无论是分布式的还是集中式的代码管理工具都可以支持，但相关工具对一些特性的支持力度，还是会影响主干开发策略的执行效果。

首先，主干开发及持续集成要求开发者尽可能早地提交手中的工作，每天至少提交一次。这就意味着开发者会频繁地同步（拉取更新）其代码，也会频繁地提交（推送）其代码到代码仓库。这就对代码仓库的同步速度提出了要求，如果同步或提交一次代码需要数十分钟，则一定会成为这种策略的阻碍。我们在前面介绍了多种方式来对这些性能进行优化，包括建立本地的仓库镜像，使用“浅”复制和“疏”检出的方式处理代码，合理优化大文件的存储，等等。

其次，开发者会创建一些短周期的临时分支用于代码审查，这就需要他们经常在本地的工作空间上创建分支、切换分支和删除分支。SVN 仓库中的分支以一个独立“目录”的形式存在，通常会选择把不同的分支放在本地不同的目录下。切换分支后还需要变换本地的工作目录，有时会不太方便（比如要重新使用 IDE 打开工程，这需要一些时间）。而 Git 相反，每个分支实际上指向了一次提交，而每次提交都是一个完整的代码树，可以在一个目录下方便地切换不同的分支内容。另外，集中式代码管理软件创建分支和提交操作时是在远程仓库上进行的，远程提交的速度并没有分布式代码管理工具的本地提交速度快。

最后，开发者还需要频繁地在主干和临时分支之间进行合并操作，更智能的合并操作也会增加使用主干开发策略的信心。以 Git 为例，其所有分支的所有提交最后组成了一张 DAG（有向无环图），在任意两次提交、合并时可以轻易地追溯两条分支的祖先，如果它们之前进行过合并，则这次合并就只会处理上一次合并之后的差异，这让合并可以更加轻松地进行。早期版本的 SVN 则不同，在提交记录中不会保留合并信息，这样在同样两个分支之间连续地合并，每次都是全量进行的，后续的合并不能以前面的合并为基础进行增量操作，这样的体验并不友好。当然，SVN 从 1.5.0 版本开始，便已经记录合并信息了，所以请记得更新工具的版本。

4.4.1.2 主干开发对持续交付设施的要求

如此频繁的提交需要保证主干分支上的每次提交都是可以交付的潜在版本，这些都是靠代码审查和自动化验证来保障的。前面例子中的主干开发采用的是“提交”后代码“审查”的模式，只有通过代码审查和流水线验证的分支才会被合并到主干上。每次推送待审查的代码分支后，都会自动触发持续集成流水线，流水线会包括尽可能多的验证，包括但不限于编译、静态代码扫描、单元测试、集成测试，等等。流水线验证通过后，才会进行人工审查，人工审查通过后再合并到主干上。最后，合并后会再对主干进行一次自动化验证（出错后还得修改及再次触发流水线，如此反复）。流水线在整个过程中需要频繁地被触发，还必须足够快速地完成验证，才不会变成整个流程的瓶颈。如果流水线需要花上一个小时才能完成一次验证，那么这是让人无法忍受的。在极限编程实践中要求，整个自动化验证构建的过程不能超过 10 分钟，这就要依靠流水线背后强大的基础设施来保障。如图 4-14 所示。

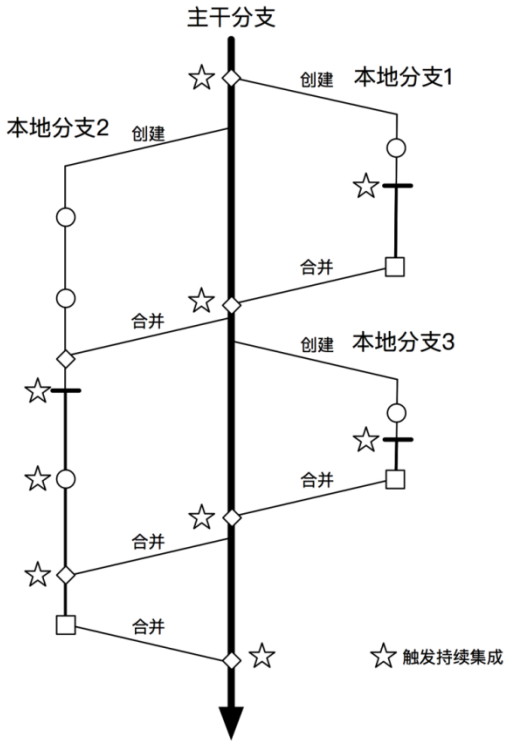

图 4-14　主干开发必须通过持续集成来保证质量

首先，流水线的执行机在性能和数量方面必须足够好。有些项目的构建需要强劲的机器性能，例如前面提到的 AOSP，构建时需要多核 CPU 和足够的内存。对于 iOS 项目来说，构建必须使用 Mac 才能完成。这都要求采购相匹配的硬件。由于同时会有一定数量的待审查分支存在，这也要求有足够的构建执行机待命，在需要时可以被调度且立即使用。而每个项目的流水线使用的工具往往各不相同，要维护这么多不同的构建环境也要耗费巨大的精力，即使可以维护，符合某种构建要求的执行机在数量上也有限。这就要使用虚拟化技术（包括虚拟机和容器）和云服务来解决这个问题，即动态地创建和分配资源，利用配置信息即代码和基础设施即代码的实践快速创建虚拟执行机。

其次，避免在流水线执行过程中使用的服务变成瓶颈。流水线包含很多步骤，而且是按顺序执行的，只有前面的步骤通过之后才能继续执行后面的步骤。有一些步骤除了依赖执行机的性能，还依赖其他服务的性能。比如，静态代码扫描的主流服务是 SONAR，在

扫描过程中会和这个服务有交互，下载规则和上传扫描结果；又比如，编译过程中会使用制品库服务下载依赖，在打包之后又上传到制品库，等等。这就要求 SONAR 服务和制品库服务拥有足够好的性能，能够支撑多个项目同时并发地访问。好在这些服务都有比较方便的扩展方式，应在选择服务时选择可扩展的服务，在遇到瓶颈之后对它们进行扩展。

对于大型企业和团队来说这尤其是挑战。这些企业出于各种各样的原因（共享资源、数据度量、统一流程、权限管控等），会组建专门的团队来统一搭建和维护这些设施，提供给数以百计的开发团队使用，并由这个团队做支持。这些服务的吞吐量也会成为制约持续集成服务效率的关键因素。笔者曾在某大型金融企业的研发团队中看到，整个开发部门有几千人及数百个项目使用一台单点服务器进行流水线调度，而大部分项目采用了定时构建的策略，这样每天下午 5 点左右，大量的构建任务集中爆发，产生了严重的阻塞。然而，这台服务器无法横向扩展，对服务进行纵向扩展带来的性能提升微乎其微，完全应付不了激增的需求。所以，对于大型企业和团队来说，在搭建这些平台时，就应该把这些服务的可扩展性作为必须考量的首要指标，对于那些不符合要求的服务应该尽早淘汰。

这个专门的支持团队也会成为另一个瓶颈。比如，这些企业会把部署环境和权限收紧（往往属于一个特殊的部门），哪怕对于在构建过程中使用的测试服务器，开发团队也会由于权限的管控无法直接访问。出现部署问题之后，开发团队没有权限查看日志，无法定位问题，只能依靠拥有权限的团队来解决。而有些企业会建立专门的团队来协助开发团队配置持续集成流水线，一方面由于部分配置的权限不会开放给开发团队；另一方面则由于这些自研或者采购的服务并不易用。一旦构建出了问题，这个支持团队就会变成瓶颈。要避免这个问题，最重要的是要区分支持团队和开发团队之间的权利和责任。随着近几年持续交付和 DevOps 不断被实践，开发团队的产出不再仅仅是一个软件的二进制制品，还要完成产品的部署和运维，因为部署和运维直接受业务的影响，也是产品不可分割的一部分。所以，环境的准备和维护已经变成了开发团队交付中的重要一环，开发团队必须有权限来完成这些操作。而持续集成流水线也渐渐升级为持续交付流水线乃至持续部署流水线，流水线的设计编排和环境准备也渐渐显露出和业务更紧密的联系，只有开发团队才了解其中的细节。所以，一切和产品业务强相关的任务都应该由开发团队自己完成，包括测试环境的准备、流水线的设计编排和配置，等等。而支持团队应该以技术专家的身份来帮助团队实施，而不是直接替代团队执行。提供的基础设施和服务也应该更多地向主流的平台或工

具倾斜，这样才能让开发团队有信心自己完成这些工作，减轻支持团队的压力。而支持团队除了肩负专家的身份，还肩负着维护整个基础设施平台的任务。只有把权利尽可能地交给开发团队，才能避免支持团队变成整个企业运作的瓶颈，让整个开发活动顺畅运行。

4.4.1.3　主干开发必须与其他敏捷实践结合

有了强有力的基础设施，只使用一个分支来开发的团队并不能称为实现了主干开发。软硬件设施只能帮助排除非人为因素的干扰，不能代替开发团队完成代码编写、测试和部署。这一切要开发团队自己完成，就还需要采用符合主干开发要求的实践。下面展示了一个需求从分析开始到最终部署上线的过程，让我们看看贯穿其中的那些实践。

要保证一个任务尽快完成并提交到主干上，这个任务就一定不能太大。任务的形式多种多样，有用户故事，也有技术研究，还有 Bug 修正，等等，无论是哪种形式，都是经过估算的、大小合适的任务。如果一个任务太大，则应该对它进行更进一步的分析，把它拆解成更小的任务来完成。在任务变得“尺寸”合适时，就可以在本地创建分支了。

接着我们可以开始编写代码了，在这个过程中除了编写实现代码，还得同时编写测试代码，并使用这些测试不断地在本地验证。在本地提交代码时，我们必须保证实现代码和测试代码一起提交，这样，才能使用这次提交的代码完成流水线的编译和测试，成为一次完整的验证。

自动化测试的策略也应该进行合理设计。我们经常会发现自动化功能测试变成整个流水线的瓶颈，因为它们依赖多、执行效率低，所以执行时间长，反馈慢。有些团队会有专门的自动化测试开发人员，完全依靠他们来开发大量的自动化功能测试，会使流水线的状况更加恶化。正确的方式应该是遵循自动化测试金字塔的原则，设定合理的分层测试体系，使用更多的效率更高的单元测试和集成测试来代替功能测试。单元测试和集成测试往往能在几分钟内执行完毕。

除了埋头开发，我们还应该注意到，主干上的代码也在不断更新，不断有其他任务代码被提交、合并到主干上。同时，在我们本地的工作分支上代码也在不断增加，久而久之（尽管整个本地分支的生命期不算长），两边的分歧会越来越大。如果两边不幸地修改到了同样文件中同样行数的代码，在最终合并时就会出现冲突。要避免大规模的合并冲突，我

们可以更频繁地从主干上把新的代码合并到本地分支上，把最后一次大规模的合并冲突分散到平时更新的小规模的冲突中解决，达到分而治之的目的。如果使用 A 通过 DAG 来组织提交记录的代码管理软件（比如 Git），则这样频繁的合并并不难。如果使用 Git，则还可以使用 rerere 功能来记录两个文件合并的细节，并在多次合并时使用记录的方法合并同样的文件。

在整个任务开发完毕并准备推送到仓库中开始代码审查时，应该非常小心谨慎。应该遵循 7 步提交法，保证在推送之前代码和最新的主干进行了合并，并且在本地经过完整的验证后再推送到远程仓库上。代码推送完毕就可以放松了吗？当然不是，我们还得关注流水线的结果，及时进行修正。对于主干上的提交来说，如果一次提交的构建结果是失败的，则在被修复之前，任何提交仍然是失败的，不能作为可以交付的版本。而对于供审查的分支来说，如果构建失败，则连审查的必要都没有，必须尽快修复。所以对于主干开发来说，流水线构建失败是优先级最高的任务，必须在最短的时间内修复。

在 PR 分支的流水线构建成功后，同事之间就可以互相审查代码了。如果一个 PR 在长时间内得不到审查，就不能合并，久而久之会被动地变成一个长生命周期的分支，最后有可能造成合并时的困难。所以团队内部应该建立代码及时审查机制，为开发者留出足够的时间，让他们有时间审查代码。如果在代码审查中发现了问题，则也需要及时修复，并再次进入审查流程，直到审查通过。

最后一步就是把 PR 的分支合并到主干上。为了让主干上的提交历史更整洁，在合并 PR 时可以采用不同的方式。不同的代码管理服务有不同的合并策略，以 GitHub 为例，默认的合并策略是使用--no-ff，这样一定会在主干上留下一条合并记录，强调这是一次有两个父提交记录的合并。如果想始终保持在主干上没有合并的记录，则可以在合并时使用 rebase 的策略。而如果 PR 分支上的提交记录很琐碎，则独立提交没有意义，只会干扰我们对主干上所提交记录的理解，这时可以选择在提交时把 PR 分支上的提交记录压缩一次（即 squash）。其他代码管理服务提供的合并选项大同小异。

除了在 PR 与合并的过程中要保留分支进行处理，在合并完成之后，这次 PR 分支的使命就结束了，就没有存在的必要了。为了避免后续错误地以这些分支为基础检出新的分支，同时避免这些分支的遗留产生噪音并干扰我们对这个提交历史的理解，它们应该在被

合并之后立即删除。

Gerrit 和其他 Git 的托管服务有一个显著的不同点。主流的托管服务都使用 GitHub 首创的 PR 来做代码审查（有的叫作 Merge Request，但本质上和 PR 一样），PR 是一个分支，在最终被合并之前，可以向这个分支提交多次。Gerrit 也使用一个本地分支来存放修改，但这个分支始终只有一次提交记录，后续的修改提交时会采用--amend 方式。这次提交会完整地表达出一个任务，而不会出现一些琐碎的带来噪音的提交记录，而且可以促使任务被分解得更小，以便放在一次提交中，也会使这次提交更快地被推送以获得审查。可以说这种方式更符合主干开发要求的小步提交和快速反馈。

综上所述，无论是基础设施的建设还是日常工作的最佳实践，都是为了满足主干分支的如下要求。

（1）主干上的每次提交都达到可以交付的质量标准。

（2）所有修改都要尽可能快地出现在主干上。

（3）除了主干，在远程仓库中不能出现其他长生命周期的分支。

摆脱了基础设施的限制后，制定日常实践规范都应该以上述要求为目标。这往往不是最困难的，最困难的是团队如何保持纪律性，不妥协地按照标准执行。代码管理软件本身很灵活，并没有把这些条条框框的限制内建在软件中，可以随心所欲地应用；另外即使规定了行为准则，也会出现忘记执行的情况。

主流代码管理软件都支持钩子（Hook）程序，可以通过强大的脚本来实现一些强制策略，强制执行一些代码管理行为或者对它们进行限制，帮助团队更好地落地最佳实践。钩子程序是在代码管理中特定事件发生前后，由代码管理软件触发执行的一些脚本程序。如果是在事件发生前触发的脚本，则甚至可以在其中阻止后续事件的发生。因为它使用脚本编写，所以可以借助服务器上的命令、工具及脚本语言（如 Python、Ruby）来完成强大的校验功能。

比如，我们可以利用服务器接收代码前的钩子，检查推送上来的提交进行验证，例如是否允许这些账号向某些分支（比如主干分支）推送。这可以用在主干开发中，强制所有

开发者先提出 PR，再由 PR 合并到主干上。推送到服务器上的提交都会自动触发持续集成服务以帮助执行验证。但如何保证在本地开发的过程中，开发编写了适量的单元测试并用它们验证过代码呢？这时可以利用客户端推送代码前的钩子，在代码推送前强制执行所有单元测试，只有测试通过并达到一定的覆盖率后，才允许推送。团队可以编写自己的脚本实现这些检查，钩子配置的方法请参考各代码管理软件的文档。

如果使用 GitHub 这类托管服务，则也会找到一些限制分支推送与合并检查的功能，不用自己编写钩子脚本就可以实现，更加方便。例如 GitHub 就支持“保护分支”（受保护的分支不允许强制推送、不允许删除，只有通过验证和审查的 PR 才能合并到该分支，等等；主干分支一般是受保护的分支）及“分支限制”（限制某些用户向某些分支推送代码）等功能。如果团队使用了其他托管服务（如 GitLab），则也可以找到相似的保护分支和分支限制的功能。

要把主干开发落到实地，最重要的是依靠团队所有成员对产品持续交付的认同，长时间按规范实践从而形成习惯，除此以外，别无他法。

4.4.2 应对并行开发

在实际项目中会出现这种情况：项目的交付件在不同的情况下是有区别的，通常以不同的变种表示，但这些变种的区别只属于产品的一部分，产品的绝大部分其他功能是一致的或者连续的，所以应该使用同一个代码库来生成不同版本的交付件。这时，团队往往会选择新建不同的分支来应付这些同时存在的变种，在每个分支上包含针对每个变种的性质各不相同的提交内容。这和单主干开发的策略相反，我们通常称之为多分支并行开发，简称为并行开发。常见的并行情况如下。

（1）开发中的新特性因为种种原因不能包含在产品的下一次发布中，比如特性的规模非常大，需要跨几个交付周期才能开发完毕，新特性是否包含在发布中需要在发布前的最后一刻再决定。为了避免干扰交付（交付毕竟是优先级最高的任务），团队会选择从主分支干上分裂出多个特性并行开发分支，在交付前来决定需要发布的功能，并把选中的这些特性（分支）集成到主干上，作为最终交付的产品的基线。这就是所谓的特性分支策略。

更有甚者，有些特性分支分裂出更多的“子”特性分支，变成层叠分支，如图 4-15 和图 4-16 所示。

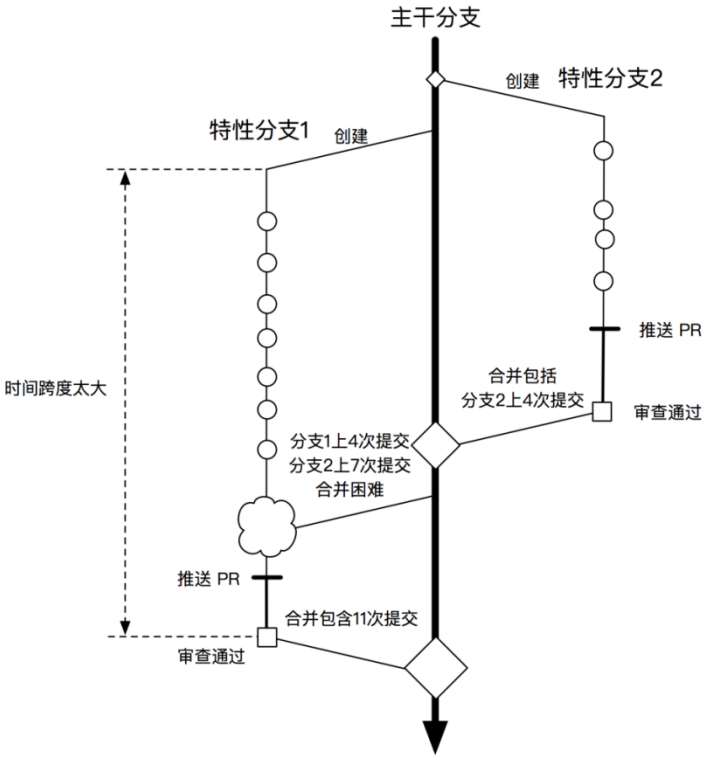

图 4-15　长期并行的特性分支引起大规模合并冲突

（2）同时存在的互不兼容的不同版本的交付件。这种情况多出现在交付件以需要安装或下载的二进制文件形式存在的项目中（比如操作系统），不兼容的版本意味着要重新安装整个产品，因此用户往往不会在新版本发布后及时升级到新版本。这样同时会存在一个或多个正在使用的老版本和正在开发的新版本。这些版本分别需要一个分支来维护，老版本的分支上包括的是 Bug 修复，而新版本分支上既包括 Bug 修复，也包括新特性。

（3）针对不同的硬件或操作系统提供的版本。这种情况一般出现在跨平台（硬件或操作系统）的大型软件中（比如浏览器），软件在发布时支持不同的平台，每个平台会具备相同的功能（或者有少量受限于平台差异的不同功能）。每个平台会分出一个长期存在的

分支，这个分支上包括针对这个平台的特性修改，以及跨平台部分的通用代码提交。

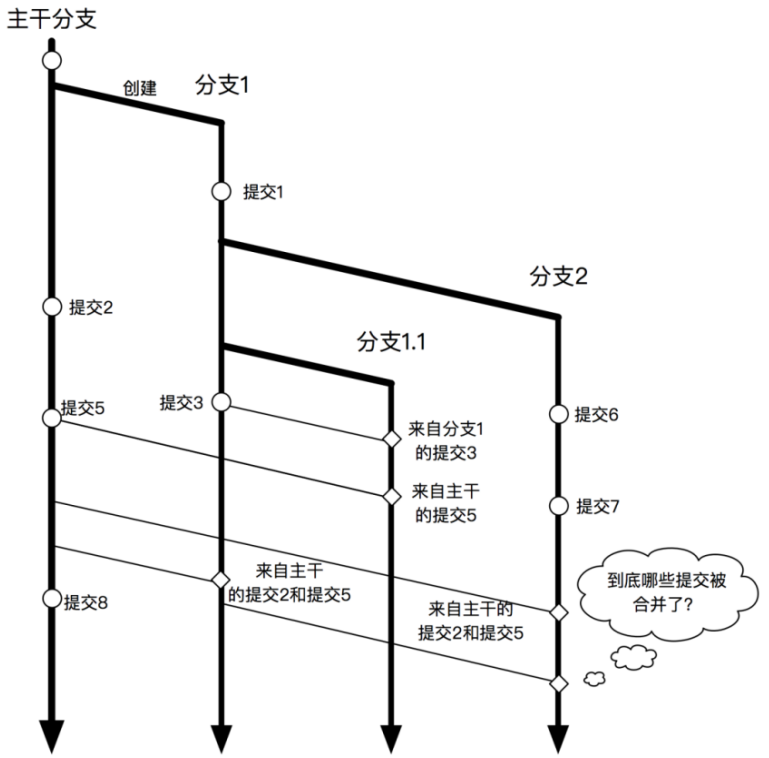

图 4-16　层叠式分支造成合并混乱

（4）提供给不同的用户使用的不同功能。这种情况可能出现在下游所依赖的接口或者服务中，不同的下游系统因为不同的业务需求，给上游依赖约定的契约也会存在少许差别；上游依赖的提供者需要同时维护这些有差别的契约接口或服务。如果产品有不同的功能需要试验，则也可以使用不同的交付件变种部署到部分服务器上来进行验证。不同功能的交付件变种会有独立的分支。

（5）大规模重构。对于一些老项目来说，由于开发时间较长，一定会存在一些过时的框架和设计，在产品开发到一定阶段后，需要使用新的框架和设计来代替，同时需要保持产品正常发布的节奏。由于过时的代码散落在代码中的不同位置，修改耗费时间，所以为了避免修改影响正常发布，会创建一个专门用于重构的分支。

可以看到在这些情况下，每个分支涉及的代码范围很大，需要投入较多的资源进行开发，因此会在代码仓库中长期存在，在多数情况下还需要多个开发者在这些分支上协作开发。由于创建分支实在是太简单、太容易了，开发者难以抵挡使用分支来解决并行问题的诱惑，尤其是在分布式代码管理软件渐渐变成主流选择之后。殊不知，创建分支只是整个分支策略的第一步，后面还有无数的“陷阱”等着团队去踩。逞一时之快创建的分支会随着时间的迁移造成让人出乎意料的困局。

长期存在的并行分支带来的最大危害就是无法做到持续集成。分支并行的时间太长，在与主干合并时会出现很多代码冲突。如果在一个分支中修改了函数名，但是在其他分支中大量使用了修改前的函数名，则会引入大量的编译错误，这被称为语义冲突（semantic conflict）。为了减少语义冲突，会尽量少做重构，而重构是持续改进代码质量的手段。如果在开发的过程中持续不断地出现功能分支，就会阻碍代码质量的改进。一旦代码库中存在了分支，就不再是真正的持续集成了。

除了和持续集成不搭配，多分支并行也会给团队协作带来困难。很难有一种方式阻止团队成员在合并代码时的随意行为，这会导致如同层叠分支策略中的混乱，我们根本不知道当前分支上到底合并了哪些分支上的哪些提交，而且长期的分支也间接地鼓励了开发者长期独立地在分支上工作，完全不考虑和其他开发者的沟通，不利于团队协作。

即使保持单主干开发，本地分支也是无法避免的：团队中的每个开发者都需要在他们自己的工作空间中保持一个正在开发的本地分支；团队中有多少个开发者在同时工作，就有多少个本地分支同时存在。但这些本地分支和上述这些场景中用到的分支有以下区别。

（1）其内容在推送前不会影响产品的集成和交付。

（2）存在的时间并不长，只有几天的生命周期，合并之后会被立即删除。

（3）产生的影响范围很小，只会集中在少数几个和任务有关的源代码文件中。

这种本地分支的规模小，范围也仅限于本地，处理起来较为轻松，但这种便利是以遵循主干开发的最佳实践为前提的。我们这里讨论的危害，是上述 5 种情况中出现的长期并行的分支造成的，它们天然就和持续集成的最佳实践有矛盾。如果我们使用本地分支的方法也违反了最佳实践（长期保留在本地不同步也不提交），则它也会变成危害（比如 GitFlow

中的 Feature 分支，稍后就会讨论到）。

并行的多个分支和持续交付就是不可调和的矛盾，分支的数量越多，交付的频率就会越低。所以必须想办法来减少同时并行的分支数量，那么这些需要“并行”的任务又该如何完成呢？经验告诉我们，并不是每次合并或更新都会出现复杂的冲突，甚至有时都不会出现冲突。如果代码结构设计得合理，模块也分割得十分合理，而各分支修改的模块又各自独立，则合并时就基本上不会遇到什么困难。所以我们处理并行的方法就是分而治之，对产品进行合理的模块划分，把并行时合并困难的代码与其他代码分开。这样可以在保持单主干的同时，支持多个任务的“并行”。

4.4.2.1 抽象“分支”

跨平台的应用要么是在不同的平台上提供统一的功能，要么是像多端应用一样，使同样的功能在不同的终端上提供不同的用户体验。这些产品的组成大概可以分成两部分，一部分满足不同的需求或受不同平台的限制，另一部分则提供通用的功能。良好的架构设计会在这两部分之间形成一个统一且独立的抽象层次，由这个统一的抽象接口来隐藏不同的实现。在构建或者部署时使用不同的实现代码，来和剩余的部分组成需要交付的产品。在现实项目中，我们可以找到不少这样的例子。

在微服务架构中，Sam Newman提出一种BFF[1]的解决方案，在一个通用的后端API基础上，为了满足不同客户端的用户体验要求，针对每一个客户端都构建一个独特的API。这个完整的应用就分成了这些模块：一个通用的后端API，多种不同的前端应用，以及多种前端需要的BFF。在微服务架构中，这些模块甚至被拆分到不同的代码仓库中。不同的前端BFF就是并行的多个版本，在部署时分别部署，互不影响。在不需要使用代码管理工具的分支上，这种跨平台的并行开发可以通过实现同一个抽象层的不同模块来同时开发。

在 AOSP 中，Android 操作系统在不同的硬件平台上需要使用不同的驱动程序，而应用开发者使用的却是同样的框架 API。Android 系统中的硬件抽象层（Hardware Abstraction Layer，HAL）是连接 Android Framework 与内核设备驱动的重要桥梁，其主要设计意图是向下屏蔽设备及其驱动的实现细节，向上为系统服务及 Framework 提供统一的设备访问接

1 Backend for Frontend，服务与前端的后端，http://www.infoq.com/cn/news/2016/01/bff-backend- frontend-pattern

口。一个特定硬件平台上的 Android 系统就是由不同的驱动程序及其实现的 HAL，和基于 HAL 完成的框架和应用组成的。这也是一种互不影响的并行，不同的硬件厂商只需开发各自的驱动程序，并暴露符合 HAL 要求的接口给上层即可。

这种方法被称为抽象“分支”，通过抽象层可以做到把产品中变化的部分和剩余的部分分开，而且让变化部分的不同实现以模块的方式共存，达到并行的目的，起到和代码仓库分支一样的作用。此外，它还可以促成良好的低耦合架构，使演进式的架构设计成为可能。如果有新的平台需要支持，则增加一个新的实现模块即可；如果有老旧的抽象层实现模块需要升级换代，则使用新框架或者库重新实现抽象接口，完成之后切换过去即可。这也使这种模式成为进行大规模的代码调整的重构手段之一。《持续交付》的作者Jez Humble曾在他的博客 [1]中介绍过他是如何在一个大规模的项目上把iBatis/Velocity/JsTemplate迁移到Hibernate/JRuby on Rails的。他总结出来的使用抽象“分支”进行重构的步骤如下，这里也加入了如何增加新平台的扩展内容。

（1）在你想改变的那部分代码之上创建一个抽象层，定义好实现者（提供方）和使用者（消费方）之间的契约。

（2）对其余部分的代码进行重构，使其使用这个抽象层下的代码提供的功能，通过使用重构手法把散落的修改点集中到一个“接缝”（抽象接口）中。

（3）在新的实现代码里实现一些新的类，让其上的抽象层根据需要，选择性地导向旧代码或新增的类上。

（4）如果是重构，则剔除原有的旧实现；如果是扩展新平台的支持，则保留原有平台的实现。

（5）清理并重复前两步，如果需要，则可同时交付软件，而这就要让脚本构建支持在不同抽象层中的实现可选择，比如使用依赖注入。

（6）如果是重构，则旧实现完全被代替后，如果你愿意，可以移除那个抽象层；如果是扩展新平台的支持，则保留抽象层。

1　https://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by- abstraction/

我们可以观察到在图 4-17 中，在第 5 步时，即便新的重构或者扩展还是半成品，它依然可以和主干分支合并，只要抽象“分支”的实现还没有切换到新的实现，原来的代码便依然可以交付。这里的关键就是切换的动作，它的切换决定了我们的交付件最终选择的功能（特性）。我们总结了 16 字来形象地描述使用抽象“分支”进行重构的过程：“旧的不变，新的创建，一步切换，旧的再见”。事实上，这个切换的开关是另外一个代替并行开发分支的模式，即特性开关，如图 4-18 所示。

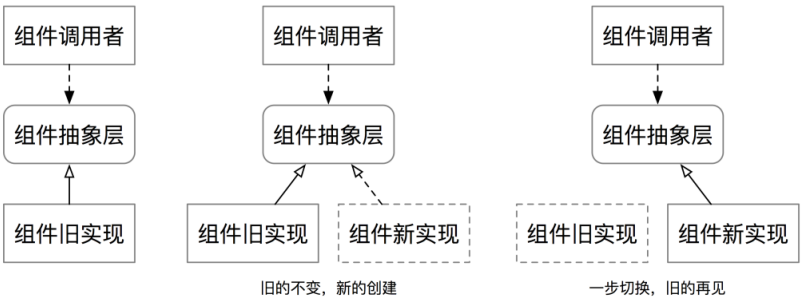

图 4-17　利用抽象分支替换旧实现

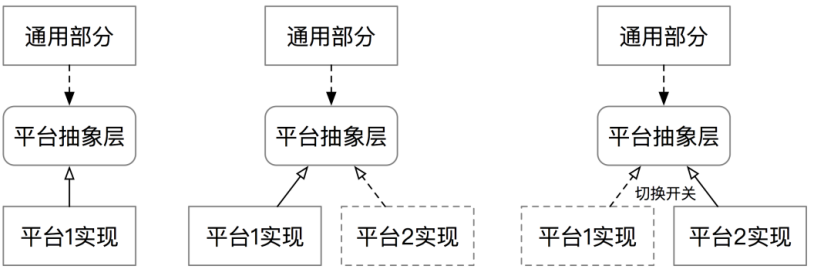

图 4-18　利用抽象“分支”和开关实现跨平台

4.4.2.2　特性开关

最常见的一个特性开关的例子就是在构建交付件时使用不同的参数构建出不同的版本（由不同的代码生成）。在 Android 应用开发中，Gradle 根据配置文件中设置的不同构建类型和不同产品风格生成不同的应用变体，比如收费版和免费版两种变体。

特性开关就是通过编译时或者运行时的不同配置，在不改变源代码的情况下，允许开

发者修改系统的行为。ThoughtWorks的Pete Hodgson把特性开关根据动态性（Dynamism）和生命周期长度（Longevity）的两个维度分成了 4 类 [1]，如图 4-19 所示。

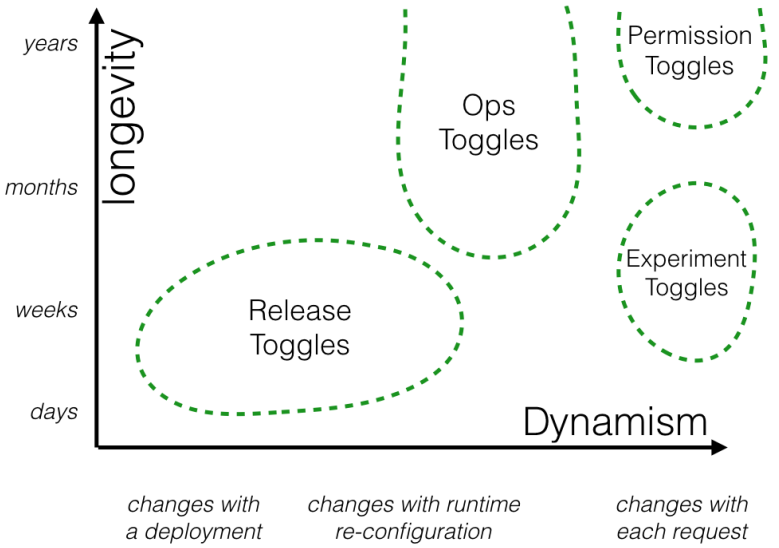

图 4-19　特性开关分类 [2]

（1）发布开关（Release Toggles）把特定的代码隔离出来，可以在发布时决定是否开启。这样产品经理就能以产品为中心，防止将半成品的产品特性暴露给终端用户。发布开关在发布时就静态地确定了状态（要么开，要么关），在上线稳定之后就应该删除，一般只存在一到两周。

（2）试验开关（Experiment Toggles）可以用于市场营销目的的 A/B 测试，通过评估不同用户群的群体行为，可以评价功能的价值。它也是短期的开关，但是在运行时由产品团队动态地打开或关闭。

（3）运维开关（Ops Toggles）在运维层面修改系统，可以用于在高负载的情况下优雅

1　https://martinfowler.com/articles/feature-toggles.html

2　https://martinfowler.com/articles/feature-toggles/chart-4.png

地停用某项服务。它是动态的，存在的时间较长。

（4）权限开关（Permission Toggles）主要是长期的可选特性开关，可以用于实现定价策略，比如实现一个付费模型，包括针对白银、黄金或铂金的不同产品层级。它长期存在于系统中。

前面提到的特性分支可以使用发布开关来替代。把与特性相关的代码使用合适的机制组织在一起，使用可以配置的开关把这些代码保护起来，可以打开或关闭，这样可以在需要时通过修改开关配置来控制特性的启用。在发布时由产品经理决定是否打开开关及上线该特性。这样在开发过程中半成品的特性依然可以提交，不会影响到已经开发完成的功能。

最简单的实现发布开关的方式就是使用 if/else 语句。把开关开闭情况下的行为分别放在条件分支的两个子语句中，在发布时设置 if 判断的值就可以了。设置条件的值有很多方式。Java 程序可以使用 System Property 来设置判断值，如果使用了依赖注入框架，则可以从配置文件中注入判断值。依赖注入也可以根据不同的条件来注入不同的行为。下面是一段使用 Spring Boot 的代码：

```
@RestController
@RequestMapping（"/foo"）
@ConditionalOnProperty（value = "feature.foo", havingValue = "true"）
public class FooController {
  @RequestMapping（""）
  public Map hello（） {
return Collections.singletonMap（"message", "hello foo!"）;
  }
}
```

这段代码使用了Spring Boot 中的注解 ConditionalOnProperty，只有在属性feature.foo的值为true时，整个Controller才会生效。利用依赖反转原则和依赖注入框架，可以很轻松地实现特性开关。除了利用设计模式和依赖反转，还可以利用很多在构建时通过参数来改变产品行为的方式，例如：代码预处理（C语言的宏及Java的注解处理器）、AOP（面向切面编程）、构建工具配置（Gradle的flavor）甚至动态的配置中心等。开发者可以选择合适

的方式来实现开关。有不少开发者总结了他们在实现特性开关过程中的经验，比如如何在项目中透明地引入特性开关[1]，如何使用特性开关更好地实现持续部署[2]，以及特性开关在iOS遗留系统重构[3]中的应用，等等。

还有一些发布开关的注意事项，如下所述。

（1）发布开关一旦稳定，就应该及时删除，避免无意义的开关积累太多而影响代码结构，增加理解难度。

（2）各个开关之间不要有依赖，要保持正交，避免相互依赖的开关过多地排列和组合并增加复杂度和测试的难度。

（3）分支开闭的行为都要有测试覆盖，都必须在流水线上验证，可以使用多个流水线步骤分别验证开关开闭的行为。

4.4.2.3　发布分支

如果团队的产品是可以“热”更新的，新的版本发布之后，下游系统和用户都被动地使用新版本，则我们可以在唯一的主干上不断向前滚动地发布。但是事与愿违，产品的不同变种基于不同的基线（提交），它们之间甚至是不兼容的（语义化版本中不同的主版本号），在这种情况下分支的并行是无法避免的。比如前面提到过，无论是操作系统、软件还是框架或库这类无法做到“热更新”的产品，只要还有用户使用着这个产品的老版本，就必然会出现多个版本需要同时维护的结果。这时我们可以使用主干分支策略的一个变种：发布分支。如图 4-20 所示。

1　http://www.infoq.com/cn/articles/introducing-characteristics-switch-in-project-transparently

2　http://www.infoq.com/cn/articles/function-switch-realize-better-continuous-implementations

3　http://www.infoq.com/cn/articles/ios-legacy-codebase-refactor

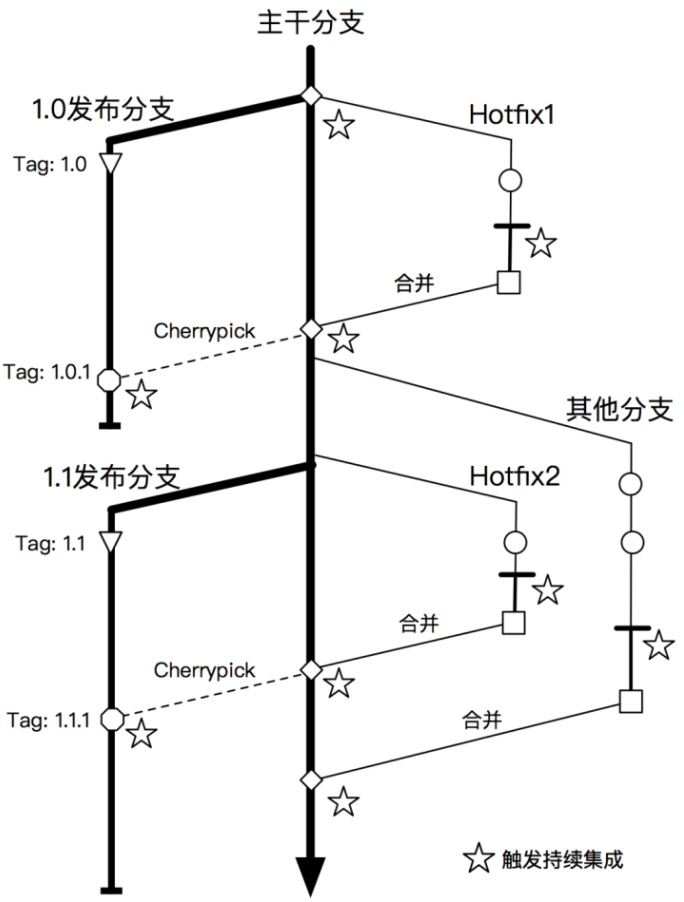

图 4-20　发布分支策略

采用发布分支的团队一般会有一个节奏固定且不那么频繁的发布周期，比如每个季度发布一次，在需要发布的时候，会在当前的主干上选择一次提交作为发布的基线，并从这里分裂出一个新的分支作为这次发布的候选版本，在这个分支上不会再有新的特性提交，所有提交都是关于 Bug 的修正（Hotfix）。当这个版本修正了所有问题并准备上线时，这次最新的提交需要打上一个标签。标签一般按照语义化版本的规则在上一次发布的版本号上递增（至少次版本号要递增，如果是破坏性的修改，则需要递增主版本号）。从这个发布分支分裂的那一刻开始，主干分支上的开发并不会被阻碍，依然按照自己的节奏向前滚动，不仅有新的特性不断增加，也会有问题不断得到修正。在发布分支上的正式版本发布

之后，就进入这个版本的运营阶段，在这个过程中也会不断有客户反馈问题，经过分拣，有一些问题需要在这个发布分支上修改。发布分支上的修改也会按节奏定期发布新的版本，或者有重大问题需要打破节奏来紧急发布，这些发布的版本号也需要按照语义化版本的规则来递增修订版本号。

可以看到发布分支上的提交都是修正，不会增加功能，也不会打破版本之间的兼容性。因为发布分支以主干分支的提交历史中的某一次提交为基线，即主干分支应该包括了发布分支上的所用功能的实现代码，所以在这些功能中发现的需要修正的问题也会在主干上重现并修改。反过来也是一样的，如果在主干上出现的问题也能在发布分支上出现，那么在主干上对这些问题的修正也必须提交到发布分支上。对于在两个分支上都能出现的问题，往往会先在其中一个分支上修改，然后把相同的内容提交到另一个分支上。那么，问题来了，到底先在哪个分支上修改呢？

几乎所有开发者都会条件反射地选择在发布分支上先修改，这种做法看似好处多多：发布分支上的干扰少，修改容易；修改提交并验证通过后，可以立即发布，非常快；修改提交可以通过合并的方式同步到主干分支上。但更推荐的做法是在主干上修正后再同步到发布分支上。经验告诉我们，在发布分支上出现的问题，我们会想办法快速地修复并上线，因为它的发布周期在下一次正式发布前。而对于时间还比较遥远的主干分支上的下一次发布，我们甚至会忽略它。这会导致严重的问题，这个隐藏的问题在下一次发布前回归测试时才会被发现，那时再修正已经太浪费时间了。而强制先在主干上重现并修正发布分支上出现的问题，再同步到发布分支上，可以帮助避免这个问题。注意，这里同步修正的方式应该使用 Cherrypick 而不是合并，因为在主干上可能已经有了其他不相干的代码，不能被合并到发布分支上。不用担心，Cherrypick 并不会比从发布分支到主干的合并复杂；这两种方式对代码的修改几乎是一致的，而且影响的范围很小。当然，对于那些无法在主干上重现的问题，只能在发布分支上修正。

把发布分支与从它上面分离出去的临时修正分支和其他分支分开来看，发布分支和从它上面分离出去的临时修正分支也是“主干”和分支的关系；每个发布分支都可以看作一个主干开发的分支模型。每个发布分支上的每次提交需要达到可以发布的标准，这些提交都应该是这个发布版本的修订的候选版本，它们也应该按照和主干开发中的主干一样的标

准来对待。也就是说这些提交也必须经过流水线的验证。

相比主干开发，现在我们增加了一些发布分支的流水线，需要更多的持续集成基础设施来支持。增加的分支也会增加流水线的配置和管理工作，如果使用老式的配置方式，则每次都要重复地为每个发布分支创建流水线，而且每个分支的代码不同，导致不同的构建和部署需求，因而产生不同的流水线配置，这些配置散落在持续集成的服务器配置中，更增加了理解和复用的难度，这就是所谓的雪花服务器现象。事实上，这些流水线的配置更应该和产品代码内聚在一起，它们也应该随着代码一起演进。同时，虚拟化和容器技术的流行也让流水线的配置可以用配置文件完成，在流水线上先执行动态的准备环境。继配置即代码（Configuration as Code）之后，流水线即代码（Pipeline as Code）也渐渐流行开来，以 Travis 为代表的持续集成服务已经成为 GitHub 生态圈中不可或缺的重要部分，它能和 PR 无缝地集成。而老牌的持续集成软件 Jenkins 也在 2016 年发布的 2.0 版本中正式支持了流水线即代码的特性。通过流水线即代码的实践，把流水线写在配置文件里，由持续集成服务根据和代码同步的流水线配置文件来执行流水线。把配置文件和代码放在一起，这样不同的分支就会对应合适的流水线。即使创建再多的分支也不用头疼于流水线的配置了。

随着时间的推移，版本号也会不断地增加，产品积累的历史版本就会越来越多，如果所有历史版本都需要无条件地维护，则作为上游的生产者（提供方）团队将很快会疲于应付，浪费大量的资源在维护工作上。下游的消费方当然会更相信经过自己验证的产品版本，对于新版本，他们并不会有足够的信心去升级，这对于他们是额外的成本。如何解决这个矛盾呢？只有双方都换位思考，做出一定的妥协，形成约定并遵守。

上游的产品生产者首先要保证每个版本的质量，严格遵守语义化版本的约定，不要随意破坏版本之间的兼容性，带给下游消费者足够的信心，让他们可以放心地升级新版本；其次，生产者还应该留给消费者足够的时间窗，在这个时间窗内，新版本和老版本共存，让消费者可以从容地安排迁移的计划；最后，要将老版本的维护结束时间告知消费者，让他们提前做好准备。Ubuntu首创了LTS[1]的概念，它是这样一种约定：Ubuntu每隔两年会发

1 Long-Term Support，长期支持版本，https://wiki.ubuntu.com/LTS

布LTS版本，维护周期为 5 年，在 5 年内，这个版本不会有结构性的修改，这些LTS版本也会有更多的测试，更稳定且兼容性更好。而且它们在很早就开始趋于稳定，会清楚地告知哪些特性包含在这次LTS版本中。Ubuntu LTS版本的用户很清楚当前使用的版本在什么时候会失去官方支持，也知道下一个LTS版本会有哪些功能及什么时候发布；如果决定升级，则他们可以在下一个LTS版本逐步稳定的过程中提前迁移。而处于下游的消费者也不能任性，也必须遵守游戏规则，做良好的守纪公民。最重要的是，在约定好的时间窗内，把依赖升级到持续维护的新版本。消费者也可以选择不升级，但同时必须接受出现问题无法得到支持的现实。我们的建议是：最好不要这样做，升级越晚困难会越大。积极的消费者会更紧密地与生产者合作，在新版本内部测试时就用上新功能，如果上游依赖产品是开源的，则甚至会参与到上游依赖的开发过程中。一个版本的发布分支在结束了维护周期后，就没有存在的必要了，可以被安全地删除。

4.4.2.4　仓库复刻

“复刻”是开源领域不算陌生的概念：开发者复制了一个开源软件包的源代码，并在此基础上进行独立的开发，最终形成一个独立的软件包时，极有可能伴随着产生一个新开发者社区或者“教派”。这种开源软件复刻版本和社区之间相爱相杀的故事数不胜数。最著名的例子就数 Linux 数量庞大的发行版形成的几个主流“教派”了：Debian、Fedora、openSUSE、Gentoo、Arch Linux，等等，开发者在选择时会根据自己的喜好来“站队”。随着开源软件不断地蓬勃发展及以 Git 为代表的分布式代码管理软件的流行，“复刻”项目的“分支”策略已经变成了向开源软件贡献代码的标准模式，这种方式更是被 GitHub 内建支持，成为独特的“社交化编程”现象，反过来更促成了这种方式的流行。这种“复刻”在本质上是一种分支，只不过这个分支存在于服务器上代码仓库的复刻中。在 6.4 节将会详细介绍在 GitHub 上参与开源软件贡献的步骤。

在大型企业中有数以千计的开发者和数以百计的产品，这些开发者也会组成一个庞大的开发者社区，而这些产品之间也会形成一个生态系统。可以将这种开源软件社区和产品的运作模式借鉴到企业内部，发挥它的作用。企业可以借助这种复刻的方式管理受信的代码仓库，引导团队之间的协作及鼓励创新。一些会使用到仓库复刻的场景如下。

（1）在守护核心组件的同时允许采纳创新。在企业中一定存在这样的组件，它们被多

个产品使用，是整个企业的核心，对它们的修改牵一发而动全身，需要经过严格的审查和验证，由核心开发者守护。而普通团队和开发者依然可以通过复刻这些核心组件的仓库进行创新，并使用 PR 向核心组件进行贡献。

（2） 组织协调不同部门之间的协作。同样是使用这些核心组件，不同的部门可能还有独特的定制化要求。它们可以各自维护自己的复刻版本，既可以从原始仓库中不断地得到修订和新特性，也可以在复刻的仓库中保持自己独特的功能。

（3）安全地和第三方团队交互。提供专门的复刻仓库给第三方团队（外包团队）可以定时审查并集成第三方团队的代码，可以隔离第三方团队的提交，保障主仓库的整洁。同时依然可以按照普通方式使用分布式代码管理软件，充分利用其优势。

（4）鼓励开发者创新。允许开发者使用个人的复刻来实现和验证创新，开发者可以长期保存自己的复刻仓库来不断地进行修改和实验，直到认为可以把这部分代码贡献回仓库。

关于中模式下的分支策略，可以独立地审视每个复刻仓库。假设上游仓库使用的是主干开发策略，则复刻仓库也会复制上游的主干开发策略，那么它自己也拥有一个主干和为了当前工作临时创建的分支，在这个复刻仓库进行独立开发时，也会使用主干开发的最佳实践。如果上游仓库使用的是发布分支策略，则复刻仓库也会复制上游的发布分支策略。复刻仓库中的改动可以根据不同的目的，在不同的分支上分裂出来。

而审视复刻仓库和它的上游仓库之间的关系时，应该只关注复刻仓库的分支（它自己的临时分支就是从这个分支分裂的，这个分支可以是主干，也可以是发布分支）和上游仓库中它的对应分支，它们之间是“分支”和“主干”的关系，需要遵循主干和分支之间协作的最佳实践：它们之间要进行频繁地同步。复刻仓库要不断地从上游仓库同步对应分支上的修改，而复刻仓库分支上的修改也要及时地向上游仓库的对应分支提交 PR 并积极地响应 PR 的反馈，让这些修改尽可能快地合并到上游仓库中。

对于复刻仓库和上游仓库之间的分歧（复刻仓库需要维护自己定制的功能），应该使用抽象“分支”的方法重构，并把共用的抽象接口部分和原来的实现合并回上游仓库。在下游仓库中保持抽象接口的定制实现。这样可以保证复刻仓库分支和上游仓库对应的分支一致，避免长期并行带来的合并难度。

可以发现，仓库复刻这种形式的分支策略主要用来解决团队间的协作问题，形成一个鼓励创新的开发者社区，这种模式甚至有了一个新的名字：内源（内部开源）。这种企业内部的关于代码仓库的管理和运作方式借鉴了开源社区生态圈的形式，在本书第 5 章和第 6 章中将会详细介绍。

4.4.3　定制分支策略

前面列出了一些需要运用分支或者其他代替分支的手段来解决的并行开发的问题，但每个产品都有它自己的特色，肯定会出现意料之外的特殊情况。团队可以按照自己的需要来设计属于自己的恰当的分支策略，包括分支定义、工作流程、纪律及工具脚本。分支策略要遵循以下原则。

（1）尽可能避免长期存在的分支。如果出现了需要并行的情况，则请一定先克制自己想要创建分支的冲动，先想一想能不能使用其他方法来解决。抽象分支、特性开关都是解决这个问题的手段。

（2）分支需要经常正确合并。无论是本地分支还是复刻仓库中的分支，分支从哪里分裂就要最终合并回哪里（或者删除），避免在无上下游关系的分支之间合并。下游分支要经常从上游分支更新代码，而下游分支也要及时合并回上游分支。

（3）每次提交都需要进行审查和验证。任何作为发布候选版本的分支上的每一次提交都要进行代码审查和流水线验证，通过 7 步提交法来保证提交的质量，也要提供足够的持续集成基础设施。

（4）使用仓库复刻来引导团队间的协作。使用复刻仓库来隔离不同部门或团队的协作，保证核心组件仓库提交的质量，同时允许部门、团队及个人有独立仓库进行创新并鼓励贡献。

如果团队使用的是Git，则该团队一定听说过GitFlow的分支策略，也许会采用这种流行的分支策略。GitFlow是基于Git的强大分支能力所构建的一套软件开发工作流，最

早由Vincent Driessen在 2010 年提出[1]。它把分支的使用发挥到了“极致”，使用了特性（feature）分支、发布（release）分支、修正（hotfix）分支、主干（master）分支及开发（develop）分支来处理不同的并行任务。这些分支又分为永久存在的两个主要分支：master分支和develop分支，以及除这两个分支外临时存在的支持分支，它们用于不同目的的任务。Vincent最早绘制的这份大图（见图 4-21）描述了这些分支之间的关系。

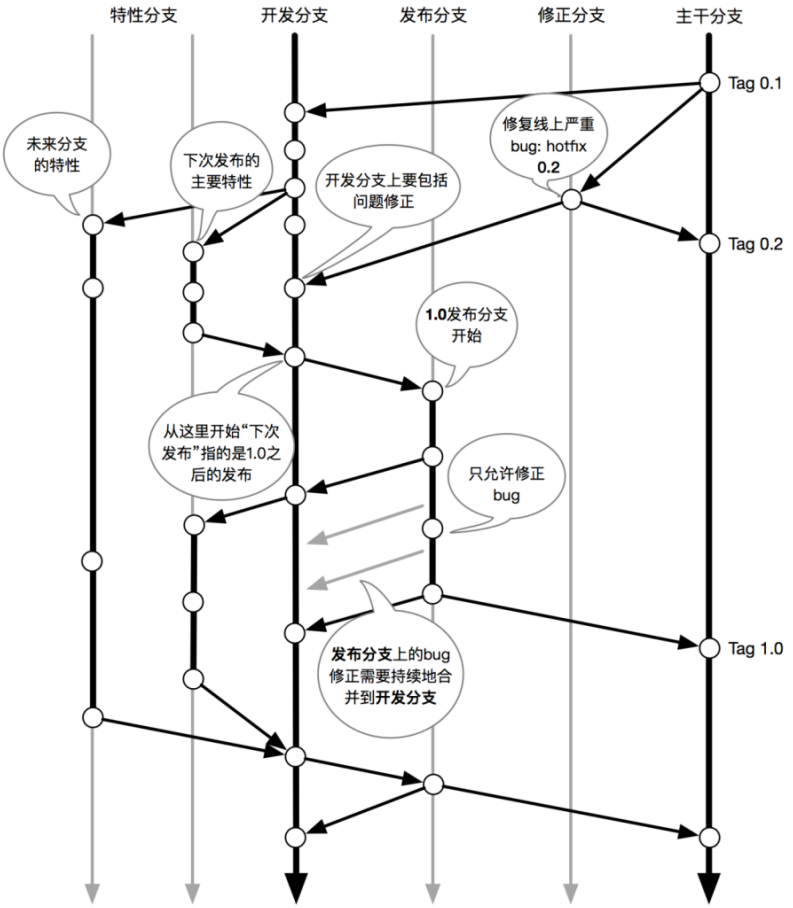

图 4-21　GitFlow的分支策略[2]

1　http://nvie.com/posts/a-successful-git-branching-model/

2　http://nvie.com/img/git-model@2x.png

GitFlow 虽然流行但备受争议，它有一些忽略了持续集成原则的地方。

（1）采用了长期独立存在的特性分支，也没有强调特性分支应该频繁地与 master 分支进行集成。

（2）长期存在的 master 分支没有实际的意义，这个分支保存的发布版本使用 tag 就足够了。

（3）hotfix 分支的修改要合并到多个分支，容易遗漏（特别是遗漏与 release 分支的合并）。

（4）如果没有支持流水线即代码（Pipeline as Code）的持续集成基础设施，则每个分支都需要配置和维护流水线，工作量大。

如果团队已经采用了 GitFlow 的分支策略，则可以思考一下上述问题，做出调整。还有其他一些分支策略供参考，比如 GitHub 的 GitHub Flow（和主干开发几乎一样）和 GitLab 的 GitLab Flow，在决定采用这些分支策略之前，需要使用这些原则检验一下，量体裁衣。

在笔者经历过的项目中，有不少企业看到了第三代码管理软件的优势（成本低、灵活度高），纷纷从传统的第一代和第二代代码管理软件迁移过来。但是，由于思维定式和使用习惯的桎梏，在迁移的过程中只注重形式，并没有从根本上转变管理的思路。前面提到的层叠分支策略就是笔者曾经遇到的一个案例。该团队把代码管理工具由 RTC 切换成了 SVN，但依然把 SVN 的分支当作 RTC 中的流来使用。而产品需求依然是按月排期的项目形式，这样就造成了多个月度的版本并行的状态，合并也没有制定规范，每次合并代码都是一次巨大的挑战。尽管第三代代码管理软件的功能十分强大，配置也很灵活，要模拟陈旧的分支策略并不是什么难事，但这种使用方式十分别扭，没有带来好处，反而打击了团队的积极性。

随着软件工程的发展，越来越多的开发工作变成代码并且代码可以越来越快地变成交付的软件产品。代码管理再也不是一个软件问题，它涉及了软件开发过程中的各个方面。老旧的代码管理软件跟不上时代进步且需要升级时，代码管理的实践也需要升级换代，在升级的过程中要考虑全面。

（1）基础设施和软件升级。即选择什么样的软件可以减少运维成本，提供足够的可扩

展性，团队成员也容易掌握。

（2）产品架构改造。即采用什么样的软件架构进行模块化重构，可以在复杂度不断攀升时把模块进行分仓库管理。

（3）依赖管理。即选择合适的依赖管理机制，制定版本号规则，搭建合适的制品库。

（4）分支策略。即团队使用哪种分支策略提高响应力和并行能力。

（5）团队协作。即代码仓库的权限如何设置能不阻碍协作，又能鼓励创新。

4.5 代码库热备份

由于分布式团队的成员可能处在不同的时区，全天 24 小时随时都有可能提交代码或者从服务器更新代码。为了不影响他们的开发工作，最好不要停止代码管理服务器。团队越大，成员越多，停止代码管理服务器带来的时间成本就越高。

对于传统的代码管理服务器，虽然有各种备份方案，比如基于文件夹级别的双机备份方案，或者基于服务器备份命令的方案等，但是这种服务器端的备份方案在大型项目中都存在一些问题需要解决，比如：如何保证备份数据的绝对一致性，如何高效地完成备份等。因此，在不少实际的大型项目中一般都会停掉代码管理服务器来做备份。而好的代码管理服务器需要提供数据安全的高效不停机热备份方案。

4.5.1 服务器端热备份方案

服务器端热备份是现在主流的备份方案，不同的代码管理服务器都有自己特定的工具来进行备份。如果没有自由备份和镜像工具，则也可以使用第三方文件的同步工具来进行备份和镜像。

服务器端备份有以下几种方式。

（1）使用代码服务器自带的工具进行代码热备份或者镜像，比如 Gerrit 自带的插件 replication 等都可以进行热备份和镜像。

（2）使用第三方文件的同步工具自动把服务器上的代码仓库目录同步到镜像服务器中，比如 rsyncd。

（3）自己编写脚本对服务器目录进行打包备份。

第 2 种方案和第 3 种方案在不停机的情况下进行备份存在数据不一致性的问题，所以不建议作为热备份方案，只有第 1 种方案可以作为热备份方案。

4.5.2　客户端热备份方案

除了服务器端热备份，在分布式代码管理工具时代使用客户端热备份代码仓库也是一种临时选择，比如使用 Git 通过客户端进行热备份能备份代码和所有提交的历史记录。但是客户端热备份不能备份服务器端的权限配置等服务器信息。如果只关注代码或者代码提交历史，且还没有好的服务器端热备份方案，则建议使用这种快速、低成本的客户端备份方式。

4.6　案例：Android 定制化系统开发

4.6.1　项目背景

全球通信是国内的通信业巨头，几年前因为决策转型慢了一拍，错过了手机发展的黄金时代。现在看着竞争对手在市场上风生水起，靠着基于 Android 的定制手机激活了新业务，全球通信想奋起直追，但无奈于市场大局已定。

与此同时，Android 通过几年的发展，已经慢慢衍生到消费者生活中的其他领域，例如平板、路由器、车载设备等。特别是电视机顶盒，作为各独角兽公司全力争夺的下一个

互联网入口，目前已经呈现出群雄逐鹿的混战状态。全球通信在传统机顶盒市场领域经营多年，有深厚的技术背景，积累了不错的市场口碑，于是决定发力机顶盒市场。但全球通信深知 Android 移植技术是自己的短板，经过慎重考察与调研，移动优先公司进入其视野。

移动优先公司提供专业的 Android 整体解决方案，包括 Android 的移植与定制。同时，移动优先公司有着强大的 UX 设计能力，推出了市场上颇有口碑的几款应用，其中一款还获得了 2016 年 Google Play 最佳应用。经过对移动优先公司的反复考察，特别是在参观了移动优先公司的研发基地之后，其敏捷的工作方式、持续集成的实践、高水准的设计能力给全球通信公司留下了深刻的印象。全球通信立即决定与移动优先公司合作，共同组建研发团队，开拓机顶盒市场。

4.6.2 项目及其代码管理介绍

两家公司的联合研发团队建立伊始，就确定了整个产品的四个目标。

（1）稳定性，保障 7×24 小时不间断运行，不黑屏，不死机。

（2）用户体验，对 Android 系统 UI 进行深度定制，和同质产品拉开差距。

（3）兼容性，借助 Android 的生态圈，通过第三方应用和游戏提高产品的可玩性。

（4）内建的核心应用包括启动器、应用商店、播放器等。

Android 是由操作系统、中间层与应用层组成的完整平台，每个层次划分清晰，业务领域与开发技术栈有着明显的区别。

（1）Linux 内核和 HAL 硬件抽象层主要使用 C 语言完成硬件驱动 HAL 层接口的适配，需要驱动开发和内核开发相关的知识。

（2）原生库与 Dalvik 虚拟机和 Framework 主要使用 C 语言开发 JNI 接口，使用 Java 语言开发 Framework。

（3）应用程序层，主要使用 Android Framework API 开发业务应用，使用 Java 语言或者 C 语言。

很快，来自全球通信和移动优先公司的 100 多名开发和测试工程师组成了产品的开发团队。按照不同的技术背景和擅长领域，他们很快找到了各自的工作重点。

全球通信的团队成员来自原先的机顶盒开发团队，他们多年同硬件与芯片打交道，对 Linux 内核和驱动开发非常了解。他们负责 AOSP 的移植与相关 HAL 硬件抽象层的开发。

移动优先公司的团队成员分为两部分：有 Framework 开发与定制经验的开发者负责 Android 系统的定制与用户体验提升；剩下的 App 开发团队成员负责核心应用的开发。

同时，移动优先公司还抽调了持续集成经验丰富的开发人员，专门支持所有小组成果的集成、分支策略的制定及持续集成基础设施的建设。

联合开发团队最终划分了以下小组。

（1）内核小组

（2）音频驱动开发小组

（3）Framework 定制小组

（4）启动器 App（HomeUI）小组

（5）应用商店 App 小组

（6）播放器 App 小组

（7）支持小组（负责基础设施建设）

很快，支持小组接到了第 1 个任务：立即准备好代码仓库。问题马上来了：全球通信位于南部沿海，而移动优先公司研发基地位于西部内陆，团队也分布在这两个城市；产品基于庞大的 AOSP（Android Open Source Project）开发，完整的项目需要占用近 20GB 的磁盘空间。这样规模的分布式团队在这样大规模的代码库上工作，谁也不想在每次提交前花上数分钟同步代码。如何保证代码检出（clone）、同步、推送和审查（review）的效率，减少对团队的影响呢？

Google 给 AOSP 的二次开发提供了一套完整流程和工具支持。联合开发团队决定遵循

AOSP 项目的指导，先建立起可工作的流程和设施，在日后的工作中再逐步优化。支持小组马上投入工作，很快就搭建起两套 AOSP 代码仓库镜像：一套部署于全球通信内部，另一套则部署于移动优先公司研发基地。其中，以全球通信内的镜像为主，以移动优先公司内的镜像为从。

主镜像顾名思义，是两套代码仓库中更重要的那一个镜像，它承载着整个产品的稳定 codeline。它需要合并来自各方面的提交，如下所述。

（1）从上游 AOSP codeline 同步最新的修改。

（2）接收并合并移动优先公司从仓库镜像 codeline 发来的提交。

主镜像同时扮演了全球通信内部的私有仓库镜像，位于南方的内核小组与音频驱动开发小组从主仓库上选取了部分相应的代码子仓库作为工作的起点。随着研发工作的推进，这些代码仓库的提交也会最终汇入稳定的 codeline。

最终产品的每一次构建、集成与烧录都来自主镜像的稳定 codeline。接下来的代码审查持续集成工作将重点围绕这个 codeline 展开。

镜像只有一个作用，就是主镜像在移动优先公司内部的复刻。

（1）从上游主镜像 codeline 同步最新的修改。

（2）移动优先公司内部的私有镜像。

Framework 定制与其他 App 小组在从仓库镜像上选取或创建了相应的代码仓库，作为工作的起点。这些小组的工作成果被推送到从仓库镜像，通过代码审查后最终进入主仓库镜像的稳定 codeline。

代码进入主仓库的流程也参考 AOSP 的代码审查流程，支持小组搭建了 Gerrit 服务器和 Jenkins 构建服务。由来自各个开发小组的技术 Lead 进行代码审查工作。

图 4-22 中，左侧是 AOSP 的 codeline，中间是主镜像仓库的 codeline，右侧是从镜象仓库的 codeline。

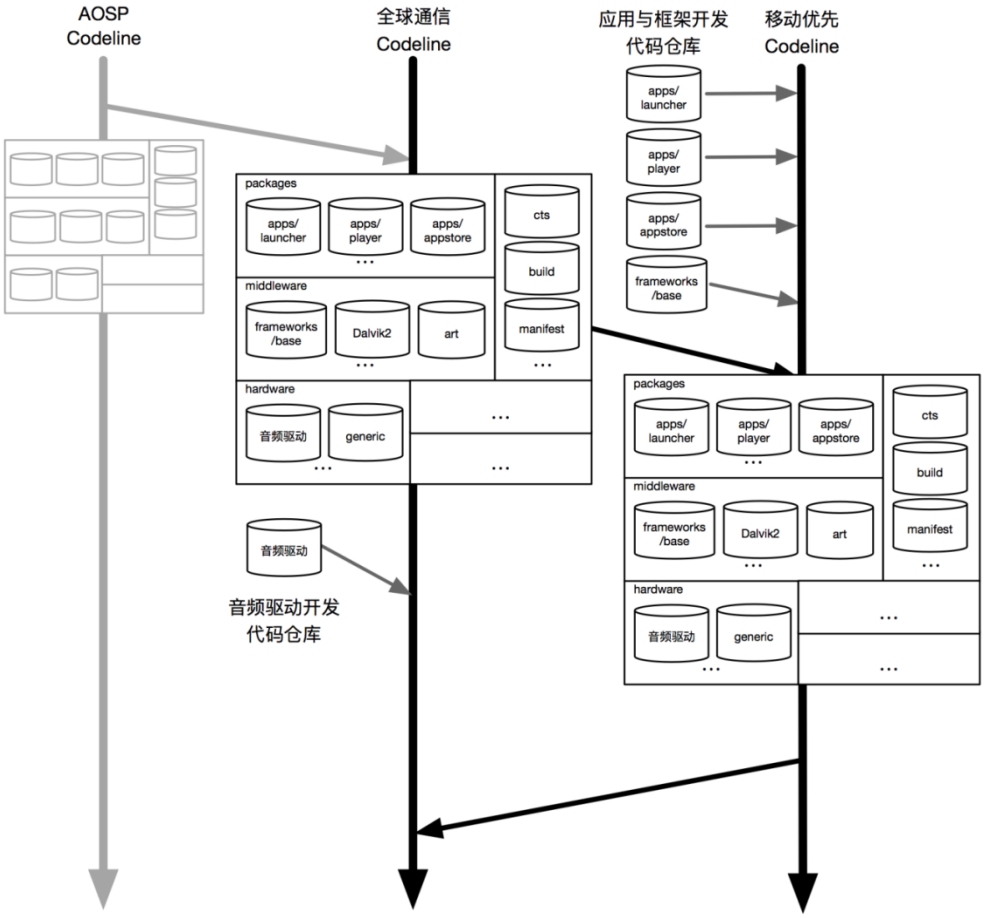

图 4-22　Android 产品代码库架构

4.6.3　分支策略

本项目很复杂，涉及的 Git 仓库和团队的数量众多，其关系比较复杂，导致其分支策略也很复杂，没有单一的分支策略可以支持所有团队的工作。根据分支策略的制定原则，支持小组定义了主干分支（新建一个当前产品的分支，并设置好清单文件，例如：新的音频设备驱动、播放器的仓库地址），用于把不同开发小组的开发成果集成到主干和发布分支上（每半年进行一次交付，每次交付有单独的一个版本号，从主干分支上分裂出来），

来进行固定节奏的交付。这就是前面介绍的发布分支策略（参见图 4-20）。所有主干和发布分支上的提交都必须通过 Gerrit 代码审查和持续集成验证。

但是由于不同的开发小组有不同的特性，比如开发人员数量不同、办公地点不同、代码复杂度不同、集成和测试的方式不同，不同的开发小组还需要根据自己的情况来定制自己的代码分支策略，所以支持小组没有强制每个开发小组都必须遵循特定的分支策略进行开发，而是推荐了不同的上下文适用的分支策略方法，例如：单功能开发分支策略、多功能开发分支策略等。支持小组也推荐各个开发小组配置各自的清单文件，只单独检出它们独自的子仓库，来提高代码仓库的访问效率。

4.6.3.1 单功能开发分支策略

单功能开发分支策略类似于主干开发策略（见图 4-13），是这个项目中的一种最为简单的小组分支策略，主要用于简单特性开发小组。由于这样的小组维护的代码量不大，人数也不多，新开发的特性很小，可以很快地完成，所以使用单功能开发分支就可以很好地运作。

音频驱动开发小组主要是维护一套成熟的音频库及音频设备驱动。由于音频库已经十分成熟，所以修改不会很多。但是对于不同的音频设备需要不同的驱动，所以针对每个不同的音频设备（单功能）都会有一个主开发分支，这就是单功能开发分支。

在这个单功能开发分支上，所有开发任务都是独立的，所有开发人员都统一在这个分支上进行代码提交和解决代码冲突，不需要创建额外的功能开发分支。但是如果开发人员想创建自己的临时开发分支，就只能在本地创建，绝不允许推送到服务器端的代码库中，从而保证服务器端的代码库中一直只有一个功能开发分支。当每个开发人员需要提交代码时，必须先从这个功能开发分支上拉取代码并解决冲突（如果存在冲突），然后推送代码。如果对每个代码提交都需要进行自动化集成和测试，则在 CI 服务器上也只需要搭建一条固定的功能开发分支流水线。

在本 Android 项目中，只有少部分开发小组选用了这种开发分支策略，比如音频、HomeUI 等开发人员不多且代码修改量不大的小组。

4.6.3.2　自定义并行分支策略

自定义并行分支策略适用于本项目中的复杂特性开发小组，单个复杂特性的体量较大，需要多种技术栈支持（需要不同技术背景的开发人员），不能做到快速开发完成。比如一个相对独立的新功能需要 3 个人（位于不同的办公室）开发超过 3 天，这时就需要针对这个新功能的开发工作建立一个独立的功能开发分支来集成这 3 个人提交的代码，从而隔离它与其他分支在开发过程中的相互影响。在这个新功能开发完毕后就需要合并回主开发分支，从而方便其他分支进行集成。

虽然不得以使用了类似特性分支（见图 4-15）的反模式，出现了分支的并行，但开发团队遵循按照分支策略的原则，尽可能做到快速提交与合并：每个功能开发分支都需要每天定时从主开发分支上合并代码到自己的功能开发分支上，从而尽快并且尽早集成其他分支已完成并提交到主开发分支上的代码，如果和主干上的其他提交有冲突，就能在第一时间解决冲突。在这个项目中，播放器开发小组就采用了这种分支策略，同时支持小组给予巨大的支持，他们增加了一台构建服务器，专门支持这类并行分支的持续集成。

除了以上两种典型的开发分支，有些小组为了适应自己项目组的一些特殊情况，对以上两种经典开发分支策略进行了一些少量的修改和定制化，但是并不会使用 GitFlow 那么复杂的分支策略。由于其修改和定制化不具有通用性，所以这里就不再赘述了。

4.7　多产品线

随着产品的客户越来越多，不同的客户会提出很多特殊需求，需要给不同的客户开发特定的功能。随着这些特定的功能越来越多，开发团队通过对客户及客户需求的分类，发现需要基于多产品线进行开发才能在客户需求和持续开发维护两者之间找到平衡点，这时多产品线也就提上了日程。

4.7.1 多产品线介绍

多产品线实际上是同一类产品，由同一个产品演化而来。多产品线的大部分功能都是相同的或者类似的，但有些功能由于不同的客户需求或者不同的业务需求而有所不同，导致它们在代码层面有不少代码是不同的。对于这样一类的产品，鉴于它们的相似性，为了节省开发和管理成本，一般不会把它们作为完全独立的产品来进行独立开发、独立代码管理和独立流水线集成，而是通过一些特定的方法、模型和流程来对这类产品进行统一的代码管理和集成等。那些大型、成功的且差异化客户较多的项目才需要多产品线进行开发和管理。

4.7.2 多产品线开发的困境

一个项目在进行多产品线开发时，需要解决在单产品线开发中各种没有遇到过的问题。

（1）如何管理多产品线之间相同和不同的代码？究竟是使用一个分支，还是使用独立的分支或者使用独立的代码库来管理不同产品的代码？

（2）是否需要同步不同产品线之间的相同代码？究竟是直接在不同产品线的代码之间进行代码合并，还是通过一个主干分支进行同步？

（3）如果需要同步不同产品线之间的代码更改，那么如何解决它们之间的代码冲突？

（4）在代码集成流水线上如何实现和管理多产品线的集成和构建？

为了解决这些问题，需要运用一些特定的技术或者方法，而且根据不同团队的技术能力和工作方式，使用不同的多产品线解决方案。

4.7.3 多产品线解决方案

前面提到，现实中在一般情况下只有大型的、差异化客户众多的项目才可能需要多产品线解决方案。尽管在这种项目中大部分开发人员都不会关注多产品线的代码管理，但是

如果开发人员能深入理解多产品线的代码解决方案，就可以在设计软件架构、模块划分或者代码的目录结构时多思考怎么配合多产品线的管理，项目在做多产品线时就能更好、更快地实施。在大量真实的项目中常用的多产品线解决方案有 4 种：基于多代码目录；基于同一代码目录；基于多代码分支；基于代码开关。

4.7.3.1　基于多代码目录的多产品线

基于多代码目录的多产品线解决方案是一种最为传统的解决方案。这种方案对每个不同的产品都建立一个独立的代码目录，基于每个代码目录开发每个独立的产品。为了维护大量的相同代码，还需要建立一个主干代码目录用于管理相同的代码，其经典工作模型如下（见图 4-23）。

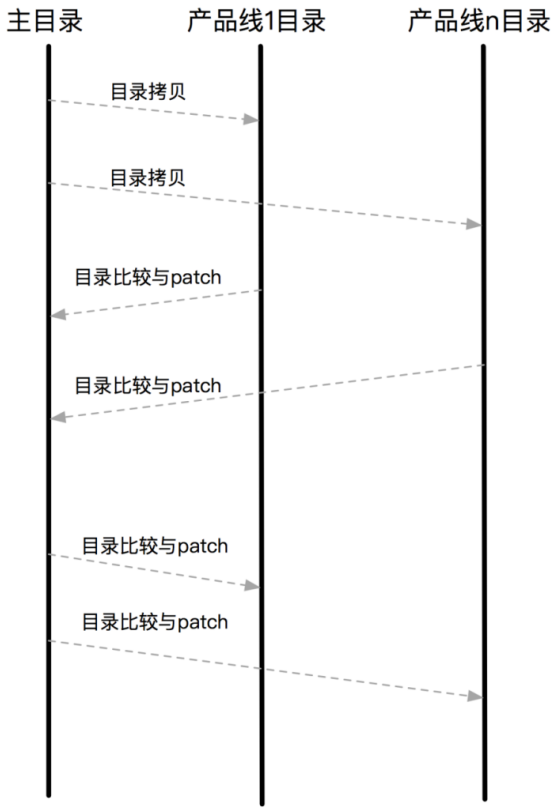

图 4-23　基于多代码目录的多产品线

（1）首先建立一个主干代码目录，可以是一个包含所有通用功能的代码库，也可以是基于一个已有的功能最多的产品线版本的代码库。（因为一般只有当一个产品基本成熟且有多个客户之后才有可能采用多产品线。对于全新的项目很少会在一开始就计划使用成本更高的多产品线的管理方案。如果是一个全新的项目，则建议先采用成本更低的基于同一代码目录或者基于特性开关的解决方案）。

（2）通过复制整个主干代码目录来生成另一个新产品的代码目录，每增加一个产品线就需要从主干代码目录进行复制（由于 Subverison 的分支本质上就是目录复制，所以 Subversion 这类代码工具使用分支来管理多产品线的方式也就属于这个模型）。

（3）开发人员在新产品代码目录中进行开发，并定期审查。如果在审查中判定新开发出来的代码是通用的功能或者修正，那么需要将这些功能或修正的代码合并到主代码目录中（合并的方法一般是通过主干代码目录和新产品代码目录之间的差异对比，定制化生成 patch，然后把 patch 应用到主干代码目录中。）

（4）新产品的开发人员需要定期查看主干代码目录，如果发现有新的功能代码或者 bug 修正，就需要将其合并入新产品代码中。

（5）不断重复步骤 3 和步骤 4。

这种解决方案是最容易被人理解和实施的，但是它的最大问题是长时间实施之后，不同产品线的代码之间的差异越来越大，这也会导致不同产品与主干代码之间的差异越来越大，因此在代码合并时会出现大量的冲突。由于合并是基于目录差异对比完成的，面对大量的冲突缺乏有效的合并工具，长时间后会导致主干代码目录和新产品代码目录之间的代码合并越来越困难，成本过高、难以为继。为了解决这些问题，出现了基于第三代代码管理工具如 Git 的解决方案：基于多代码分支的多产品线。

4.7.3.2 基于多代码分支的多产品线

由于绝大多数的第一代和第二代代码管理工具的分支都是基于不同的代码目录实现的，所以当需要在不同的分支之间进行跟踪、比较和合并等工作时，由于缺乏高效的工具支持，多分支代码的管理工作会很难进行。因此使用第三代代码管理工具（比如 Git）中

的同目录轻量分支，以及高效的跟踪、比较和合并工具，可以部分解决多代码目录解决方案中的代码合并困难的问题，其经典工作模型如下（见图 4-24 和图 4-25）。

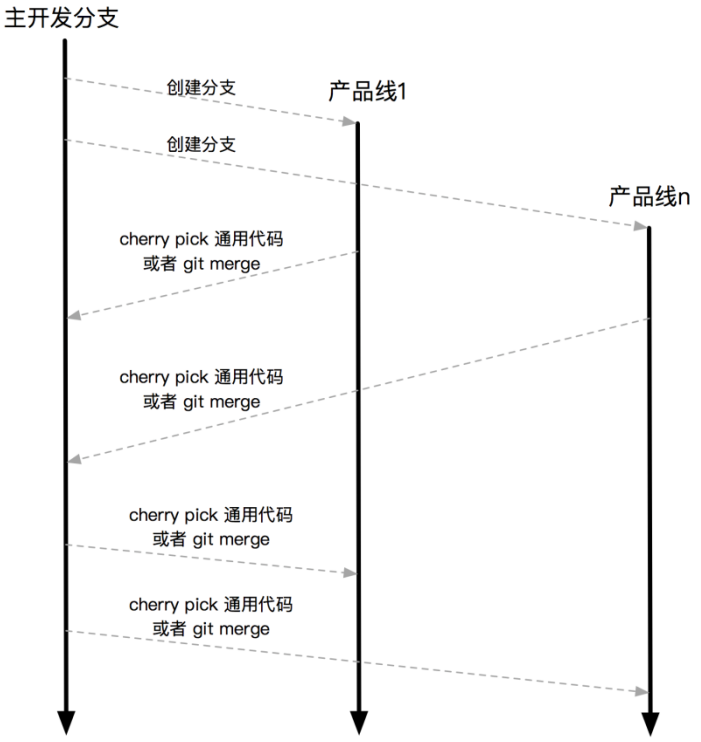

图 4-24　基于多代码分支的多产品线 1

（1）首先建立一个可以直接使用一个包含所有通用功能的代码分支的主干代码分支，或者从一个已有的功能最多的代码分支中拉出来。

（2）通过这个主干分支直接拉出一个新的分支作为新产品的代码库，有多个新产品就创建多个分支，每个产品线对应一个主分支（另外，通过 GitHub 提供的代码库复刻方式创建的新产品代码库虽然属于独立的代码库，但是它提供了有效代码的跟踪、比较和合并方式，所以也属于这个模型）。

（3）开发人员基于新产品代码分支进行开发，并定期审查。如果审查中判定新开发出来的代码是通用的功能或者修正，就需要将这个代码合并到主干分支上（如果使用的是

Git，那么合并的方法就包括高效的 git cherry-pick、git merge 等，可以使用它的分支跟踪图来进行可视化产品分支跟踪）。

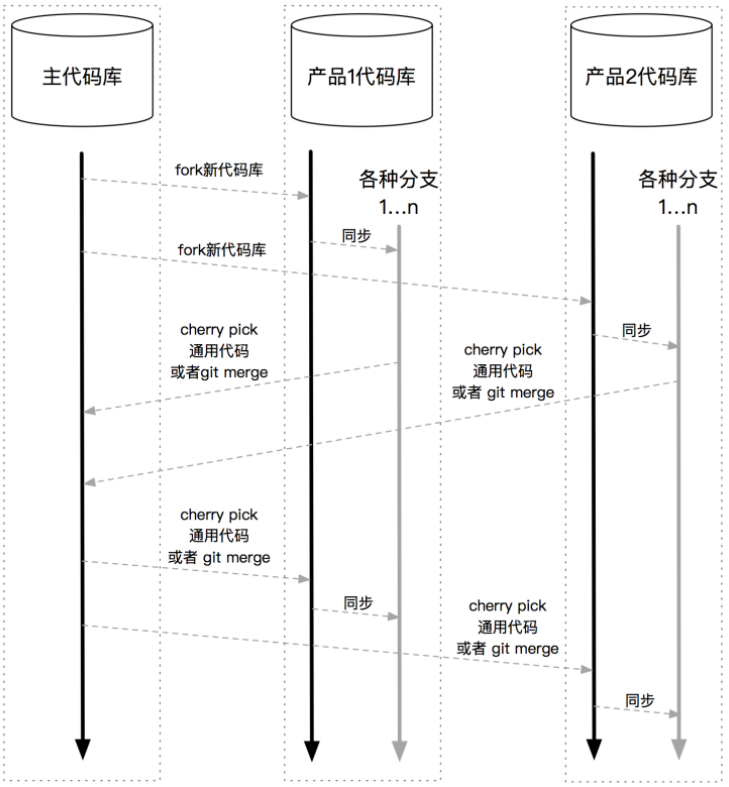

图 4-25　基于多代码分支的多产品线 2

（4）新产品的开发人员需要定期审查主干代码分支，如果发现有新的功能代码或者修正，就需要将其合并入新产品代码分支中（审查的时候也可以使用第三代代码管理工具所提供的分支对比功能进行高效对比）。

（5）不断重复步骤 3 和步骤 4。

这种解决方案虽然解决了上一种方案中关于产品之间代码跟踪、比较和合并低效的问题，但仍然存在一些局限和不足：第三代代码管理工具和代码管理系统的学习曲线和成本较高，而且实施时容易出错，需要有经验的人实施。

但是对于如何正确审核和合并需要的代码的核心问题，无论是基于多代码目录的方案还是基于多代码分支的方案都无法有效解决，需要有经验和有能力的开发人员来解决。如果想减少审核代码和合并代码的工作量，则下一个基于多产品子目录的解决方案能有效地解决这个问题，却是靠牺牲代码的可读性和可维护性来达到目的的。

4.7.3.3　基于多产品子目录的多产品线

基于多产品子目录的多产品线解决方案是一种非常特别的方案，因为它需要团队自己开发或者定制产品的构建工具，用这个特定的构建工具才能实现该多产品线解决方案。它不依赖于特定的代码管理工具或者管理系统，所以基于任何一种代码管理工具的项目都可以使用它，其经典工作模型如下（见图 4-26）。

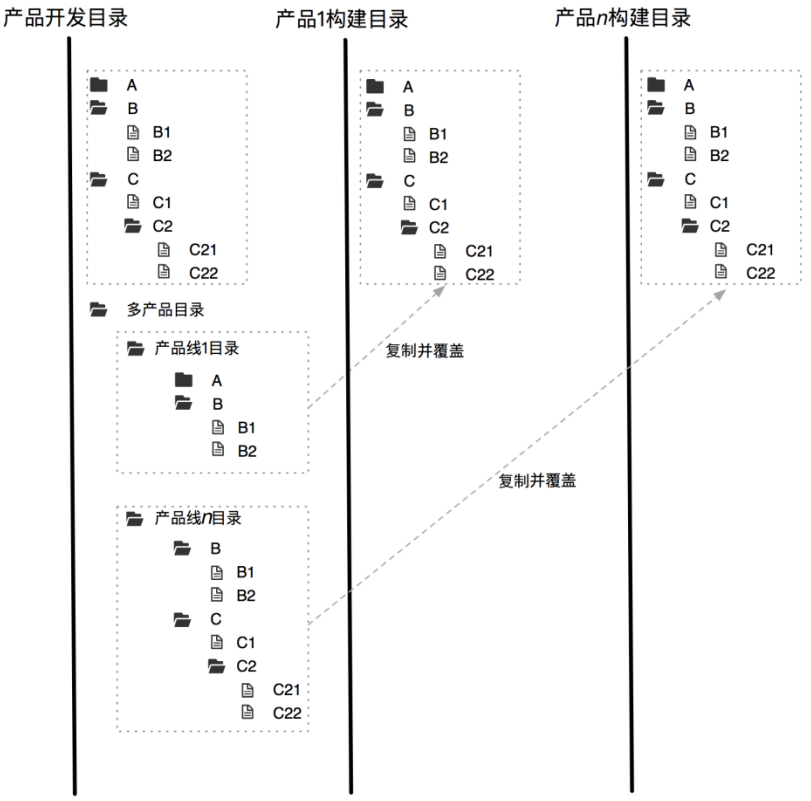

图 4-26　基于多产品子目录的多产品线

（1）选取主干代码作为工作开发的唯一目录。

（2）在代码主目录下创建一个多产品子目录，并在此目录下再为每个产品创建一个独立的子目录，并在这些独立的子目录里按照和主干目录一致的结构存放和开发产品特有的代码文件。

（3）重新开发或者修改构建工具，使得整个构建流程在开始构建之前加入一个预构建步骤。第 1 步，动态地创建一个空的临时构建目录，并将整个主干代码中除多产品目录外的所有文件直接复制到下面。第 2 步，根据从命令行输入的某个产品参数动态地将这个产品在多产品目录下面的文件按照相同的目录结构也复制到临时构建目录下，并覆盖临时构建目录下的相同文件。第 3 步，开始在临时构建目录中执行和构建流程。最后，将构建成功的生成物复制到指定的生成目录下。

（4）开发人员在正式开发功能代码时，需要思考并与其他产品团队进行沟通，以确定新的功能代码是通用的还是自己的产品特有的。将通用的功能代码放到主干代码目录树下面，而将产品特有的代码放到多产品子目录下面，从而避免代码合并的问题。如果要对主干通用的代码进行修改，则需要及时和其他团队进行沟通。

虽然这种方案解决了多分支或者多目录之间的代码合并问题，但是它的不足与局限也是十分突出的，以至于很多大型项目不愿意用这种解决方案。它的不足和局限如下。

（1）由于所有产品的特有代码都放在同一个主干代码目录下面，并且每个产品子目录中的产品代码文件一般都在主干代码目录中有对应的名字相同、内容相似的“占位”文件，所以在阅读代码时非常困难。

（2）产品子目录下面的产品代码越多，管理成本就越高。比如如果在某个产品子目录下面开发了一种通用功能，若要合并入主干代码中，就需要改变代码的设计，将通用功能移到另外一个没有在任何产品子目录中被定制过的主干代码文件中。如果无法避免在某一个产品子目录中定制过这个文件，那么仍然需要进行代码合并。随着产品定制代码的增多，改变代码设计和代码合并的情况可能会随之增多。但是只要有足够好的代码架构和设计，就可以避免大量的设计改变和合并工作。

如果不愿意使用多产品子目录的方式来管理多产品线，只想用一个主干代码目录，也不想用分支系统，则仍然有一种解决方案可以实现多产品线，那就是使用代码开关。

4.7.3.4　基于代码开关的多产品线

基于代码开关的多产品线解决方案是 4 种方案里对技术（包括代码架构、设计及编写代码）和团队沟通要求最高的一种解决方案。它和 Feature Toggle（特性开关）是相同的原理和实现方法，所以也可以叫作 Product Toggle（产品开关）。Product Toggle 和 Feature Toggle 唯一的不同就是面对的对象不一样，Product Toggle 是面向产品级别的，而 Feature Toggle 是面向同一个产品里功能级别的。但是它们的实现技术都是使用代码级别的预编译技术在构建前生成只包含特定产品开关的代码用于构建，其原理和基于多产品子目录的解决方案有异曲同工之妙。其经典工作模型如下。

（1）基于项目的技术栈选取开关库。

（2）选定产品的开关并写好相关注释或者多产品线开发文档，以便团队里的其他开发人员和其他产品线的开发人员阅读。这点非常重要，这个注释或者文档一定要描述清楚。

（3）开发人员基于这个开关来开发自己的产品特有的功能代码，然后使用自己的产品开关进行构建。如果需要对通用代码进行修改，则请及时和其他团队进行沟通。

由于这种解决方案对代码架构、设计和团队沟通的要求非常高，所以在真实的大型项目中很少被用到，其具体的局限性和要求如下。

（1）如果一定要在一个已经成熟的大型项目上使用 Product Toggle，那么很有可能需要调整整个软件的架构用于适应 Product Toggle，成本会很高。

（2）需要对整个代码架构进行很好的设计，不然开关越加越多就难以维护了，而且不同产品之间的代码冲突也会越来越多。

（3）在开关多了之后，代码阅读和新加入代码开发的困难度就越来越大，开发人员必须对自己负责的产品所使用的全部开关有整体的认识和理解。

（4）团队沟通必须非常高效，否则整个多产品线的开关和代码管理就会越来越混乱。

4.8 超大型分布式团队

除了中大型分布式团队，还有一种例外：超大型分布式团队或者大型独角兽公司，比如Google、Facebook和Microsoft等。它们的代码量是超级巨大的，比如在 2015 年Google的中心代码库中的文件总数就超过了 10 亿个，有 86TB的数据，代码文件总数超过 9 百万个，而代码行超过了 20 亿行 [1]，并且有超过 25000 名软件工程师在这个代码库上进行软件开发工作。

两位Google工程师在《美国计算机学会通讯》上发表了一篇论文*Why Google Stores Billions of Lines of Code in a Single Repository*[2]，介绍了Google为什么采用一个定制的大型单体中心代码库，并且在多个大会上分享了这个话题。互联网上还有更多的讨论 [3]，InfoQ中文网站也发表了一篇较为客观的文章“Google为什么要把数十亿行代码放到一个库中？”[4]来评论Google这种代码管理方法 ，其中总结了Google宣称的这种唯一中心库代码管理方式的优势。

◎ 统一版本控制。

◎ 广泛地进行代码共享和重用。

◎ 简化依赖管理，避免菱形依赖。

◎ 原子修改。

◎ 大规模重构。

◎ 跨团队协作。

◎ 灵活的团队边界和代码所有权。

---

1 https://www.youtube.com/watch?v=W71BTkUbdqE

2 http://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

3 https://news.ycombinator.com/item?id=11991479

4 http://www.infoq.com/cn/news/2016/07/Google-why-10

◎　代码可见性及清晰的树形结构提供了隐含的团队命名空间。

并且总结了 Google 这种唯一中心库代码管理方式的一些问题。

◎　工具投入（Google 开发了自己专用的 Eclipse ID 插件）。

◎　代码库复杂性（需要有依赖重构和代码清理辅助工具）。

◎　代码健康（专用工具可以自动检测和删除无用代码、分派代码评审任务等）。

对于 Google 这样的大型团队或者公司，它们的代码管理方式看起来是简单的单体代码库管理方式，其实背后并不简单，需要大量的额外投入来辅助管理，因为这是在各种前提和限制条件下的历史产物，所以其中最为重要的两点如下。

◎　当前大部分商业和开源代码管理工具或者系统在管理一个超过 10 亿个文件、20 亿行代码的中心库时效率都十分低下，而且随时都有大量的代码同步（包括代码获取和提交）请求。一般情况下，为了高效地管理如此海量的代码，同时不影响程序员的日程工作效率，这样的团队或者公司都会开发或者定制化自己专用的代码管理工具和系统，比如Google开发的Piper[1]、Facebook定制化的Mercurial[2]和Microsoft定制化的Git系统GVFS等。

◎　大型公司一般经过长时间的积累才有如此大量的代码，并且都有自己特定的经历和原因，比如开发了大量定制化的外围辅助工具和系统，形成了特有的一套代码管理模型和流程。更换这种大型代码库的管理工具的成本非常高，何况现实中很难找到一个代码管理系统能满足已有的管理和流程需求，所以一般情况下都不会更换。比如 Google 一开始使用 Perforce 来管理其单体中心代码库，后来发现它无法支持其巨大的代码量，所以开发了 Piper 来管理中心库管理，并且在代码审查上投入了大量的成本，比如开发了专用的工具来自动检测和删除无用的代码、分派代码评审任务等。虽然 Google 也尝试过向 Git 进行迁移，却最终由于文化和

1　http://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

2　https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/

工作流程的巨大变更而放弃了，但是仍然对于一些新的实验性的或者一些开源的项目尝试使用一些新的代码管理工具。

虽然Google的大部分核心代码都是使用Piper在一个中心代码库中进行管理和维护的，但是它仍然有不少开源项目，其中包括Android Open Source Project（2008）和Chromium（在 2014 年转向Git）这样的大型项目[1]，或者创新的初始项目依然可以选择使用Git这样的开源代码管理工具进行代码管理。所以应该给予项目组足够的权利去选择适合自己项目的代码管理工具，从而让团队感受到足够的尊重和动力。

而世界范围内像 Google 和 Microsoft 等有财力和物力去开发或者定制一款适合自己的专用代码管理及其周边辅助工具的公司是很少的，绝大多数公司只适合通过购买商用的或者开源、免费的或者基于云的代码管理系统来管理自己的代码。

由于是选择单体代码库还是分布式代码库直接影响了团队对于代码管理工具的选择和使用，所以一些正在快速增长或者需要转型的中小型公司就对代码管理方式和代码管理工具的选择产生了疑惑，是应该学习 Google 的核心代码库而继续使用单体代码库的管理方式，然后自己开发和定制化自有的代码管理工具，还是学习 Linux、Android 及 OpenStack 等开源项目从而转向分布式代码管理方式和免费的分布式代码管理工具，还是直接使用基于云端的代码管理系统呢？

因此，我们总结了一个代码管理工具选择四象限图来帮助中小型公司选择代码管理方式和代码管理工具，如图 4-27 所示。

其中，资源主要包括资金和人力资源，而技术是指项目组或者公司里的大部分工程师的技术能力。通过这个四象限图，中小型公司就可以通过另一个角度去思考和判断自己应该选用什么样的代码管理方式和代码管理工具。而对于大型软件公司，比如类似于 Google、Facebook、Microsoft 等这样规模的公司就不适合用这个四象限模型，而是需要根据自身的具体情况自己开发或者定制代码管理工具，可以是中心服务器式的，也可以是分布式的，无论是什么形式，只要适合自己的实际情况就可以了。

1 https://www.youtube.com/watch?v=cY34mr71ky8

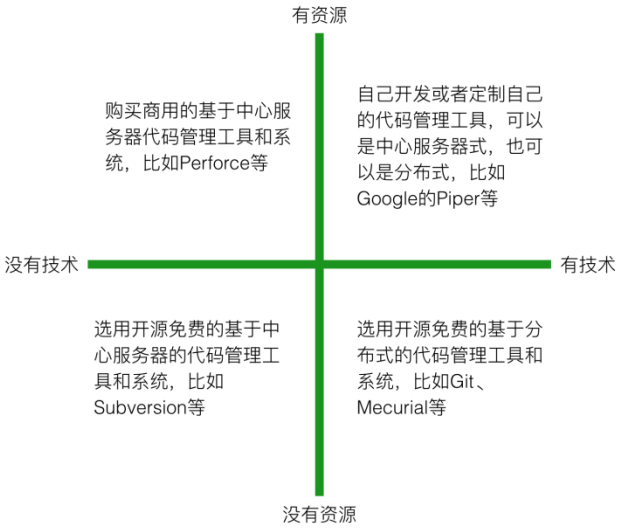

图 4-27　代码管理工具四象限

第 3 部分

发展与未来

第 5 章

云时代微服务大型分布式团队

5.1 云时代和微服务架构

微服务架构（Microservices）是在 2014 年总结出来的架构风格，由于 Google、Netflix、Amazon 等明星企业对号入座，微服务开始风靡整个软件业。云时代的到来又起到了推波助澜的作用，从某种意义上说未来任何云应用由于在水平扩展和弹性伸缩方面的要求会很自然地倾向于微服务架构。当然，微服务架构对整个基础设施和开发过程的要求也是比较高的，这里虽然讨论细节，但还是要提醒有志于采用微服务架构的团队在这方面更多地进行学习和思考。

从代码管理的角度来说，微服务架构带来了一个很有意思的突破：每个微服务的代码库由于受制于开发团队的人数（一般小于 10 人）都会很小。微服务的定义者 Martin Flower 和 Jame Lewis 关于微服务的代码库大小的建议是，从团队规模入手，如果团队里没有人能够掌握所有代码，则这个服务就太大了。当然具体追问这到底意味着多少行代码是毫无意义的，针对不同的个体、技术栈及业务上下文，这个答案可能会有很大的不同，但肯定不是几十万行，甚至可能不到几万行。

这个变化自然把之前讨论的是特性分支还是单一主干的问题简化了，稍加思考的团队肯定不会在这样规模的代码库上实施维护成本昂贵的分支，单一主干自然成了必然的选择。如果在代码管理层面也遵循微服务的“服务技术无关性”原则，那么每个微服务都可能存在于不同的代码管理平台上，比如一个微服务构建的平台生态圈可能既有在 SVN 上的传统基础服务，也有采用了开源框架在 GitHub 上的服务。当然，笔者目前还没有看到这样的主动行为，这种多代码管理平台的采用大多数还是由于遗留系统拆分后新产生的服务被移到了更开放的代码管理平台上，而留下的部分还在之前的代码管理平台上。

这里再提一下 Google，显然 Google 是微服务架构的实践者，而其单一代码库可能是目前公开出来的世界之最——超过 20 亿行代码，这也就意味着所有微服务的代码都是在一个代码库里的。Google 在总结这样的决策结果时指出，一体化单一代码库给他们带来的是灵活的代码结构调整和方便的依赖管理，可以很容易找到所有依赖关系。显然，从微服务的角度来说，由于服务的边界随着业务的变化而改变，代码的归属也会随之变化，Google 这样的单一库把代码的迁移工作变得非常简单。现在服务之间是通过 API 的调用来通信的，单一库也可以帮助 Google 快速明确一个 API 的全部使用情况。当然，Google 的平台全部是自研的，凭借其强大的技术能力作为支撑，一般的研发团队不论在技术实力上还是成本上都是不可能这么做的。

下面将更多地从微服务架构带来的代码管理实践变化的角度，帮助大家理解如何更好地支撑云时代的到来。

5.2　Everything as Code（一切即代码）

最近几年，业界有很多 XXX as Code 的提法，从最早的 Test as Code（测试即代码）到最近的 Infra as Code（基础设施即代码）。这种提法本身是希望开发人员意识到在现代软件开发的过程中，需要管理的不仅仅是代码，还有相关的测试用例、配置脚本、基础设施等。而如果把这些内容像代码一样管理起来，就意味着应用与代码同步的生命周期是一样的。比如，测试用例（特别是自动化的测试脚本）需要和代码一起被提交和维护，代码的

重构也涉及相关测试的重构。

在微服务架构下，这种思想变得越来越重要。为了能够快速发布服务，每个服务都需要有较为完善的自动化测试集和自动化的打包部署脚本。甚至很多团队开始实践 Pipeline as Code（流水线即代码），将整个持续交付流水线的定义脚本作为代码管理起来。结合现在的容器化技术，这实际上就重新定义了整个代码管理的目标：从过去仅仅维护代码的历史版本信息，到现在维护一整套能够完全自动化构建且可服务的综合仓库。这样的转变在 GitHub 的很多开源项目上都成了标配，至少没有自动化构建脚本的代码库不再被开发人员所接受了。

如图 5-1 所示为 Pipeline as Code 示例，由 ThoughtWorks 的开源持续交付流水线工具 GoCD 编写。该示例中的 GoCD 服务器引用了 4 个在不同 Git 仓库里的 Pipeline 定义：P1、P2、P3 和 P4。

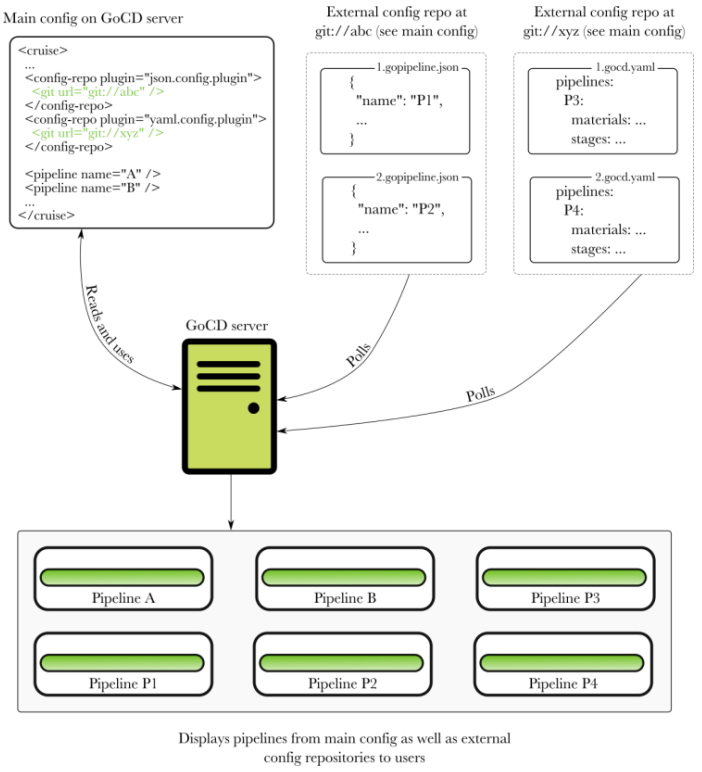

图 5-1　GoCD

在这样的要求下，我们自然倾向于采用在测试、构建、基础设施甚至文档方面都能够很好支撑“脚本化”的工具和平台。如此才能有效地利用代码管理的实践来管理这些重要的内容。比如在测试方面，QTP 这样的依靠图形界面的测试工具就很难成为合理的选择，而在文档方面常用的 Office 套件也会给我们带来很大的麻烦，自然而然地，Markdown 格式逐渐成为新的标准，以至于在 GitHub 上能够轻松构建出一套类似于编写程序用的博客系统，用 Markdown 格式本地编辑后就能够提交发布出去。

Everything as Code 应该说逐步成为了微服务架构下的标准管理方式。它的实质是把现有代码管理的成熟实践应用到软件开发的其他内容上，从而能够在真正意义上形成对可工作的软件的统一管理，为全面自动化技术的应用提供支撑。

5.3　代码管理团队自治

微服务架构的另一个隐含条件就是去中心化的管理模式，每个服务实际上都是由高度自治的团队来开发和维护的，并通过API的形式向外提供服务。除了Google采用单一库，大部分采用微服务架构的组织都采用多库的方式，每个库都对应相关的团队或者服务。按照微服务的原则，技术选型是由团队自主负责的，所以代码的管理自然也是自治的。这样会经常产生一个有趣的现象，如同Uber的架构师在 2016 年GOTO[1]技术大会上讲到的，只经过了一年半，他们就已经无法准确知道有多少个服务了，当然也没有人会关注到底有几千个服务。实际上，Uber有超过 8000 个Git库，各个团队都能够根据自己的需求建立代码库。

这样的去中心化管理模式对接口 API 的规范要求较高，每个团队都需要遵循一个组织既定的接口协议，例如 RESTful 或 RPC。只有遵循协议，才能够保证服务之间能够相互理解和调用。在这一方面很多团队都开始采用 Swagger 这样的 API 定义工具框架来绑定前期的 API 定义和后期的代码实现。Swagger 也提供了从 API 定义自动生成代码框架的功能。当然这是否是微服务接口设计的最佳实践，目前还没有定论，很多资深程序员始终对自动生成代码持保留意见。但无论如何，遵循规范的 API 定义应该像代码一样被管理起来，这

1　https://blog.gotocon.com

样 API 的使用者便能够通过代码管理平台找到相关的定义及使用案例。

如图 5-2 所示为采用 Swagger 进行的 API 定义，左边窗口是采用 Yaml 格式的 API 定义代码，右边是对应的 Web 展示界面。

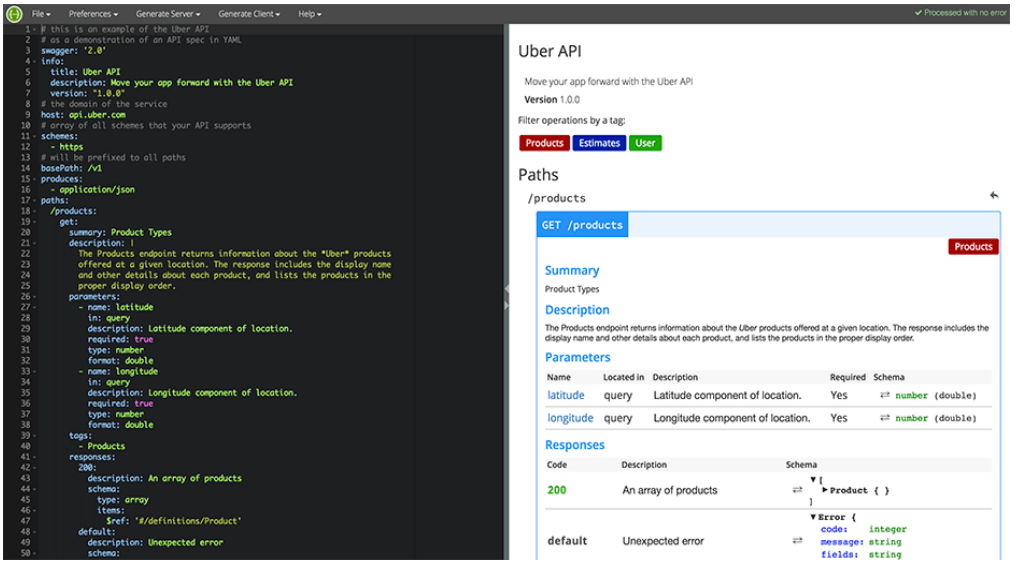

图 5-2　Swagger 中的 API 定义

另一个自治的结果是不同的团队采用了不同的技术栈，比如有的团队利用了成熟的 PHP 框架，而有的团队希望追求性能而采用 Golang。理论上而言，微服务架构组织里经常会有采用技术栈的规则，这样可能降低了技术维护的成本，也能够采用统一的代码管理平台。笔者也经历过一些特殊情况，比如历史上用 COBOL 实现的核心业务系统，其相应的代码管理平台是内嵌的。笔者曾经尝试着辅导一个团队把一个 COBOL 代码库迁移到 SVN 上，虽然迁移是成功的，也在后续的开发过程中给开发人员带来了一定的灵活性和便捷性，但最后从 SVN 合并到主代码库的过程是痛苦的，代码提交的宝贵历史信息也都丢失了。

在团队自治管理的大原则下，微服务架构的代码管理仍然可能有下面两个维度：围绕服务和围绕团队。随着整个应用系统的演进，我们不可能保持每个团队一个微服务的“完美”匹配。类似经典组织里的矩阵管理模式，团队和服务这两个维度也是在管理过程中动态平衡的。我们希望通过下面两个小节来谈谈对这两个维度的管理。

5.3.1　围绕团队的代码库管理

在敏捷开发下提倡小团队的相对稳定，有利于磨合出有一致愿景和凝聚力的队伍。在微服务架构下必然形成这样的小团队集群组织，而在初期保证每个团队的稳定也是必须做到的。这时很显然我们的代码库管理更多地围绕着团队而来，即每个团队自己有相对独立的代码管理仓库。

类似前面 Uber 的案例，在强团队管理的模式下每个团队自然会派生出多个服务，这样的进化过程更像一个大集市，每个团队都在集市上陈列自己的各种服务，团队外部的“客户”更关注的是自己需要的服务，并不很关心自己需求之外的服务，当然也可能有市场管理员（系统架构师）关注整个集市的规划。然而市场管理员也没有办法准确说出自己的大集市里到底卖了多少种商品或服务。

这样的治理模式下的代码管理相对简单，不管有几个服务，都由团队自己管理，理论上团队自己承担了整个管理生命周期。当然，我们认为组织还是有必要统一代码管理平台的，这样有利于降低管理成本和推行相关的组织代码规范。特别是在管理对外接口部分（如 API）时，必须有统一的代码管理平台。在最近帮助一个大型组织打造服务化 API 生态圈时，我们就在统一的代码平台上针对 API 规范打造了相关的元数据管理，从而能够保证提交的 API 在规格和安全等方面能够达到组织的基本标准。

5.3.2　围绕服务的代码库管理

当我们的微服务生态圈演进到一定阶段时，不可避免地会有不同的团队希望修改同一个服务。强制约束服务和团队的一对一关系可能会增加整个生态圈的不必要的复杂度，正如康威定律指出的，如果拿掉了团队组织结构上的灵活性，则显然带来的就是产品的僵化。

如图 5-3 所示，人为强制要求服务和团队的强对应关系，必然会造成重复服务 A 的出现，进而增加整个系统的复杂度。

这时我们会遇到很多经典的代码管理问题：跨团队如何进行有效的代码层面协作，如何维护代码分支，以及如何有效管理代码权限。庆幸的是这些问题其实在开源社区早已经

逐步演化出很多相关方法和实践。大型开源软件本身的开发就是分布式的，甚至很多开发人员在整个开发过程中都没有机会面对面。在这样的场景下，一些受到推荐的技术实践如持续集成、代码走查就成为了必需的工程纪律。

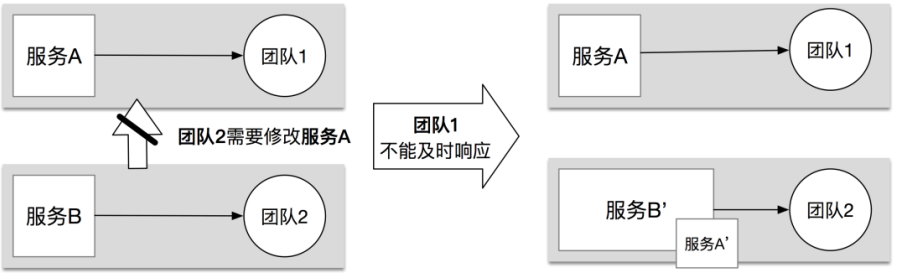

图 5-3　微服务中的团队与代码库管理

在代码管理层面，分布式代码管理（如 Git）成了必要条件，中央管理的方式不但成本高而且不再是可持续的方式。想象一下，如果每个团队都需要修改其他任何一个不归属于自己的服务，则都必须要向某个中央部门申请，其结果一定是团队选择自己重新实现一个类似的服务。这显然不是一个组织希望看到的情况，所以分布式的管理机制是必需的。这时带来的一个挑战就是如何控制代码的质量，以及如何保证其他团队提交的代码不破坏服务负责团队的架构设计。和大型开源软件的做法一样，在线的代码走查、合并也成为必要的控制过程。每次关于一个服务的代码提交都应该由负责此服务的团队走查后再合并到主干分支上。前面讲到的 GitLab 和 Gerrit 很早就已经支持这种在线的代码走查了。

Git 这样的分布式代码管理平台因缺乏必要的权限管理被很多企业诟病，所以我们在 Git 的基础上增加了权限模型。这显然违背了 Git 的提出者 Linux 之父 Linus Torvalds 的开源思想。笔者经历过多次这样的尝试，很多时候都事与愿违，不但没有得到开发人员的认可，反而增加了代码管理平台的运营成本。现代 IT 系统大部分是适配特定业务需求的，拿到代码不等于理解了核心业务。这一点已经逐步在行业内达成了共识，比如英国政府的数字化部门 UK Digital Service 不但内部创造了互联网式的开放文化，它的代码也被逐步迁移到了 GitHub 上为大众使用。当然，一些需要在代码级别（如核心算法）进行保密的公司大可以采用加密技术对代码进行混淆。在微服务时代，我们认为再去构建类似中央控制时代精确到代码文件级别的权限控制意义不大。

5.4　微服务架构下的代码管理挑战

我们对微服务架构的认知还需要通过对更多的组织在未来的实践来观察。正如前面讲到的，在 Everything as Code 的趋势下，我们对代码的定义正在发生演进。我们观察到有些团队甚至将容器的 Docker 镜像也作为“代码”一起进行了提交和管理。在一些大型产品团队中，更有团队在尝试将文档和产品的使用手册作为“代码”管理起来，这样便能够和整个产品的开发生命周期保持同步，进而解决产品在最后发布时文档的不同步问题。

我们认为在微服务架构继续发酵的时代，在代码管理实践方面需要考虑以下三方面的挑战。

◎　跨团队的代码协作和共享。如何建立有效的分布式代码管理过程从而保证多个团队在代码修改层面的协作和共识？大型开源软件的做法可以作为很好的参考。但一个企业一般会有更多的流程规范（比如在质量和安全方面的），需要考虑如何适配这些流程和规范。

◎　跨服务的代码重构及维护。随着各个微服务的演进，跨服务的代码改变很可能无法避免，比如 Google 对全部 C++代码基于 C11 标准的升级。大部分组织在实施微服务架构时都不会采用 Google 这样的单一库，那么跨服务的变更就要求在多个代码库中同时进行操作。如果没有建立相关的自动化机制，那么这样的跨服务（代码库）的改变是不可能完成的。

◎　多代码库的版本策略。微服务架构带来的一个显著优势就是每个服务可以持续快速地进行更新和发布，而服务之间不可避免地存在集成调用的关系，理论上每个服务升级都不应该破坏之前已经存在的调用。但每个服务都有可能因为自身的升级出现不可预期的问题，这时需要有快速的自动化回滚能力，回滚的版本需要根据既定的代码版本策略来决定。同样，这个版本策略会对应到之后的代码问题分析及修复过程中。

5.5 微服务代码管理实例

该企业是一个大型在线广告业务平台，在 2013 年左右在房地产领域成为在线行业的领头羊，希望能够构建更能支撑其业务发展的信息平台。这时微服务架构进入了该企业架构师们的视线，出于业务多元化发展的预期，该企业最后选择了这个架构方向。

经过了半年的架构解耦工作，其主力开发团队把包括财务和资源在内的系统都进行了领域分解，形成了服务化平台的雏形。整体的服务架构如图 5-4 所示。

在如图 5-4 所示的企业服务架构中，Domain 为领域模型层，包含核心的业务对象抽象；Application 为业务逻辑及规则的封装，以服务（service）的形式对外暴露；Interface 为各个数字化渠道提供交互。

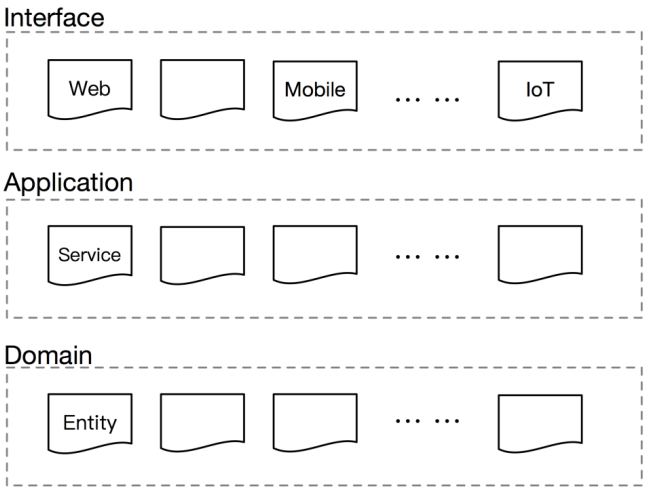

图 5-4　微服务代码管理实例一

每个服务模块都被迁移到了 GitHub 之上，每个业务单元对应了一个代码库，每个业务单元里的服务对应了代码库下的一个目录。基本结构如图 5-5 所示。

在图 5-5 中，resi/agent、developer/investor 和 media/analyst 等都对应到了相应的业务单元，这里的 agent 针对的是中介机构，investor 针对的是建房投资者，都是和业务单元直

接对应的。每个业务单元对应一个 repo，在 GitHub 上以组织（organization）的形式存在。

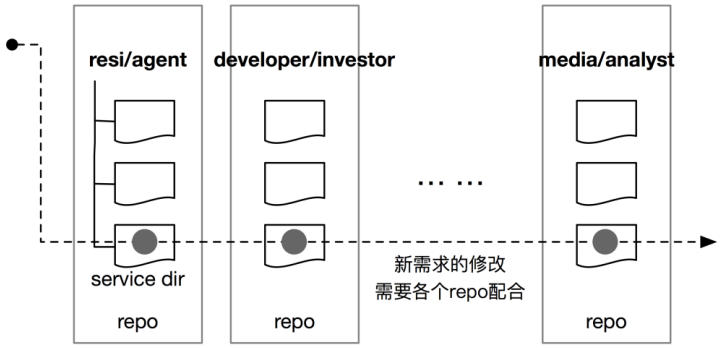

图 5-5　微服务代码管理实例二

每个业务单元都有多个小开发团队，每个服务都会有一个比较固定的小团队来负责开发。由于转到 GitHub 带来了开放性，这自然受到了团队开发人员的拥护，在一段时间里应该说整个组织的氛围是积极的。

值得一提的是，每个服务都提供了相对标准的自动化部署脚本及自动化流水线脚本，遵循了前面提到的 Pipeline as Code 的实践。每个库都会下面的两个 Yaml 脚本。

◎ repo/.buildkite/pipeline.yml：遵循了 Pipeline as Code 的实践，定义了这个库中服务的构建流水线。

◎ repo/.config/deployer/deployer-production.yml：遵循了 Infra as Code 的实践，定义了容器及其他运算资源的要求。

当然，一个经典的问题很快出现了，业务变化如期而至，为了满足业务的需求，需要有多个服务来配合修改。这是一个敏捷文化已经相当深入的组织，大多数时候能够靠团队间的协作来完成同步的实现。在一段时间后团队的管理者感觉到了效率上存在的问题，这时开始思考如何在这样的微服务化架构下提升开发的效率。

由于了解敏捷开发方法，负责人首先想到的是 Kanban 方法，那么是否可以通过建立端到端的价值流意识来提升业务需求的交付效率呢？这本质上是让各个小团队能够从业务需求的角度思考流动效率，而不仅仅是自身的服务为了实现这个业务需求而进行局部变更。从

某种意义上讲，Kanban 及其相关度量方法的应用帮助团队意识到了协作过程中的成本。

该团队开始意识到真正的问题并非各个小团队不关注一个业务需求的端到端实现，而是为什么有那么多的业务需求在实现的过程中存在对各个服务的严重依赖。

当然，经过微服务这几年的实践，我们知道这样的问题出现在服务的划分之上，需要按照业务的上下文去划分服务的边界，这样才能够实现服务之间的低耦合，避免前面提到的微服务架构实施后服务之间错综复杂的集成关系。值得注意的是，这样的服务边界划分的调整也对应着代码结构的调整，新的代码结构按照这种划分形成了如图 5-6 所示的组织结构。

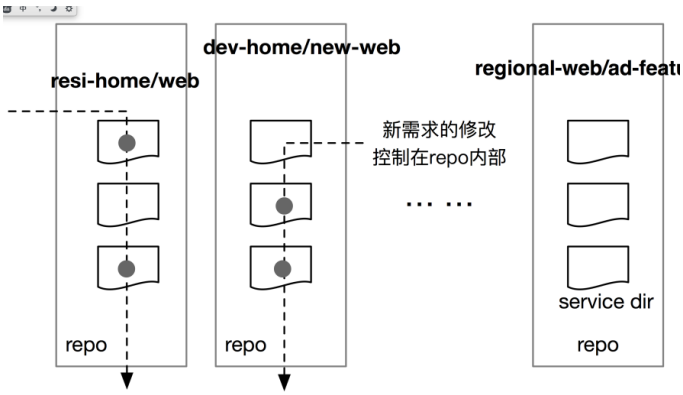

图 5-6　微服务代码管理实例三

在图 5-6 中，resi-home/web 是针对寻找公寓用户的 Web 应用，而 dev-home/new-web 是针对准备自建房用户的新 Web 应用。显然，当这两个领域能够在业务上独立存在时，对新需求的修改基本上可以控制在对应的代码库内部。

从代码管理的实践角度出发，大家可能会觉得这样的调整并没有改变多少，毕竟我们仍然使用的是 GitHub，一个服务仍然主要由一个团队负责。但从代码管理实践的角度，我们需要理解的是：只有良好的微服务架构设计、低耦合的服务划分才能最终简化微服务架构下的代码管理，让我们可以把更多的精力应用到对业务需求的响应及系统架构的演进上。

云和微服务时代也让开源这个去中心化的模式得到了进一步的推广，很多传统企业也开始思考如何将开源思想应用到自己的 IT 系统建设中。这些年出现的“内源”（即内部开

源，对企业的所有员工逐步开放系统源代码）已经超越了简单的对开源软件包的拿来主义，而是开始学习开源的开发模式甚至思想文化。我们在第 6 章里将简要回顾开源的前世今生。

第 6 章

开源项目与开源社区

6.1 开源软件

开源软件从 1969 年的 UNIX 操作系统在 AT&T 贝尔实验室诞生并开源后，到现在已经存在近半个世纪。在这近半个世纪中，开源软件在全世界有志之士的努力下得到飞速发展。1983 年 Richard Stallman 创立了 GNU 项目（包括 GCC、Emacs 等伟大的开源软件），引领了开源软件的潮流。1985 年 Richard Stallman 成立了 Free Software Foundation。1989 年第 1 个 GNU 通用公共授权条款 GPL（GNU General Public License）正式发布。1991 年 Linus Torvalds 发布了举世瞩目的 Linux 第 1 版内核，引领了开源操作系统的热潮。1995 年 Apache HTTP Server 和 1998 年 Netscape 的开源开启了第 1 次互联网时代的到来。1999 年 Apache 软件基金成立，从而开启了大规模支持开源软件商业的新时代。2005 年 Linus Torvalds 发布了分布式开源代码管理工具 Git，造就了当前最大的开源社区 GitHub。2015 年微软宣布拥抱 Linux，使世界上用户量最大的两个操作系统（一个开源的 Linux 和一个收费的 Windows）开始融合。整个软件行业在近半个世纪的高速发展，离不开开源软件和其开发者的巨大贡献。许多人因为开源软件实现了梦想，许多公司也因为开源软件获得了

巨大的财富，现在许多新的 IT 前沿黑科技也是基于开源软件建立，并在开源社区的帮助下蓬勃发展的。

6.1.1　开源软件的特点

开源软件一般使用去中心化的公开协作的模式来开发软件，其中去中心化的公开协作是指开发工作并不是由一个独立的中心团队进行的，而是由许多分布在世界各地的开发者通过社区等其他公开方式协作开发软件的（一些小型开源软件可以由一个人开发完成，但是绝大多数开源软件都需要由多人甚至大量的人员协作才能开发完成）。

任何人都可以免费且自由地学习、运行、复制、修改，甚至重新免费分发开源软件。某些开源协议允许进行收费的重新分发，常用的开源协议[1]也有许多个。随着 21 世纪互联网的飞速发展，开源软件以惊人的速度发展。现在几乎每个人的计算机上都运行着某些开源软件，而大部分软件产品也基于大量的开源软件进行开发、测试和运维等。所以，现代的软件开发已经离不开开源软件了，而且开发模式也逐步向开源软件学习，使用去中心化的公开协作模式来开发。而这种开源软件的开发模式的核心就是代码管理的去中心化的公开协作管理模式。

6.1.2　开源软件和社区

中大型的开源软件一般是由开源社区进行开发和维护的，而开源社区的核心是代码管理系统，并辅助问题跟踪、文档管理、信息交流等模块的复杂在线系统。现在世界上最活跃、最大的开源社区是基于 Git 代码管理工具的 GitHub。虽然开源社区的先驱，即最大的 Subversion 开源社区 SourceForge，已经无法企及 GitHub 了，但是这个社区仍然维护着一些老的重量级的开源软件，仍然不容忽视。

由于Git天生就是去中心化的分布式代码管理工具，所以基于Git的GitHub受到了全世

1　https://opensource.org/licenses

界开源开发者的拥护。现在绝大部分新的开源软件都是使用GitHub进行代码管理的，它们基于GitHub进行开发和维护，很多老的开源项目也迁移到了GitHub上（截至 2017 年 5 月，通过API确认，GitHub上已经有超过 9 千万个代码库 [1]）。

GitHub 获得了大量开源开发者的拥护，一些大型商业软件公司（比如微软、IBM、Oracle等）和一些大型互联网公司（比如 Google、Facebook、阿里巴巴和腾讯等）也将其开源项目放到了 GitHub 上，由此可见开源社区及其精神已经深深浸入到商业软件公司中，并且已经慢慢地改变整个软件行业的开发方式、管理方式甚至未来。

6.1.3 开源软件和商业

对于很多商业企业来讲，开源软件已经成为其商业软件中不可或缺的组成部分，无论是其代码开发、管理代码还是运维阶段，都使用了许多开源软件，比如各种开源框架如Spring、ROR 等，开源代码管理工具如 Subversion、Git 等，开源部署工具如 Puppet、Ansible等。这些开源软件中有很多在背后都有商业企业的扶持，比如出资聘请社区里面的人员，或者在公司内部成立专门的团队来开发开源软件，或者将其进行企业级功能扩展，变成商业软件，也可以基于这些开源软件开发商业软件以获得巨额利润。当某款开源软件不再满足某个企业的需求时，企业就需要对这款开源软件进行修改和定制，比如阿里巴巴定制MySQL，以及小米和华为定制 Android 等。但是当官方版本的开源软件升级时，企业的定制版一般也需要升级，所以需要使用和官方版本一样的代码管理方式，包括管理分支等，因此在未来，开源软件和商业软件将密不可分。

6.1.4 开源软件的代码管理

开源软件的主要代码管理模式就是去中心化地公开协作及管理代码。首先，开发人员能在任何地方及地点开发代码并提交代码，甚至能在网络不稳定或者无网络的情况下查看日志并进行代码比较。其次，由于开源社区中开发人员的能力参差不齐，所以需要功能强

1 https://api.github.com/repositories?since=90000000

大的分支系统、强大的权限管理及在线代码评审系统。另外，由于开源社区的人员可能分布在全世界的任何地方，所以高效的协作也需要一个功能强大的协作系统，比如问题追踪、在线文档、在线构建、在线部署和在线配置等协作系统。

6.2　开源社区中的开源项目

6.2.1　简介

开源界中大部分开源软件都是由开源社区开发和维护的，这些开源软件一开始是由某个人或者某个小型组织发起并开发的，然后越来越多的爱好者和使用者逐步加入到开发队伍中参与开发或者修复问题，甚至有商业公司出资，让社区里面的贡献者可以获得报酬从而可以投入更多的时间甚至全职投入到开源软件的开发和维护中。所以一般情况下，一个开源项目最初的代码管理相对简单，可能不需要复杂的分支管理，也没有复杂的权限和代码评审机制，只需要依照小型团队的代码管理模式就可以了。但是由于需要和其他贡献者一起协作开发和维护，所以一般会选择一个大型的、功能强大的基于互联网的云代码管理系统，并使用其强大的协作功能来开发开源软件，比如 GitHub、码云或者 SourceForge 等。有些开源软件随着使用者越来越多，影响越来越大，贡献者也越来越多，代码的规模也越来越大，导致代码管理越来越复杂，比如贡献者太多，每天的代码提交量巨大，导致代码评审、合并难以有效完成。所以开源社区中的大型开源项目也有各种困难需要解决，比如 Git 就是 Linus 为了解决 Linux 内核源代码开发过程中的代码管理问题而专门设计的。

6.2.2　代码管理模型

由于开源社区的特殊性，权限管理是最为重要的，比如在严格的权限控制下还要允许任何人都有机会向项目贡献代码等；其次项目协作也非常重要，比如可以在线跟踪问题、在线审查代码，以及在线对代码进行持续集成等。

开源社区中开源软件的代码管理模型一般分为以下两种。

1. 一个开源代码库，多个开发分支

此模型是开源社区以 Subversion 为主的第二代代码管理工具时代的主要代码管理模型，它主要使用标准的代码库管理方式，贡献者首先需要申请获得代码库的特定分支的权限，然后在特定的代码分支上进行开发，最后由特定的人员在审核代码后将代码合并到主干分支上。如图 6-1 所示。

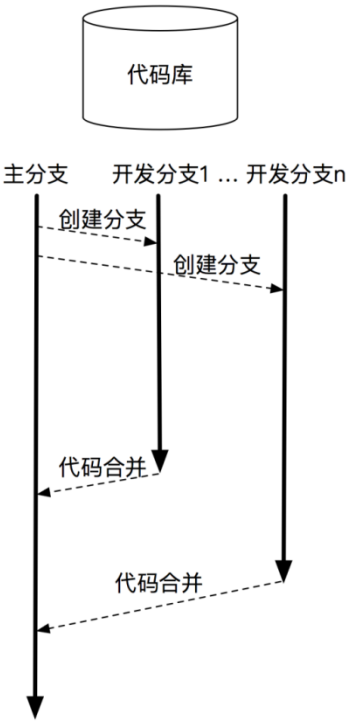

图 6-1　开源代码管理模型一

2. 一个主开源代码库，多个开发分代码库

此模型是当前开源社区以 Git 为主的第三代代码管理工具时代的主要管理模型（见图 6-2），它主要使用代码库复制并自动同步，比如 GitHub 的代码库复刻，每个贡献者都可以在没有经过授权的情况下使用代码库复刻对主代码库进行复制及同步，然后通过在线

代码审查和合并的方式将代码提交到主代码库，比如 GitHub 的 Pull Request。

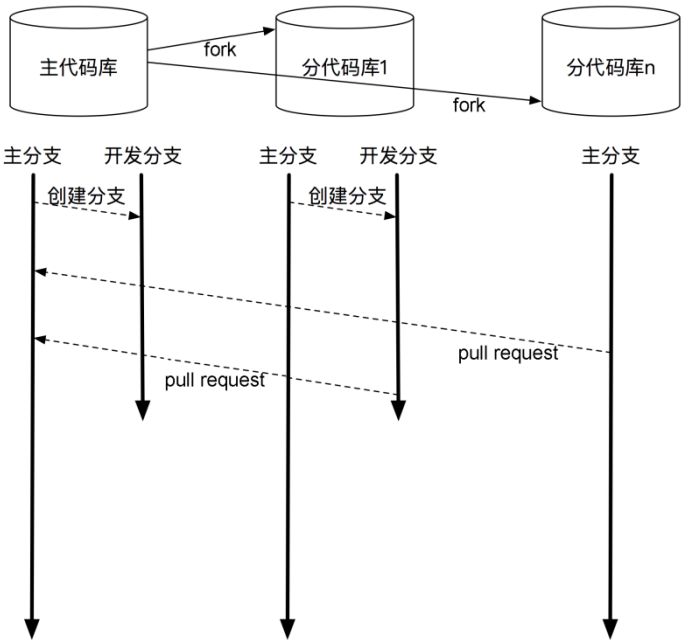

图 6-2　开源代码管理模型二

6.2.3　典型的大型分布式开源项目

OpenStack 是一款典型的大型分布式的开源项目，其规模很大，贡献者很多，在开源社区中规模很大。截至 2017 年 6 月，OpenStack 一共拥有超过 1700 个独立代码库，超过 300 万行代码，有超过 6000 名开发者贡献过超过 30 多万次代码提交。他们分布在世界各地，并且从 2014 年以来每月的代码提交量基本都在 5000 次以上，在不少高峰月还超过了 10000 次。因为使用了和 Android 一样的 Gerrit 来管理项目代码，所以也用了和 Android 类似的分支策略和工作流程，如图 6-3 所示。

OpenStack 项目遇到了一个棘手的问题：每天的代码审核提交量巨大，很难在有限的时间内及时对所有提交审核的代码进行构建和测试，并在测试通过后合并到主干。特别是有多次代码提交同时发生，并且其中某次提交会编译失败或者测试失败，在这种情况下开发

者很难定位编译或者测试失败是由哪次提交导致的，所以他们需要耗费大量的时间进行定位等，导致很多提交和合并的时间间隔太长，所以 OpenStack 社区开发了 Zuul 来解决这个问题。Zuul 的工作原理如下。

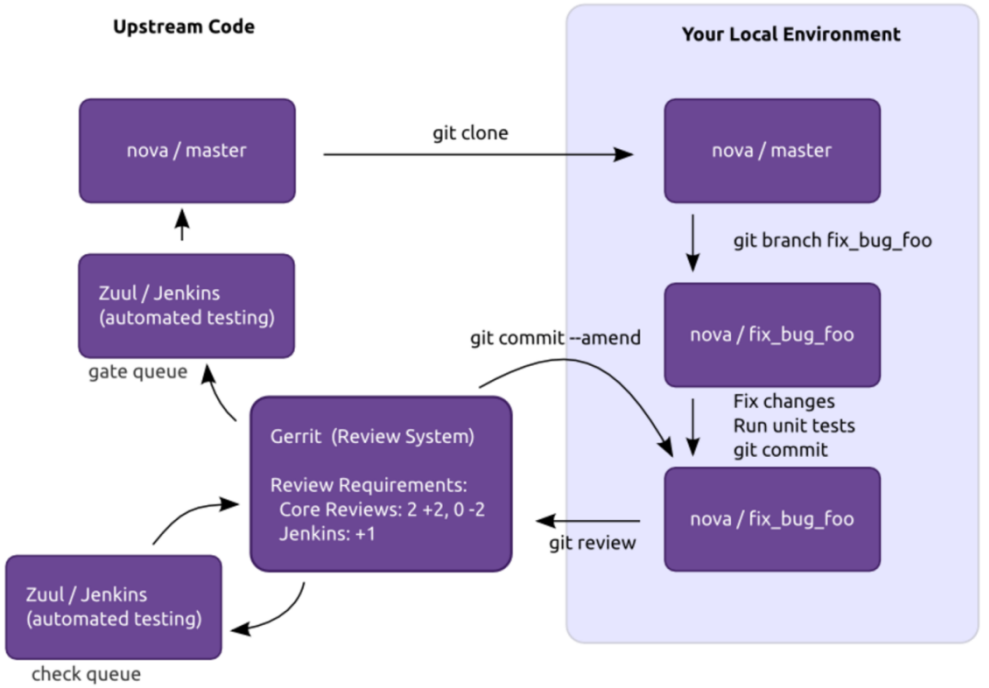

图 6-3　OpenStack的代码管理流程 [1]

假如核心开发者一次性推送了 5 次代码提交（分别表示成 A、B、C、D、E）并产生了一个合并队列，如图 6-4 所示。

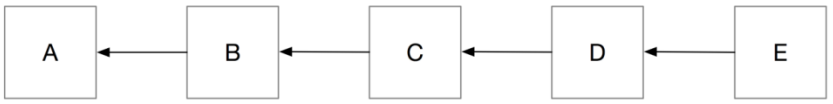

图 6-4　一次性推送的 5 次提交

Zuul 将会为这 5 次代码提交生成 5 个不同的代码提交组合，并为每个提交组合各生成

1　https://docs.openstack.org/infra/manual/developers.html

一个 CI 测试任务（一共 5 个），它们分别是：

（1）A

（2）A + B

（3）A + B + C

（4）A + B + C + D

（5）A + B + C + D + E

首先，由于它们是并行执行的，所以可以节约大量的时间。其次，由于这样的组合包含了不同的代码提交，所以在某个 CI 任务失败后，结合其他 CI 任务的构建结果就能马上知道是哪次代码提交造成的失败，而哪些代码提交是成功的，Zuul 会自动合并成功的代码提交，并忽略失败的及后续的代码提交。

如果第 3 个组合 CI 任务失败，而前两个组合 CI 任务成功，则它能自动将 A、B 合并到主代码仓库中，并忽略 C、D、E 的代码提交。如图 6-5 所示。

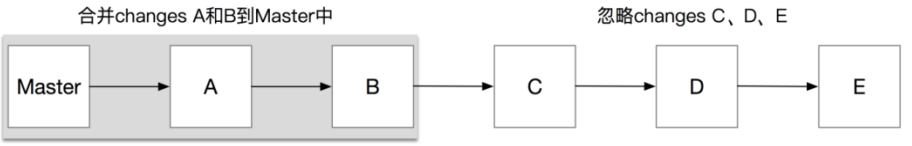

图 6-5　合并验证成功的提交

而如果这 5 个组合的 CI 任务全部成功，则它会一次性将所有代码提交（A、B、C、D、E）都合并到主代码仓库中。如图 6-6 所示。

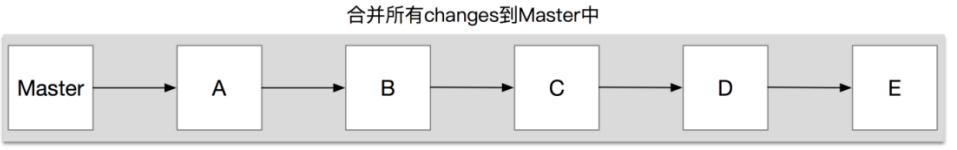

图 6-6　如果全部组合验证成功，就合并所有提交

对于 CI 任务失败的情况，Zuul 会忽略错误的代码提交，重新生成 CI 测试任务。比如

上面那个例子，若第 3 个组合测试失败，则说明代码提交 C 肯定有问题，但是代码提交 D 和 E 有可能是好的。如图 6-7 所示。

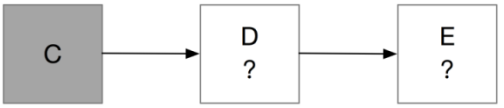

图 6-7　C 之后的两次提交状态是未知的

所以 Zuul 首先将 A 和 B 合并到主干代码中并忽略 C、D、E，其次重新对 D 和 E 组合后进行测试，组合如下：

（1）D

（2）D + E

如果第 1 个组合（D）的 CI 任务成功，而第 2 个组合（D 和 E）任务失败，那么 Zuul 会将代码 D 合并到主干代码中并忽略 E。如图 6-8 所示。

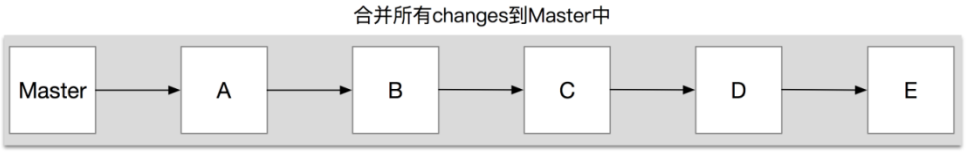

图 6-8　只有 D 被合并

如果第 2 个组合的 CI 任务也成功了，那么 Zuul 会将 D 和 E 一起合并到主干代码中。如图 6-9 所示。

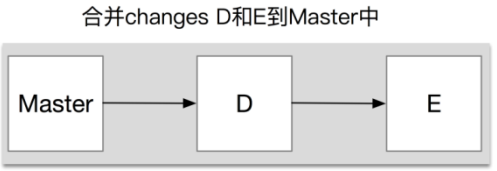

图 6-9　D 和 E 被合并

由于 Zuul 的这个特性，在 OpenStack 上有大量的代码被提交，同时推送到 Gerrit 进行 review 后，代码测试和合并的时间大大减少，从而提升了 OpenStack 管理代码的效率。

6.3　企业中的开源项目

6.3.1　简介

企业在一般情况下都是通过开发商用软件并销售获利的，但有时因为一些原因将部分商业软件进行部分开源或者全部开源，其原因包括提高公司或者产品的知名度、获得免费的社区资源、支持开源、回馈社区等。一般企业选择的开源软件都是由商业公司通过大量的人力和物力开发出来的，代码规模非常大，所以当这些软件被开源并开放给开源社区之后，一开始并不会有很多来自社区的贡献者，其开发主要由商业公司的员工来完成。但是为了让其他贡献者顺利加入，代码的各种管理都必须简单、易用且功能强大，比如分支管理及权限管理等。还有一些企业在公司内部成立了专门的开源社区或者开源部门来开发内部的开源软件，在达到一定的成熟度后马上开放给开源社区，并借助开源社区的力量来帮助其开源软件成长。

6.3.2　代码管理模型

代码管理模型一般分为如下两种。

（1）一个主开源代码库、多个开发分支或者多个开发分代码库

这种模型是本书 6.2.2 节中提到的两种开源模型的合集，此处不再赘述。这个模型虽然可以帮助商业公司节约代码的管理成本，但如果公司还想保留一些私有代码，那么这些代码就存在泄漏的风险。因为在这种模型下面需要将整个代码库放到开源社区里面，就算进行良好的权限管理，也难免不会泄露。所以这个模型只适合完全开源的企业的内部项目。

（2）一个开源代码库、一个商业代码库

这个模型适合大部分需要保留商业产品的企业，对内有一个私有代码库，对外有一个开源代码库。首先，需要在对内的私有库中创建一个或多个特定的开源分支，每隔一段时

间就需要将这些私有库中的代码分支同步到开源库中，或者把开源库中的更新同步到私有库的开源分支中，再筛选有用的代码提交、合并到私有库的私有分支中，具体的同步时机由每个项目根据自己的实际情况指定：比如可以是在修复了一些重要的 Bug 后，可以是在完成了一个特定的功能后；也可以是在一个大的版本升级后，等等。由于同步合并时可能出现冲突，而且大量的合并本身就是一种高成本和高风险的活动，所以这个模型的成本和风险都不小，需要由特定的人员来维护这两个代码库，包括分支、同步策略等都需要根据项目的情况自己定制，一般可以分为三种策略：从开源到商用、从商用到开源、商用和开源并行。

6.3.2.1 从开源到商用

Google 开源了 Android 手机操作系统，使得很多手机厂商或者软件厂商能够对其进行定制化，然后变成私有的代码库，比如定制化操作系统驱动，或者修改平台层用于对外提供特定的额外接口，并用到自己的产品中。公司 A 正在定制一款智能电视，其中电视使用的是 MIPS 芯片，而其操作系统使用的是 Android，而且为了配合运营商的点播系统，还要加入定制化的点播系统协议。所以公司 A 需要在公司内部创建一个私有代码库进行定制开发，并在 Android 官方发布重大更新或者需要某些特定修复时从 Android 官方开源库中将其同步到私有库中的特定合并分支，然后进行烦琐的合并工作，并在合并完成且通过测试后再合并入自己的主开发分支中。

6.3.2.2 从商用到开源

商业公司为了各自的目的将自己的商业软件开源，比如有些企业是为了获得更好的知名度和更多的用户等而开源的，例如Symbian一开始是一个标准的商用操作系统，但是随着Nokia的没落，Symbian也逐步开源了，尽管现在Symbian的开源已经名不符实了。在这种策略下一般只把商业软件的特定版本定时同步到开源代码库中，而不需要从开源代码库中同步代码到私有代码库，或者直接使用开源代码库进行开发，比如英国政府官网 [1]。这个策略的最大好处就是简单且易于管理，但缺点是需要完全暴露源代码，并承担风险。

1 https://github.com/alphagov

6.3.3.3　商用和开源并行

这种策略是当前最为常见的商业软件开源策略，比如 MySQL 等。它的最大特点是企业保留了一些特定的功能或者配置，只将核心功能或者通用功能开源，开源社区的开发者可以通过类似 pull request 的方法提交代码到开源库里，然后项目的所有者会将开源库中有效的提交合并回私有代码分支或者代码库，并同时将私有代码库中的一些更新同步到开源代码库中。这个策略的最大好处是可以充分调动开源社区的积极性，让感兴趣的人可以贡献代码，但是缺点也很明显，代码评审和同步的工作量巨大，并且管理较为复杂。但是一个大型项目为了获取社区的支持及良好的知名度，付出一些代价也是值得的。

6.4　GitHub 中的开源项目实践

无论是开源社区中的开源项目还是商业公司中的开源项目，GitHub 都是一个比较好的选择。首先，GitHub 同时支持公开代码库和私有代码库；其次，由于在全世界范围内拥有强大的社区影响力，GitHub 上面的开源项目可以获得开源社区的更多关注，甚至带来可以利用的社区资源。所以在需要开发一个开源项目或者开源一个商业项目时，GitHub 是首选，可以很好地实现开源软件开发中的各种代码管理模型。

6.4.1　分支管理

一个开源代码库且多个开发分支模型中最为重要的就是分支管理。GitHub 虽然没有提供和 Gerrit 一样强大的分支管理功能，却提供了一个关键但是简单的分支保护功能，即要求每次代码提交都必须通过审查 pull request 进行合并，而不像 Gerrit 和 GitLab 一样对每个贡献者进行权限控制，使对于分支的权限管理变得简单。由于 GitHub 主要针对公开的开源软件代码库，所以这种简单的分支权限管理非常适用。而对于 GitHub 上的私有代码库，一般也不是一个公司的核心代码库，所以一般也不需要十分严格的权限管理，所以这样的分支权限管理也基本适用。

要设置分支权限保护，需要先进入代码库的设置界面，并在“Protected branches”下选择需要保护的分支（默认是 master），如图 6-10 所示。

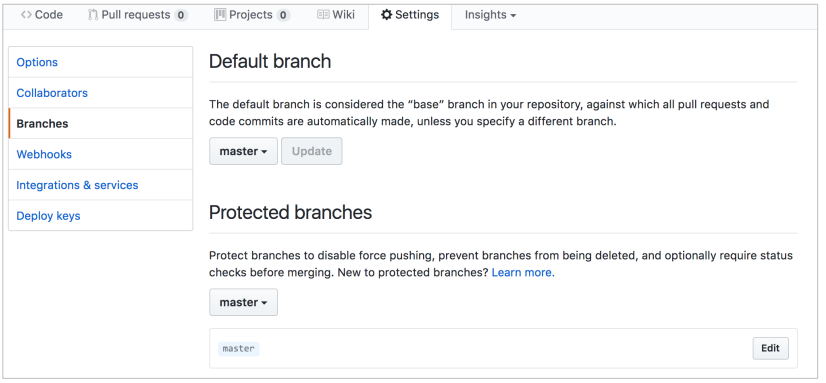

图 6-10　选择 GitHub 保护分支

在选择了分支后就会进入分支保护配置界面，在一般情况下默认配置（包括禁止强制推送此分支、必须通过审查才能合并、合并之前需要检查特定的状态等）就够用了，单击“Save changes”就完成了分支保护配置了，如图 6-11 所示。

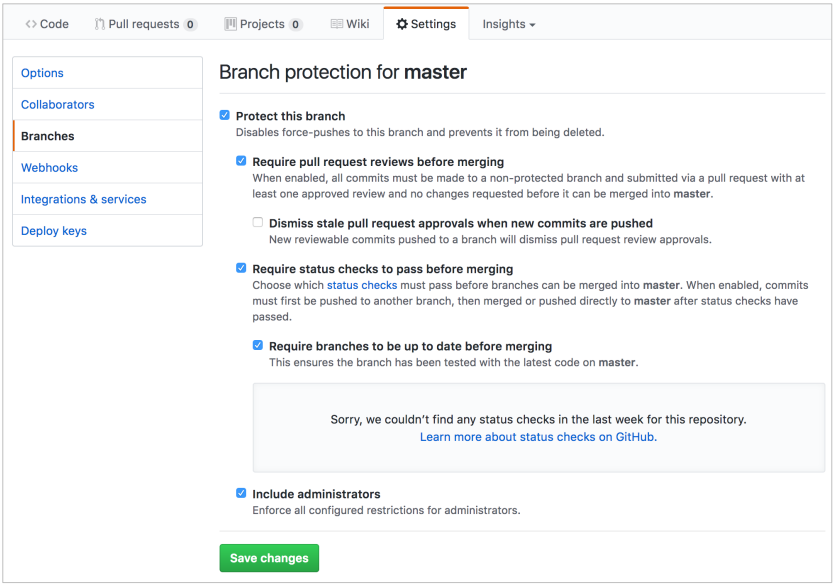

图 6-11　保护 GitHub 分支配置

6.4.2　分库管理

一个主开源代码库且多个开发分代码库是当前大型开源项目使用的主要模型，而这个模型的重点就是创建和管理多个开发分代码库。

在GitHub上创建一个开发分代码库的方法就是使用代码库复刻复制主开源代码库，只要在对应主开源代码库的GitHub界面里单击Fork[1]就可以完成对这个代码库的复刻，如图 6-12 所示。

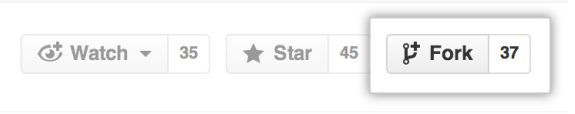

图 6-12　GitHub Fork图 [2]

在长时间使用后可能出现复刻的库和被复刻的库之间存在代码差异，这时需要同步两个库之间的差异。GitHub 记录了复刻的库和被复刻的库之间的关系，使得复刻的库可以同步被复刻的库的代码更新，有两种方法可以完成这个更新。

方法一：使用命令行

（1）首先通过 git clone 命令将代码库下载到本地：

```
$ git clone https://github.com/YOUR_USERNAME/YOUR_REPOSITORY.git
```

注意：YOUR_USERNAME 是代码库的拥有者的名字，YOUR_REPOSITORY.git 是代码库在 GitHub 上的名字。

（2）查看代码库的 remote 配置，并设置上游的原始代码库（被复刻的代码库）与本地代码库的关联。

首先使用 git remote -v 命令查看当前的本地代码库和远端代码库的关联关系：

1　https://help.github.com/articles/fork-a-repo/

2　https://guides.github.com/activities/forking/

```
$ git remote -v
origin  https://github.com/YOUR_USERNAME/YOUR_REPOSITORY.git (fetch)
origin  https://github.com/YOUR_USERNAME/YOUR_REPOSITORY.git (push)
```

在初始情况下，本地代码库只有上面两个关联配置。为了能和上游原始代码库（被复刻的代码库）进行同步，还需要设置这个上游原始代码库的管理关系，使用以下命令实现：

```
$ git remote add upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git
```

注意：ORIGINAL_OWNER 是原始代码库的拥有者的名字，ORIGINAL_REPOSITORY 是该代码库在 GitHub 上的名字

运行这个命令后再通过 git remote -v 就可以查看到新的关联关系了：

```
$ git remote -v
origin  https://github.com/YOUR_USERNAME/YOUR_REPOSITORY.git (fetch)
origin  https://github.com/YOUR_USERNAME/YOUR_REPOSITORY.git (push)
upstream        https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (fetch)
upstream  https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (push)
```

（3）获取远端上游原始代码库的内容。

使用 git fetch upstream 命令可以从远端上游原始代码库获取代码、代码分支及代码提交等，它们会被保存在本地的 upstream/master 分支中。

（4）合并远端上游原始代码库的更新。

首先切换到本地的开发分支上，在这个例子里 master 是开发分支：

```
$ git checkout master
```

把 upstream/master 分支合并到本地的 master 分支上，本地的 master 分支便与远端上游仓库保持同步了，并且没有丢失本地的修改。

```
$ git merge upstream/master
```

如果这时 merge 有冲突，则需要手动解决冲突。

提示：同步后的代码仅仅保存在本地仓库中，记得推送到 GitHub。

方法二：使用 GitHub 的 Web 界面

首先打开代码库的主页，如果它复刻的上游原始代码库有新的代码提交，则可以在主页中发现这样的提示：“Thi Branch is 多少 commit behind 原始代码库”。它表示本代码库比原始代码库落后多少个代码提交，然后单击“Pull Request”可以开始代码的合并流程，如图 6-13 所示。

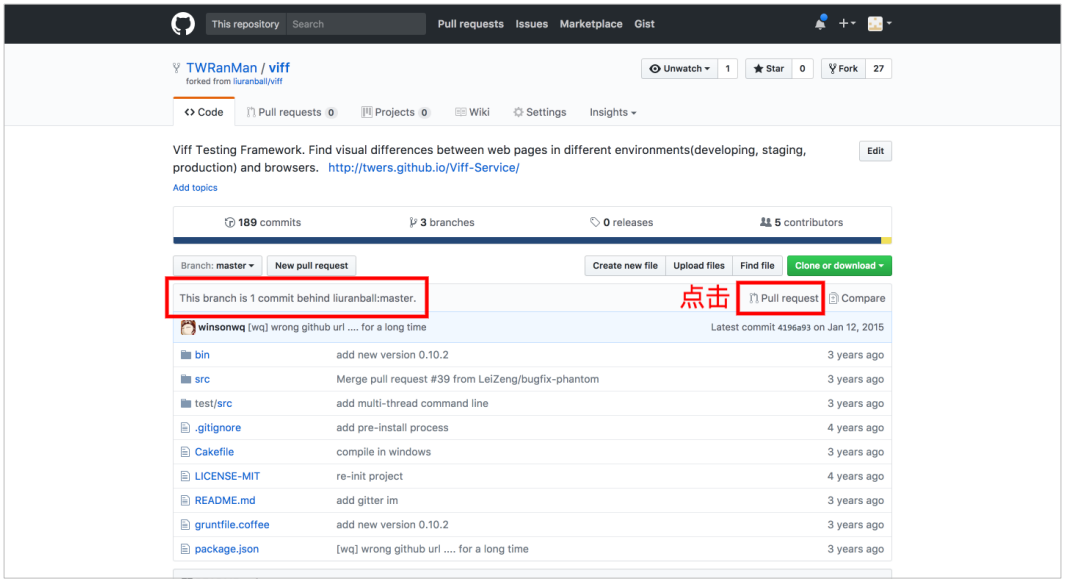

图 6-13　GitHub 多库代码合并一：打开“Pull Request”界面

“Pull Request”的第 1 个页面是代码比较页面。由于默认情况下，GitHub 的“Pull Request”将当前代码库的内容合并到被 fork 的原始代码库中，即将代码库作为 base，而将本代码库作为 head，所以不会有任何代码提交。而我们要将复刻的原始代码库合并到当前代码库中，所以需要单击“switching the base”，对“base”和“head”进行交换才能获得两个代码库之间的差异修改，如图 6-14 所示。

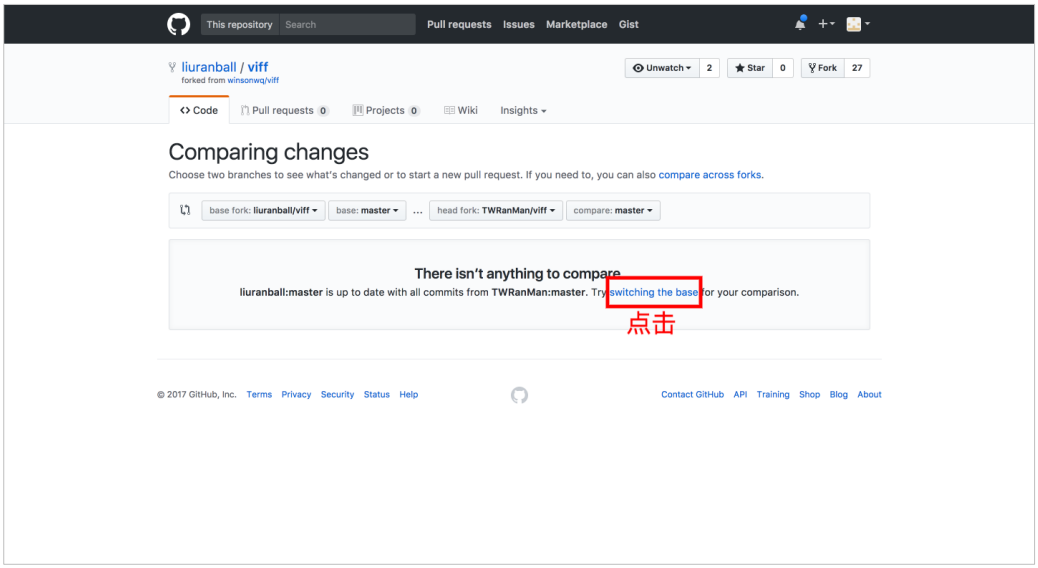

图 6-14　多库代码合并二：交换“Pull Request”的方向

在当页面上显示代码库的修改差异后，就可以审查这些代码库的修改了，如果对代码提交没有任何意见，就需要单击“Create pull request”来创建一个真正的 Pull Request。如果在代码审查中发现代码有功能性冲突的问题，则一般需要先记录下来，等到代码合并之后再来修改。

单击“Create pull request”后就会进入“Open a pull request”页面，在这里可以编辑这个 pull requst 的名字和描述，然后单击“Create pull request”，如图 6-15 所示。

下一步就是合并这个“Pull Request”了，这时需要选择恰当的合并方式，GitHub 一共提供了三种合并方式（见图 6-16）。

◎ Create a merge commit：提供标准的 git merge 方式进行代码合并。

◎ Squash and merge：如果需要合并的 commit 有多个，则首先会将它们合并成一个 commit，再对其使用 git merge 的方式来进行代码合并。

◎ Rebase and merge：提供标准的 git rebase 方式进行代码合并。

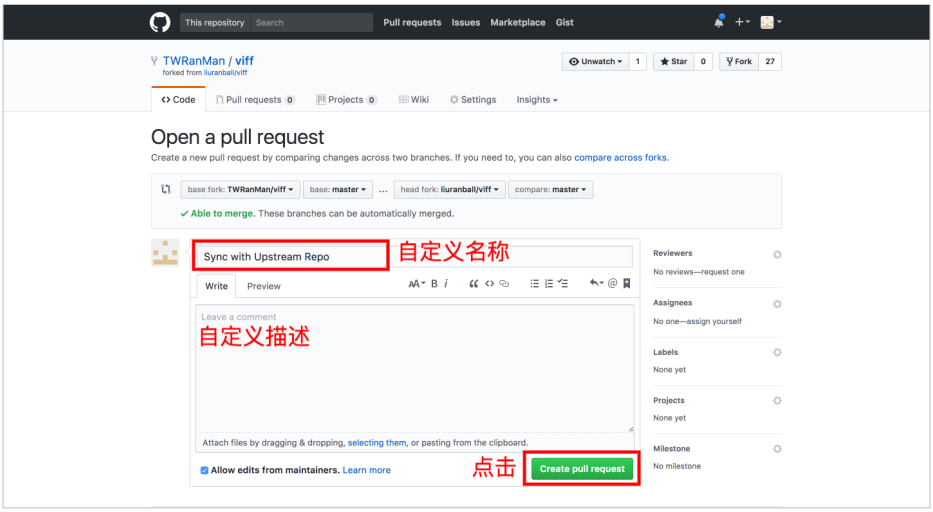

图 6-15　多库代码合并三：编辑“Pull Request”

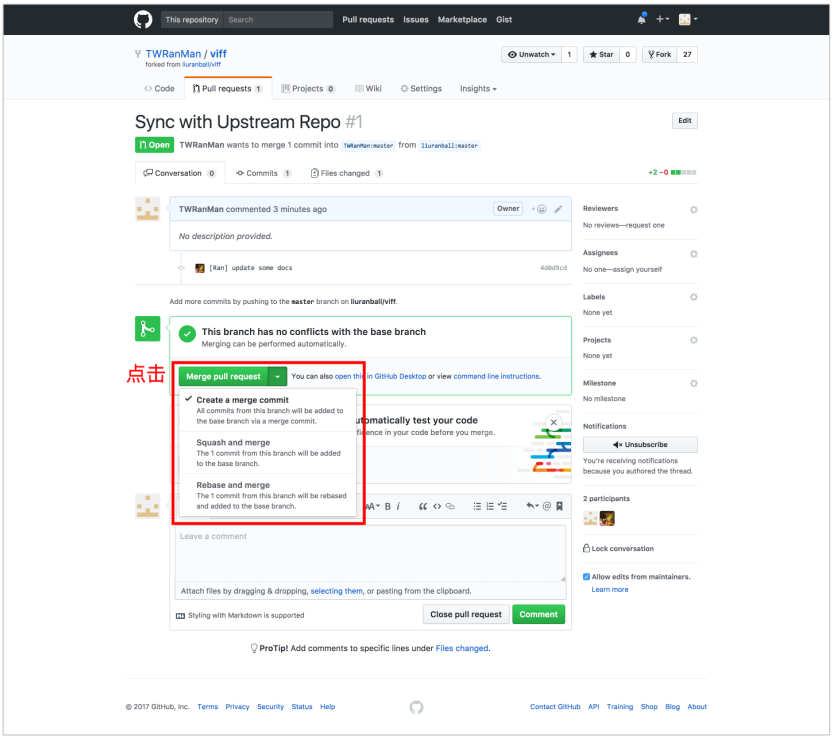

图 6-16　多库代码合并四：选择“Pull Request”合并方式

在选择了合并方法后就来到最后一个页面，单击“Confirm merge”就可以完成这个“Pull Request”的合并了，如图 6-17 所示。

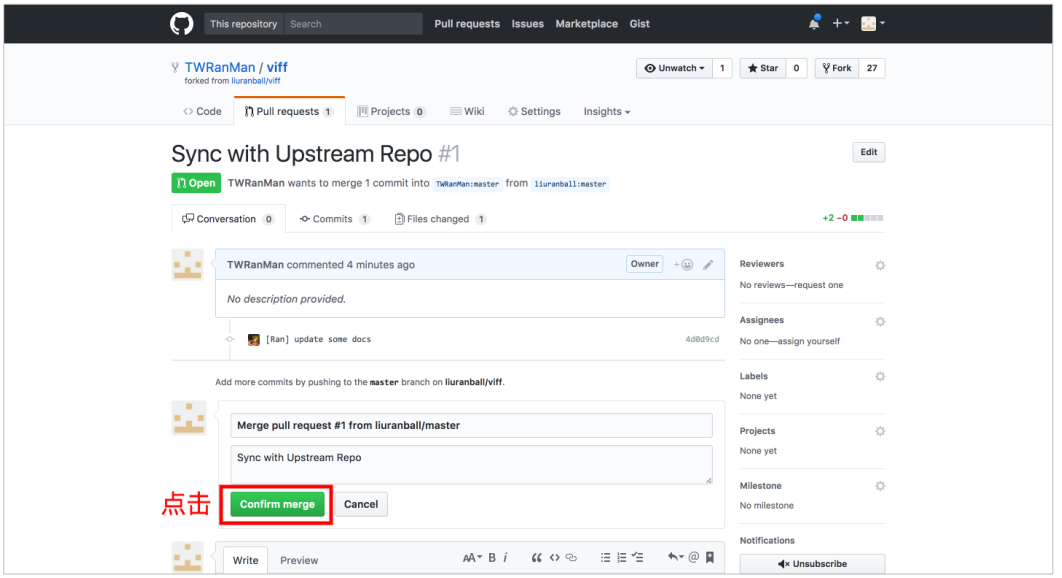

图 6-17　多库代码合并五：完成“Pull Request”合并

另外，核心贡献者可以不通过复刻代码库的方式来进行代码管理，而是直接使用GitHub的合作者功能（Collaborators）[1]将它们加入项目认证体系中，使其获得项目的代码提交等权限，如图 6-18 所示。

除了使用复刻对代码库进行复制，还可以使用传统的代码库复制方法：在本地修改 remote 的配置，直接将本地的代码库推送到远端的新代码库里，但是这样做的缺点是 GitHub 无法记录原代码库和新代码库之间的关系。

1　https://help.github.com/articles/inviting-collaborators-to-a-personal-repository/ 和 https://help. github. com/articles/adding-outside-collaborators-to-repositories-in-your-organization/

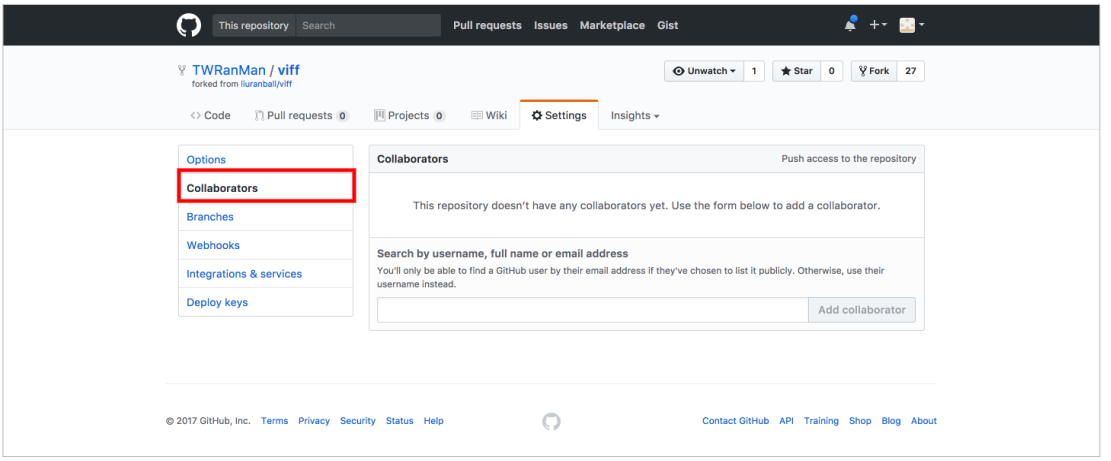

图 6-18　多库代码合并六：添加仓库合作者

6.4.3　把公开代码库转换成私有代码库

这对于从开源到商业是最为常见的操作，因为现在很多企业开发商用软件时会使用到大量的开源软件，而 GitHub 上管理着当前世界上数量最多的开源软件，也是最大的开源社区，所以使用 GitHub 来完成从开源软件到商业软件的开发工作是最为方便的。如果只是简单的使用，那么只需要直接使用 git clone 将开源代码库同步到本地代码库中，或者将原始开源代码库复刻到自己企业的开源代码库中。如果还需要基于开源项目进行二次开发，并且二次开发的内容不希望被开源，那么就需要将公开的代码库转换成私有代码库。

首先，通过 GitHub 的 Web 页面创建一个全新的私有代码库 YOUR_PRIVATE_REPOSITORY，然后复刻需要转换成私有的原始公开代码库到一个新的公开代码库（下面例子中的 YOUR_PUBLIC_REPOSITORY.git，通过 mirror 命令将公开代码库导入私有代码库中（下面例子中的 YOUR_PRIVATE_REPOSITORY.git）：

```
    $ git clone --bare https://github.com/YOUR_USERNAME/YOUR_PUBLIC_
REPOSITORY.git public-repo
    $ cd public-repo.git
```

```
$ git push --mirror https://github.com/YOUR_USERNAME/YOUR_PRIVATE_REPOSITORY.git
```

导入成功后就可以删除本地的 public-repo，并复制私有代码库进行开发工作：

```
$ git clone https://github.com/YOUR_USERNAME/YOUR_PRIVATE_REPOSITORY.git private-repo
```

然后在 private-repo 中通过添加 remote 的公开代码库的配置，可以使本地私有代码库和远端公开代码库关联起来：

```
$ git remote add public https://github.com/YOUR_USERNAME/YOUR_PUBLIC_REPOSITORY.git
```

通过 git pull 和 git push 可以让本地私有代码库获得远端公开代码库的代码更新：

```
$ git pull public master
$ git push origin master
```

相反，如果私有代码库中有一些代码提交需要合并入远端的公开代码库中，则需要在远端公开仓库中使用 GitHub 的 Pull Request，整个过程如下。

先将远端公开代码库复制到本地：

```
$ git clone https://github.com/YOUR_USERNAME/YOUR_PUBLIC_REPOSITORY.git public-repo
```

然后在 public-repo 中通过添加 remote 的私有代码库的配置，使本地代码库和远端私有代码库关联起来：

```
$ git remote add private https://github.com/YOUR_USERNAME/YOUR_PRIVATE_REPOSITORY.git
```

接着在 public-repor 中创建一个新的本地开发分支 dev-branch：

```
$ git checkout -b dev_branch
```

使用 git pull 将远端私有代码库的代码合并到本地代码库的这个开发分支（dev-branch）上，然后通过 git push 命令将本地代码库的开发分支提交到远端共有代码库中：

```
$ git pull private master
$ git push origin dev_branch
```

接着，通过 GitHub 的 Web 页面在远端公开代码库中简单地创建一个 Pull Request，将私有代码库中的指定代码提交给共有代码库的拥有者进行审核。

最后，当远端公开代码库的拥有者审核了提交的 Pull Request 并通过之后，就可以把提交的代码合并入远端的公开代码库中。

虽然复刻使得管理成本有所增加，但是复刻的公开库可以专门用来定制私有库和原始公开库之间需要同步的代码库修改。如果不使用复刻的公开库，则在定制私有库和原始公开库之间需要同步的代码库修改会变得比较困难。但是如果没有定制代码提交同步的需求，且拥有原始公开库的管理权限，那么可以省去复刻这个步骤，直接与原始公开库进行同步，在上面的例子中添加 remote 的关联的命令要从复刻的公开库改成原始公开库：

```
$ git remote add public https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git
```

6.4.4　GitHub 的分支与复刻

代码分支是代码管理系统的一个基本功能，大部分代码管理系统都支持分支，而复刻是 GitHub 首创的一种分代码库的管理方式，它的主要使用场景就是基于开源社区的分布式开发模式及任何人都可以贡献代码的原则，不希望针对每一个贡献者进行权限配置等需求。所以复刻是专门针对开源社区中的开源代码库而专门定制的。但是现在越来越多的人在企业内部的商用代码库中也使用复刻这样的管理方式，比如 Bitbucket 就支持复刻这个功能（Bitbucket 可以安装在企业内部），而使用 Bitbucket 的很多开发团队也在使用其提供的和 GitHub 类似的复刻和 Pull Request 进行代码管理。所以在 Github 上如果一个贡献者要向一个开源代码库提交代码修改，就需要复刻这个代码库来进行开发，对自己复刻的代

码库进行功能开发时则应该选择分支来进行开发。

GitHub是一个看似简单却蕴藏着丰富功能的云端代码管理系统，如果想深入研究GitHub，则应该多阅读其官方教程和文档，从而做到事半功倍。

参考文献

[1] 《Version Control with Subversion》，Ben Collins-Sussman,Brian W. Fitzpatrick & C. Michael Pilato

[2] 《Pragmatic Subversion》，Brian Fitzpatrick, C. Pilato, Ben Collins-Sussman

[3] 《Pragmatic Version Control Using Git》, Travis Swicegood

[4] 《Pro Git book》, Scott Chacon and Ben Straub

[5] 《Git 版本控制管理（第 2 版）》，on Loeliger，Matthew McCullough

[6] 《GitHub 入门与实践》，[日] 大塚弘记

[7] 《开源软件之道》，蔡俊杰

[8] http://nvie.com/posts/a-successful-git-branching-model/

[9] http://www.subgit.com/documentation.html

[10] https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow

[11] http://www.openfoundry.org/tw/foss-forum/9266-why-git-better

[12] http://heartmurs.blogspot.com/2014/09/mercurial-over-git-not.html

[13] https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

[14] http://www.kafeitu.me/git/2012/03/27/git-submodule.html

[15] http://john.albin.net/git/convert-subversion-to-git

[16] https://www.atlassian.com/git/tutorials/svn-to-git-prepping-your-team-migration/git-migration-tools

[17] http://www.drdobbs.com/architecture-and-design/migrating-from-subversion-to-git-and-the/240009175

[18] http://duanqz.github.io/2015-08-30-Intro-to-automerger

[19] http://source.android.com/source/developing.html

[20] https://en.wikipedia.org/w/index.php?title=Comparison_of_revision_control_software&printable=yes

[21] https://subversion.apache.org/faq.html

[22] https://github.com/alphagov

[23] https://www.openhub.net/p/openstack

[24] https://www.openhub.net/p/android

[25] https://en.wikipedia.org/wiki/List_of_commercial_open-source_applications_and_services

[26] http://stackalytics.com/

[27] https://en.wikipedia.org/wiki/List_of_trademarked_open-source_software

[28] https://docs.openstack.org/infra/zuul/index.html

[29] http://status.openstack.org/zuul/

名词解释

◎ 代码库：存放代码的一个仓库，英文可以是 repo 或者 repository。

◎ 代码主干：一般是指代码的主要开发分支，在不同的分支策略里用途有一定的区别，英文一般为 master。

◎ 代码分支：为了对同一套代码进行不同的开发和修改工作，需要不同的代码工作区域，英文为 branch。

◎ 代码检出（复制）：在本地没有代码库的情况下，将代码从服务器端同步到本地磁盘中，比如 git clone。

◎ 代码更新（同步、拉取）：在本地有代码库的情况下，将代码服务器中的代码更新同步到本地代码库中，比如 git pull。

◎ 代码提交：在本地完成代码修改的情况下，将代码提交到代码库中，但是不同的代码管理工具对于代码提交的定义不一样。比如对于 Git，它是将代码提交到本地代码库中；而对于 Subversion 和 Performance 等则是将代码提交到远端服务器中。

◎ 代码上传（推送）：不同的代码管理工具对于代码上传的定义不一样，比如对于 Git，它是将本地代码库中的代码提交、上传到远端代码服务中，而对于 Subversion 和 Performance 等，则等同于代码提交。

◎ 代码审查：在代码需要合并到远端代码服务器的开发代码库之前，代码库的管理者或者其他审查者会对代码进行全方位的审查和验证，并将符合要求的代码合并到目标代码库，或者拒绝不符合要求的代码提交。

◎ Pull Request：GitHub 提供的一个代码合并和审查功能，当开发者想把自己代码库中的一些代码提交（commits）合并到另一个代码库中时，如果没有权限，则

可以先提交一个 Pull Request，让另外一个代码库的拥有者在审核了这个 Pull Request 中的代码提交并同意授权后，才能将 Pull Request 中的代码提交合并到目标代码库中。

◎ Merge Request：Gitlab 提供的一个代码合并和审查功能，等同于 GitHub 中的 Pull Request。

◎ 构建：通过预处理、编译、链接、打包等步骤将源代码或者资源文件构建成目标文件，英文为 build。

◎ 编译：软件构建中的一步，使用编译器对源代码进行编译，英文为 compile。

◎ 部署：将软件的产品文件部署到产品环境中并启动起来，英文为 deployment。

◎ 发布：完成软件的开发并交付给客户或者用户使用的行为，英文为 release。