

# 版本控制之道——使用Git

# Pragmatic Version Control

## *Using Git*

程序员修炼三部曲 第一部



[美]Travis Swicegood 著  
董越 付昭伟 等译

# 程序员修炼三部曲

程序员修炼三部曲丛书包含三个部分，介绍了每个注重成效的程序员和成功的团队所必备的一些工具。

- 您在这里 ► • 第一部曲：版本控制之道  
• 第二部曲：单元测试之道  
• 第三部曲：项目自动化之道

事情正在发生变化。一些著名的项目如 Linux 内核、Mozilla、Gnome，以及 Ruby on Rails 等都已选择使用分布式版本控制系统 (Distributed Version Control System, DVCSs)，而非 CVS 或 Subversion。

Git 是分布式版本控制系统的代表，具有优异的功能和性能。然而，若缺乏指导，把 Git 应用到您所在的开发环境中可不是件容易的事。本书将向您讲述 Git，引领您进入分布式版本控制的世界。

或许您打算从传统的集中式版本控制系统

迁移到 Git 上；或许您是编程新手，刚刚接触版本控制。不论哪种情况，本书都会让您掌握 Git，以便在日常工作中使用。

《版本控制之道——使用 Git》首先概述了版本控制系统，特别是分布式版本控制系统，并展示分布方式，使您在移动性不断增加的环境中更有效地工作。接着，本书介绍了开始 Git 旅程的必备基础知识。

通过本书，您将全面了解到如何充分利用 Git 的先进功能。本书还将为您打下坚实的基础，使您掌握个人和在团队中如何运用 Git。

Travis Swicegood 是 Ning 项目 (<http://www.ning.com>，聚集了全球用户的兴趣与热情的在线社交平台) AppDev 团队的成员。他已从事专业编程工作将近十年，并愿意继续享受此项工作的乐趣，尽管曾经为生计而卖过汽车。他积极参与了 PHP 社区若干开源自动化的开发，包括自动测试框架。他也活跃于 Lawrence Programmers 社区，事实上，他是该社区的创始人。闲暇时，他常骑车出游，或者品尝自制美味，以及自酿美酒。



图书分类 软件工程>版本控制

ISBN 978-7-121-10719-1



9 787121 107191 >

定价：39.80元



策划编辑：徐定翔  
责任编辑：陈元玉  
责任美编：杨小勤

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。



程序员修炼三部曲 第一部  
The Pragmatic Starter Kit-Volume I

# 版本控制之道——使用 Git

Pragmatic Version Control Using Git

[美] Travis Swicegood 著

董 越 付昭伟 等译

电子工业出版社  
Publishing House of Electronics Industry  
北京 · BEIJING



## 内 容 简 介

《程序员修炼三部曲》丛书包含了三个部分，旨在帮助程序员解决日常工作中遇到的一些具体问题，内容覆盖了对于现代软件开发非常重要的基础知识。这套丛书展现了注重实效的实际技巧及工具使用方面的内容。

《版本控制之道》系列是三部曲中的第一部，它讲述了如何使用版本控制为项目提供安全保障，并提高开发、集成和发布的效率。随着版本控制工具 Git 越来越流行，《版本控制之道》在陆续推出了 CVS 版、Subversion 版后，现在也推出了 Git 版，即本书。

Git 是一个功能强大的工具，这也意味着完全掌握它是件颇费时间的事。而本书的特点是实用：介绍 Git 的精髓和常用的功能，让读者迅速上手，很快就可以在实际项目中使用 Git 并受益。

978-1-934356-15-9: Pragmatic Version Control Using Git.

All rights reserved. Authorized translation from the English language edition published by The Pragmatic Programmers, LLC.

本书简体中文专有翻译出版权由 The Pragmatic Programmers, LLC. 授予电子工业出版社。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2010-2385

## 图书在版编目（CIP）数据

版本控制之道：使用 Git / (美) 斯威司古德 (Swicegood,T.) 著，董越等译。—北京：电子工业出版社，2010.5  
(程序员修炼三部曲 第一部)

书名原文：Pragmatic Version Control Using Git

ISBN 978-7-121-10719-1

I .①版… II .①斯… ②董… III. ①软件工具—程序设计 IV. ①TP311.56

中国版本图书馆 CIP 数据核字 (2010) 第 068450 号

策划编辑：徐定翔

责任编辑：陈元玉

印 刷：北京智力达印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：12.75 字数：195 千字

印 次：2010 年 5 月第 1 次印刷

定 价：39.80 元

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

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：(010) 88258888。

# 读者是怎么评价 《版本控制之道——使用 Git》的

What readers are saying about Pragmatic Version Control Using Git

《版本控制之道——使用 Git》是一名优秀的向导，把你带入 Git 世界。它不仅教会你如何开始使用 Git，而且告诉你如何通过 Git 与团队成员合作，以及如何让历史记录保持整洁。

► **Pieter de Bie**

Author, GitX

如果你正在考虑开始使用 Git，那么我强烈推荐你阅读此书。如果你还没有使用任何版本控制系统，请放下这本书，给自己两下，然后重新拿起这本书，去结账。

► **Jacob Taylor**

Entrepreneur and Cofounder, SugarCRM Inc.

这本书不仅让我确信 Git 比 CVS 或 Subversion 都要好，而且它向我展示了可以自己使用 Git 并受益，同时保持与周围 CVS/Subversion 环境的连接。尽管 Git 是我接触的第一个分布式版本控制系统，期待在阅读本书后，也能尽快成为 Git 的使用者。

► **Chuck Burgess**

2008 PEAR Group Member

把 Git 这样一件复杂的事情用简单的话说明白，Travis 卓越地完成了这项任务。阅读本书，你会发现分布式版本控制系统不再是一件神秘难解的事。

► **Mike Mason**

Author, Pragmatic Version Control Using Subversion

## 联系博文

您可以通过如下方式与本书的出版方取得联系。

读者信箱：*reader@broadview.com.cn*

投稿信箱：*bvtougao@gmail.com*

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码：430074

电 话：027-87690813

传 真：027-87690595

欢迎您访问博文视点官方博客：<http://blog.csdn.net/bvbook>



# 译序

《版本控制之道——使用 CVS》原著出版于 2003 年秋，中文版出版于 2005 年春。

《版本控制之道——使用 Subversion》原著出版于 2005 年春，中文版（原著第 2 版）出版于 2007 年春。

《版本控制之道——使用 Git》原著出版于 2008 年冬，中文版（即本书）即将出版。

Git 的时代到来了。一些著名的项目如 Linux 内核、Mozilla、Gnome 及 Ruby on Rails 等都已选择使用以 Git 为代表的分布式版本控制系统（Distributed Version Control System, DVCS）。与传统的集中式版本控制系统（如 CVS、Subversion 和 ClearCase）相比，Git 具有若干显著优势，其中最重要的两点是：Git 轻而易举地支持多站点开发和离线工作；Git 的性能优于传统版本控制工具若干数量级。前者使得 Git 适用于更多的应用场景，后者使得 Git 适用于更大的研发项目。当然，即便是普通的中小型项目，也同样可以从 Git 的强大功能和高性能中受益。

《版本控制之道——使用 Git》秉承了 Pragmatic Bookshelf 系列图书的一贯传统：通俗易懂，精练实用。事实上，这一点对本书原作者是很大的挑战，因为 Git 本身博大精深，能以务实的精神讲清楚 Git，并在内容上取舍得当，实属不易。

本书由 SCMLife 社区 ([www.SCMLife.com](http://www.SCMLife.com)) 的如下成员翻译：董越（流水先生）、石振勇（Zhenyong）、林伟玲（pink\_vivi）、陈靖贤（chenjingxian）、付昭伟（v2\_smth）、李晓琴（lxq8081）、文娟（luck\_cywj）、任文科（kane）、杨素萍（michelle\_0510）、霍林莉（vino）、戴娅利（Daly）、侯颖卓（风信子）、刘贺（He1983）。他们都是富有经验的软件配置管理工作者，熟悉包括 Git 在内的各种版本控制工具。他们分工翻译了全书，并进行了交叉互审。最后，由付昭伟和董越分别进行了总审。

本书在翻译过程中添加了少量有关注释，主要有三种情形：

- 方便 Windows 上的读者使用 Git；
- 方便使用 ClearCase 的读者理解 Git；
- 对少量话题再多讲一点，讲得更透彻一点。

本书的翻译工作本身就是用 Git 管理的：每章对应于单独的文本文件；完成翻译后，译者检入并上传；评审人员通过 Git 看到不同的版本，以及每个版本的改动。英汉术语对照表是一个单独的文件，大家偶尔会同时改动它，Git 会负责合并和报告冲突。每个人工作在不同的地点，但 Git 把大家连在了一起。使用 Git，得心应手。

由衷感谢所有翻译人员，感谢他们严肃认真、尽心尽力的工作。在此也向博文视点的白爱萍、徐定翔和陈元玉三位编辑致以特别的感谢。本书的出版因为流程上的原因一波三折，耗费了三位编辑很多心力，是他们坚持不懈的努力使得本书能够在国内出版。

最后，期待所有有志于推动国内配置管理行业发展的朋友加入到我们中间来，加入到 SCMLife ([www.SCMLife.com](http://www.SCMLife.com)) 的大家庭中来。共同努力推广 Git 等配置管理技术、思想，做一名配置管理专家。

SCMLife 创始人 孙常波（懂你）

2010 年 3 月于北京



# 致谢

## Acknowledgments

是的，我是本书作者。然而，本书能够出版发行，全靠众人相助。请允许我在这里占用少许篇幅，向他们致以诚挚谢意。

感谢 Dave Thomas 和 Andy Hunt，他们把写作本书的机会提供给我这个新作者。

他们所组织的 Pragmatic Bookshelf 团队令人惊叹。感谢本书编辑 Susannah Davidson Pfalzer 所提供的无时无刻不在的帮助。

感谢所有浏览本书并提供意见和建议的人们。特别是技术评审人员：Chuck Burgess、Pieter de Bie、Stuart Halloway、Junio Hamano、Chris Hartjes、Mike Mason、John Mertic、Gary Sherman、Jacob Taylor 和 Tommi “Tv” Virtanen。也感谢 SugarCRM 和 Ning 的开发人员和其他同事，在本书写作过程中，他们给了我很多的支持。

最后，感谢我的朋友和家人，感谢他们的理解和支持。本书写作期间，我的夫人 Meg 不得不承担全部家务，却毫无怨言。没有她及其他朋友和家人的支持，本书无法面世。



# 序言

软件开发领域正在发生着全球性的变化。越来越多的软件开发团队放弃了老式的、笨重的集中式（*centralized*）版本控制系统（Version Control System, VCS），转而采用一种称为 Git 的新型的、轻量级的分布式版本控制系统（Distributed Version Control System, DVCS）。

下面是关于版本控制系统的简要介绍：版本控制系统就好比银行保险箱。保险箱保管有价值的资产，保证它们的安全。对程序员来讲，源代码就是这样的资产，由版本控制系统来保管。<sup>1</sup>版本控制系统把程序员所完成并提交（*commit*）的任何修改都记录下来，供日后查询检索。这个功能就好像银行提供的对账单一样。

在 Git 的世界里，可以随身带着有自动取款机的保险箱到处走。这意味着，你既可以完全断开和别人的连接以独立工作，也可以在适当的时候与大家沟通分享工作成果。当然，Git 也具有记录和跟踪代码修改历史这样的基本功能。当初 Linus Torvalds 创造 Git 是为了跟踪 Linux 内核的修改情况。他用几周时间完成了基本的雏形。随着时间的推移，Git 已经从最初简陋的脚本发展成为功能丰富的工具包。对程序员来说，Git 具有如下优势：

- **分布式体系结构：**可以完全断网工作，不受网络连接的限制。
- **分支与合并操作很容易：**创建分支简单、经济、快速，这与其他版本控制系统不一样。Git 把分支上的所有修改合并回父分支，即使多次，也只是一眨眼的功夫。

---

<sup>1</sup> 写作本书的时候，美国政府正对银行系统实施 7 000 亿美元的救助，所以银行可能不是最合适的地方。不必考虑这方面，只要考虑银行的工作方式就可以了。

- 跟 Subversion 进行交互：如果你是公司里唯一打算使用 Git 的人，其他人仍然在使用 Subversion，不要担心。Git 可以从 Subversion 的版本库中导入所有的历史，并把你你在 Git 中的改动发送回 Subversion 的版本库。

以上是对 Git 的一分钟介绍，本书将在此基础上展开。

## 谁适合阅读本书

每个人都可以从本书中学到新东西。没有接触过版本控制的人，也可以从头学习本书前几章，了解版本控制和 Git 的基本概念。

能够熟练使用 Subversion 或 CVS 的开发人员，可以在大致翻阅第 1 篇后仔细阅读第 2 篇，以深入细致地学习 Git 命令及其用法。

## 内容概述

本书分 4 篇：

- 第 1 篇从 Git 的视角来介绍版本控制。

第 1 章“Git 的版本控制之道”（第 3 页）从总体上介绍了版本控制和一些 VCS 基础概念，以及 DVCS 的不同之处。

第 2 章“Git 安装与设置”（第 15 页）详细介绍了 Git 的安装和配置。

第 3 章“创建第一个项目”（第 25 页）在一个简单的 HTML 项目<sup>2</sup>中演示 Git 的使用，以展示 Git 的丰富功能。

第 2 章和第 3 章须要上机操作，所以请准备好计算机。

- 第 2 篇开始展开介绍。第 4 章“添加与提交：Git 基础”（第 41 页）介绍了最基本的命令，包括准备好版本库、提交修改等。

接下来是关于分支的介绍，第 5 章“理解和使用分支”（第 55 页）。在 Git 里，分支是非常重要的操作。这里将用一整章来解释什么是分支，以及怎样使用它们。

---

<sup>2</sup> 即使不了解 HTML，也不用担心——我们的例子很简单。

学习了以上两章之后，将开始讲解如何查看历史记录：

第 6 章“查询 Git 历史记录”（第 71 页）。

第 7 章“与远程版本库协作”（第 91 页）介绍了使用远程版本库共享工作的方法。对任何版本管理工具来说，支持协作都是其关键功能，Git 也是这样。

当学习完如何使用 Git，以及如何与其他开发人员的版本库进行交互后，你还要了解一些组织管理版本库的技巧。第 8 章“管理本地版本库”（第 101 页）介绍了如何让本地版本库保持整洁。

第 9 章“高阶功能”（第 115 页）是第 2 篇的结尾，本章介绍了一些在特定情况下非常有用 的命令。

这些章节均包含了丰富的实例，但请注意，这些仅仅是起点。熟悉了这些实例以后，要进行更多的探索和实践，以便在真实项目中运用自如。

- 第 3 篇介绍系统管理。如果你所在的团队或你的公司已经有专人负责系统管理，则可以略过这部分。第 10 章“迁移到 Git”（第 131 页）介绍了如何从其他常见的版本控制系统迁移到 Git。第 11 章“使用 Gitosis 管理 Git 服务器”（第 143 页）介绍了如何利用 Gitosis 管理公共版本库。
- 本书最后是附录。附录 A（第 155 页）是命令索引，可用于快速查阅常用命令及其使用方法。

附录 B（第 165 页）介绍了 Git 配套工具，以及其网上资源的链接。其中一些工具是 Git 安装包自带的，另一些链接到在线资源，须要另行安装。

最后，附录 C（第 173 页）是参考书目列表。

## 排版约定

本书使用如下排版约定：

Git	首字母大写表示 Git 工具本身。
git	表示命令行中的命令。
黑体	表示新概念。
files 和 directories	文件和目录用这种字体显示。
prompt>	表示须要输入的命令。较长的命令将在每行行尾用“\”截为几行。当然，它们可以在一行中输入完。但为了便于排版，本书中的长命令通常分成几行。 <sup>3</sup>

## 在线资源

在第 2 篇中，每章开头给出了一个版本库，其内容与你按顺序完成本书此前各章的例子后所得到的版本库的内容相同。如果在学习时跳过了前面的一些章节，也可以从 GitHub 克隆这个版本库。在我的 GitHub 主页上可以找到这些版本库：  
<http://github.com/twicgood>

每章的开头都给出了克隆相应版本库的命令。

本书在 [pragprog.com](http://pragprog.com) 网站上有个主页<sup>4</sup>，可以浏览该页面，获得本书的相关信息。你可以在论坛里给我发消息，也可以在勘误表页面中提出建议或修正。

读到这里，现在脑海里是不是有很多关于 Git 的问题，想知道答案？不要再耽搁了，让我们马上开始学习 Git！

---

<sup>3</sup> Windows 命令行不支持 “\” 这种使用方式。

<sup>4</sup> <http://www.pragprog.com/titles/tsgit>

# 目录

Contents

致谢 .....	I
序言 .....	III
谁适合阅读本书 .....	IV
内容概述 .....	IV
排版约定 .....	VI
在线资源 .....	VI
<b>第 1 篇 欢迎来到分布式世界.....</b>	<b>1</b>
<b>第 1 章 Git 的版本控制之道 .....</b>	<b>3</b>
1.1 版本库 .....	4
1.2 版本库中存储什么 .....	5
1.3 工作目录树 .....	6
1.4 代码修改与文件同步 .....	6
1.5 跟踪项目、目录和文件 .....	7
1.6 使用标签跟踪里程碑 .....	8
1.7 使用分支来跟踪并行演进 .....	9
1.8 合并 .....	10
1.9 锁机制 .....	12
1.10 下一步 .....	13
<b>第 2 章 Git 安装与设置.....</b>	<b>15</b>
2.1 安装 Git .....	15
2.2 设置 Git .....	20
2.3 使用 Git 图形界面 (GUI) .....	22
2.4 获取 Git 内置帮助信息 .....	23

<b>第 3 章 创建第一个项目 .....</b>	<b>25</b>
3.1 创建版本库 .....	26
3.2 代码修改 .....	26
3.3 在项目中工作 .....	29
3.4 理解并使用分支 .....	32
3.5 处理发布 .....	33
3.6 克隆远程版本库 .....	37
<b>第 2 篇 Git 日常用法 .....</b>	<b>39</b>
<b>第 4 章 添加与提交：Git 基础 .....</b>	<b>41</b>
4.1 添加文件到暂存区 .....	42
4.2 提交修改 .....	45
4.3 查看修改内容 .....	48
4.4 管理文件 .....	51
<b>第 5 章 理解和使用分支 .....</b>	<b>55</b>
5.1 什么叫分支 .....	56
5.2 创建新分支 .....	57
5.3 合并分支间的修改 .....	59
5.4 冲突处理 .....	64
5.5 删除分支 .....	67
5.6 分支重命名 .....	68
<b>第 6 章 查询 Git 历史记录 .....</b>	<b>71</b>
6.1 查看 Git 日志 .....	72
6.2 指定查找范围 .....	73
6.3 查看版本之间的差异 .....	76
6.4 查明该向谁问责 .....	77
6.5 跟踪内容 .....	79
6.6 撤销修改 .....	82
6.7 重新改写历史记录 .....	85
<b>第 7 章 与远程版本库协作 .....</b>	<b>91</b>
7.1 网络协议 .....	91
7.2 克隆远程版本库 .....	94
7.3 版本库同步 .....	95

7.4 推入改动 .....	96
7.5 添加新的远程版本库 .....	97
<b>第 8 章 管理本地版本库 .....</b>	<b>101</b>
8.1 使用标签标记里程碑 .....	102
8.2 发布分支的处理 .....	104
8.3 标签与分支的有效名称 .....	106
8.4 记录和跟踪多个项目 .....	107
8.5 使用 Git 子模块跟踪外部版本库 .....	108
<b>第 9 章 高阶功能 .....</b>	<b>115</b>
9.1 压缩版本库 .....	116
9.2 导出版本库 .....	117
9.3 分支变基 .....	118
9.4 重现隐藏的历史 .....	121
9.5 二分查找 .....	124
<b>第 3 篇 系统管理 .....</b>	<b>129</b>
<b>第 10 章 迁移到 Git .....</b>	<b>131</b>
10.1 与 SVN 的通信 .....	131
10.2 确保 git-svn 是可用的 .....	135
10.3 导入 Subversion 版本库 .....	136
10.4 与 Subversion 版本库保持同步更新 .....	138
10.5 将修改推入 SVN .....	140
10.6 从 CVS 导入 .....	141
<b>第 11 章 使用 Gitosis 管理 Git 服务器 .....</b>	<b>143</b>
11.1 确定 Gitosis 所依赖的程序已经安装 .....	144
11.2 安装 Gitosis .....	145
11.3 创建管理员 SSH 证书 .....	145
11.4 配置 Gitosis 服务器 .....	146
11.5 初始化 Gitosis .....	147
11.6 配置 Gitosis .....	147
11.7 添加新版本库 .....	148
11.8 设置公共版本库 .....	150
11.9 结束语 .....	151

<b>第 4 篇 附录 .....</b>	<b>153</b>
<b>    附录 A Git 命令快速参考 .....</b>	<b>155</b>
A.1 安装和初始化 .....	155
A.2 日常操作 .....	156
A.3 分支 .....	157
A.4 历史 .....	159
A.5 远程版本库 .....	161
A.6 连接 Git 和 SVN .....	162
<b>    附录 B 其他资源和工具 .....</b>	<b>165</b>
B.1 Git 附带工具 .....	165
B.2 第三方工具 .....	166
B.3 Git 版本库托管服务 .....	169
B.4 在线资源 .....	170
<b>    附录 C 参考书目 .....</b>	<b>173</b>
<b>    索引 .....</b>	<b>175</b>







# 第1章

## Git的版本控制之道

Version Control the Git Way

版本控制系统（Version Control System, VCS）可以帮助我们记录和跟踪项目中各文件内容的修改变化。它最简单的、手工的实现形式是：复制文件以备份，在备份文件的文件名中添加上时间和日期。

然而，为了效率起见，我们希望这类操作在某种程度上是自动的。这是我们  
需要版本控制工具的原因，这类工具能够自动地备份和跟踪项目中所有代码的修改。

分布式版本控制系统（Distributed version control system, DVCS）也是这样，  
它的主要目标仍然是帮助记录和跟踪项目中所做的修改。而它与传统版本控制系统的  
区别在于，开发人员相互同步修改内容的方式不同。

本章将研究什么是版本控制系统，以及分布式版本控制系统——特别是 Git——与  
传统的集中式版本控制系统的区别。

本章内容如下：

- 什么是版本库。
- 如何判断该存储些什么。
- 什么是工作目录树。
- 如何管理和同步文件。
- 如何跟踪项目、目录和文件的内容。
- 如何使用标签来标识项目里程碑。
- 如何使用分支来跟踪并行开发。
- 什么是合并。
- Git 中的锁机制。

这些论题全都是围绕着版本库展开的。所以，让我们从学习版本库开始。

## 1.1 版本库

从“对账单”到本地仓库

版本库（Repository）是版本控制系统用来存储所有历史数据的地方。大多数版本控制系统在版本库中存储各个文件的当前状态、历史修改时间、谁做的修改，以及修改的原因。

版本控制系统就好比是银行保险箱，而它所保存的历史信息就好比对账单。每当存入一笔存款时，或者用版本控制系统的行话来说，每当进行一次提交（Commit）的时候，版本控制系统就会在“对账单”上添加一个条目，并且把提交的内容保存在版本库里。

在早期的版本控制系统中，必须登录版本库所在的服务器，才能访问这些版本库。这会带来可扩展性方面的问题。而较新的版本控制系统，比如 CVS 和 Subversion，解决了这样的问题。这类版本控制系统允许程序员通过网络来获取版本库中的代码，并在修改后提交回来。

这类版本控制系统属于集中式版本库（Centralized Repository）模式。在这种模式中，所有的程序员都会把他们的改动提交到服务器上的一个公共版本库中。具体来说，每一个程序员在本地有一个工作目录树，其内容是该版本库中最新的代码。当他们在工作目录树中完成代码修改后，就把改动提交回该版本库中。

这类使用集中式版本库的版本控制系统与早期的直接访问式版本控制系统相比，有很大进步，但仍有其局限性。首先，在本地工作目录树中，只能看到代码的最新版本。如果想查询历史修改记录，就必须与服务器上的版本库打交道。这就带来另一个问题：同远程的版本库连接，通常须要使用网络。

在这个“永不断线”的宽带互联网时代，我们几乎忘记了有时候不能上网。以本书的写作过程为例，有时候是在家中，有时候是在咖啡店里，有时候是在飞机上，还有的时候是在长途汽车上。甚至，本书最后的修订是在密苏里州奥扎克族印第安人的小木屋里。

如果使用分布式版本控制系统，就不会遇到不能上网所带来的问题。这是以 Git 为代表的分布式版本控制系统最大的优势。使用分布式版本控制系统，每个人都会在本地有自己的版本库，而不是连接到服务器上的一个公共的版本库。所有的历史记录都存储在本地的版本库中。向版本库提交代码无须连接远程版本库，而是记录在本地的版本库中。

让我们回到银行保险箱这个类比看一下。集中式版本控制系统就好比是程序员们共用一个保险箱。而分布式版本控制系统就好像是每个程序员都有他自己的个人保险箱。

那么，在分布式版本控制系统中，程序员之间如何传递各自的修改，如何同步呢？程序员还是将修改上传到项目主版本库。这有两种实现方法：可以通过 Git 的“推入”（Push）操作直接把修改上传到主版本库，也可以生成包含少量修改的补丁包，把补丁包提交给项目维护人员，再由项目维护人员更新主版本库。

## 1.2 版本库中存储什么

What does a version control system store?

对此问题的简短回答：所有内容。

稍微复杂一些的回答是：存储项目开发所必需的所有内容。这使得程序员能够修改代码，编译构建，使项目前进。显然，首先要存储的是项目源代码，否则，既无法修复缺陷也无法继续开发新功能。

大多数项目都有一些构建文件。常见的有：Makefile、Rakefile 及 Ant 的 build.xml。这些文件都要放入版本库，以方便源代码编译构建。

版本库中其他常见存储内容还有：配置文件样例、各类文档、程序使用的图片，当然还有单元测试脚本。

要判断哪些内容应该纳入版本控制很简单，自己想一下，“如果没有某样东西，我能在该项目中工作吗？”如果答案是否定的，那么这样东西就应当纳入版本控制。

但凡事皆有例外：项目中使用的开发工具，通常不用放到版本控制中去。比如，Ant 的“build.xml”应该放到版本控制中，但 Ant 工具本身不需要。

这个例外并不是这么严格的。可以考虑把 Ant、JUnit 等工具都放入版本库中，以保证大家都使用相同版本的工具。当然，最好把它们与项目本身的内容分开存放。

## 1.3 工作目录树

Working Trees

至此，我们已经介绍了版本库，以及应该在其中存储哪些内容。下面将介绍工作目录树（Working Tree），也就是程序员进行程序开发的地方。

工作目录树是版本库的一个“断面视图”。它包括了开发该项目所需要的全部文件，包括源代码文件、构建文件、单元测试文件等。

一些版本控制系统把工作目录树称为工作拷贝（Working Copy）。Git新手经常会混淆 Git 中的版本库和工作目录树。因为在 Subversion 等传统的版本控制工具中，工作目录在本地，版本库在服务器上，而 Git 中并非如此。

在 Git 中，版本库不在服务器上，而存储在本地工作目录树的“.git”目录中。这意味着，要想知道历史信息，只和本地的版本库打交道即可，无须与服务器上的版本库通信。

那么，工作目录树最初是怎么创建出来的呢？有两个方法：第一个方法是用 Git 相关命令初始化版本库，也就是生成“.git”目录，于是“.git”目录的父目录就成了工作目录树。第二个方法是克隆（Clone）一个已有的版本库，也就连带创建了相应的工作目录树。

克隆一个已有的版本库，就是创建该版本库的一个拷贝，并把版本库中主分支（Master Branch）的内容检出（Check out）到工作目录树。在 Git 中，检出是指把工作目录树更新，使其内容与版本库中某个特定的历史版本相同。关于克隆版本库，将在 7.2 节“克隆远程版本库”（第 103 页）中详细介绍。

跟踪变更是版本控制系统（CVS）的核心功能。到目前为止，我们已经介绍了版本库和工作目录树——也就是版本库的一个“断面视图”——但还没有讲解变更相关的话题，这将在下一节中介绍。

## 1.4 代码修改与文件同步

Manipulating Files and Staying in Sync

使用版本控制系统是为了记录和跟踪对文件的修改和变更。在修改了文件内容之后，须要进行单元测试以保证这次修改不会有任何负面影响，然后提交（Commit）这些改动。

每次提交操作都使得版本库中新增一个版本（Revision）。除了记录改动内容本身外，版本库还记录改动的日志信息（Log Message）或称提交留言（Commit Message），以便将来能够很方便地查询为什么要做这个改动，并能够很方便地查找某个缺陷是何时引入的。

使用像 Git 这样的分布式版本控制系统时，除了把改动提交到本地版本库之外，还要通过某种方法将改动共享，以便其他程序员能够得到。为此，须要把改动推入（Push）上游版本库（*upstream repository*）。

上游版本库是一个公共版本库。一般来说，程序员们都把自己的改动推入到这里。这里所说的推入，是指把自己所在版本库中的内容，推入到另一个版本库中。把改动推入公共版本库后，其他程序员就能“看到”了。

当然，推入操作只完成了程序员间代码同步工作的一半。另一半是，必须把别人完成的改动从公共版本库拿到本地版本库中来。

具体来说，把远程版本库里的改动拿到本地版本库中<sup>1</sup>，需要两步操作。第一步，把改动取来（Fetch），把远程版本库中的版本和分支复制到本地版本库中。这有点像是推入操作的反操作：推入操作是把本地版本库中的改动发送给另一个版本库，而取来操作是把远程版本库里的改动取到本地版本库。

第二步，在本地版本库中，把从远程版本库里取来的改动与自己本地的改动合并（Merge）。Git 工具包提供了这样的合并工具。一般来说，取来操作和合并操作总是先后执行的。因此，在 Git 中可以用一个命令完成这两步操作：拖入（Pull）。拖入操作有点像是 Subversion 或 CVS 中的更新（Update）操作。

然而，与传统版本控制工具不同的是，Git 是完全分布式的。可以把本地版本库中所做的改动推入到不只一个远程版本库中，也可以把不止一个远程版本库中的改动拖入到本地版本库中。如何与远程版本库间进行通信是理解 Git 工作方式的关键。本书第 7 章“与远程版本库间协作”（第 91 页）将介绍这方面的内容。

## 1.5 跟踪项目、目录和文件

Tracking Projects, Directories, and Files

到目前为止，我们已经介绍了如何把代码存入版本库。下面来介绍如何组织版本库中的内容。

在最底层，Git 记录和跟踪版本库中组成文件的各部分内容，而其他众多版本控制工具则是以文件为单位存储。Git 并不把整个文件，比如 `models.py`，作为不可分的整体来记录和跟踪，而是记录和跟踪组成该文件的各部分内容，也就是若干字符和代码行（它们构成了变量和函数）。Git 为这些内容添加一系列元数据，比如所在文件的文件名、文件属性，以及该文件是否为符号链接等。听起来虽只是一些细微差别，却很重要。

---

<sup>1</sup> 准确地讲，“本地版本库”是指执行 Git 操作所在的版本库。而“远程版本库”是“另外一个”版本库，它不一定要在远方的另一台计算机上，与“本地版本库”在同一台计算机上也可以。—译者注

从技术角度讲，这样做具有许多优势。首先，它能显著降低版本库存储全部历史版本所需的磁盘空间。其次，它使得“判断某个函数或类在文件间移动和拷贝的情况”之类的操作变得快捷容易。在6.5节“跟踪文件内容”（第88页）中，会有详细介绍。

而对这些文件的修改操作，则与寻常无异。所有的修改都在工作目录树中完成。工作目录树由目录和文件组成，它们是版本库中的一个“断面视图”。

版本库中的文件和目录结构的组织方式是对应于实际项目的。大部分项目遵循特定的目录组织结构，以便团队成员能够比较容易确定某个文件的存取位置。

但是，大多数公司拥有不止一个项目。甚至，一些大型项目被分解为若干模块，以便管理。Git本身并不规定如何在版本库中组织这些模块。

Git支持任意的组织方式。可以把所有项目放在一个版本库中，每个项目一个目录，以便这些项目拥有共同的版本树；也可以每个版本库只存储一个项目。

对于不熟悉版本控制领域的人来说，如何权衡以找到最佳组织方式，确实是一个挑战。第8章“版本库的组织方式”（第109页）中将详细介绍若干种组织版本库的方法。

Git的最基础部分就介绍完了，包括版本库的定义，工作目录树的定义，以及如何基于项目特点组织版本库内容等。接下来介绍标签（Tag）的使用。

## 1.6 使用标签跟踪里程碑

Tracking Milestones with Tags

不论采用何种开发方法，里程碑都标志着项目的进程：也许是敏捷开发方法，在每周一次的迭代中，不断加入新特性；也许是传统开发方法，迭代与发布之间相隔好几个月。

不论采用哪种开发方法，当到达里程碑时，都须要记录下当时的代码版本。使用标签能够记录下版本库在特定历史时刻的“断面视图”，以便于日后查找和恢复。

标签以一个简单的名称（即标签名）来标记版本库历史中某个特定的点。它可以是一个重要的里程碑，比如对外正式发布的版本；也可以是相对普通的时间点，比如在修复一个缺陷后打个标签。

本质上，标签是一个对于使用者来说易于理解易于记忆的名字，用来标识版本库中一个难读难记的内部版本号，以此帮助使用者跟踪版本历史。在介绍了标签后，下一节将介绍分支（Branch）——版本控制工具所提供的跟踪并行演进的不同轨迹的方法。

## 1.7 使用分支来跟踪并行演进

*Creating Alternate Histories with Branches*

我小时候很喜欢一套叫《Choose Your Own Adventure》<sup>2</sup>的丛书。读过此类书籍的读者也许还记得，这类书每读上几页，就要作出一个选择：Joe 钻进那个洞穴吗？若是，请翻到第 42 页；若不是，Joe 转身离开森林，则请翻到第 23 页继续阅读。

版本库类似于一本可以从头到尾按顺序阅读整个故事的书。而版本库中的分支就好像《Choose Your Own Adventure》，故事可以有多种发展方向，各自形成不同的历史轨迹。

那么，如何从版本控制系统的角度理解这类事情呢？下面举例说明：假定你打算为项目重写日历组件。从经验上看，日历的算法很难一次写对。其实不只是日历，很多其他编程也是这样，总是事后才察觉到当初犯下的错误。

为重写日历组件，须组建工作小团队，并开始管理团队成员的修改。一种简单办法是把所有的代码都拷贝到一个新的目录下，然后开始修改，但会带来两个问题：其一，无法记录和跟踪所做的修改；其二，也是很重要的一点，无法回退错误的试验性的修改。

分支可以解决此类问题。首先，在版本库中创建分支的起点。自此，两路发展平行并进。每条分支记录这条分支上发生的变更，而与其他分支隔离。

---

<sup>2</sup> [http://en.wikipedia.org/wiki/Choose\\_Your\\_Own\\_Adventure](http://en.wikipedia.org/wiki/Choose_Your_Own_Adventure)

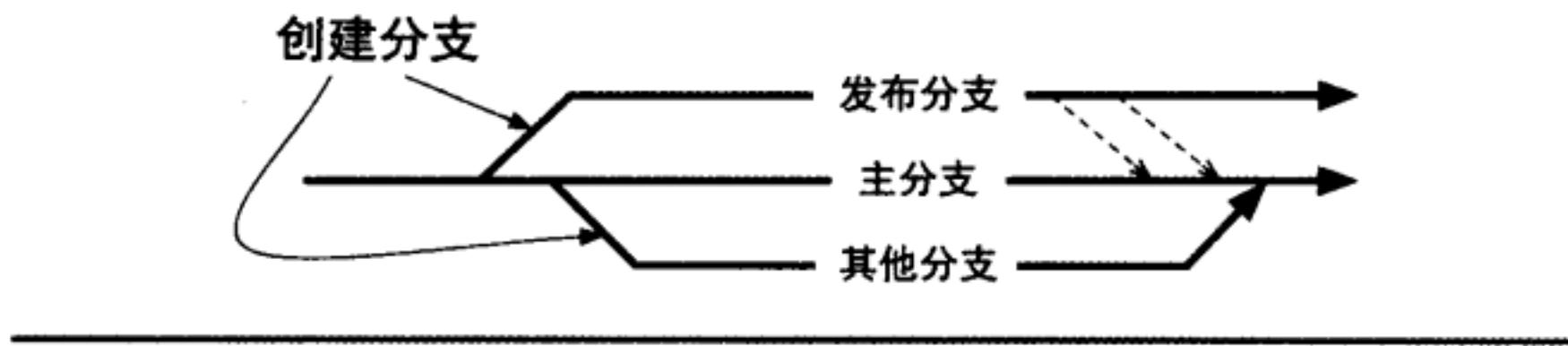


图 1.1 分支原理

图 1.1 展示了分支原理。主分支（Master Branch）是研发的主线。一些版本控制工具也把主分支称作主干（*trunk*）。

分支可以长期存在，也可以仅存在数小时。分支可以合并到别的分支，但并非所有的分支都必须合并。

有些情况下，分支不应该合并，譬如用分支来记录项目的不同发布版本的开发时；用分支来记录试验性的工作时也是这样，也许试验结束后分支就删除了。

就像 Git 中的其他事情一样，分支也可以在本地创建，并留作私用。创建本地分支并留作私用是有意义的。在完成试验性的工作后，如果有价值，再让大家拿到也不迟，而如果没价值，那就把它悄无声息地删除。

分支和标签是 Git 中的重要内容。第 5 章“理解和使用分支”（第 55 页）将用一整章的篇幅来讲述如何使用它们。

大部分分支都要合并到其他分支上去，以便让它们“跟上时代”。下面将介绍 Git 处理分支合并（*merge*）的方法。

## 1.8 合并

### Merging

是的，在《Choose Your Own Adventure》故事中，Joe 有两条道路可以选择：钻进洞穴，或者离开森林。然而，不论故事沿哪条道路发展，早晚都会走到一起来。这类书就是这样。

Joe 可能从钻入洞穴开始，最后抵达一条河的岸边。而这条河正是从洞穴所在的森林蜿蜒流过来的。因此如果 Joe 顺着河走，也会到达这里。有时候，本书作者找不到更多的地方可以让 Joe 去了，所以 Joe 不得不多次回到同一个地方。

类似的，有几条甚至几十条分支可能是合理的，但它们的轨迹有时可能会重新聚集到一起。这时就会发生合并。

合并操作把两条或两条以上的分支合并到一起。Git 自动处理分支合并的方法与程序员人工处理的方法是相同的：Git 比较各分支上的变化，确定变化在哪里发生——哪个文件的哪个位置。

当不同的变化发生在文件的不同部分时，Git 能够自动合并。但情况并不总是这样理想，当 Git 不能自动合并时，就会提示冲突（conflict）。

假定团队中的两名程序员，为不同的目的，都修改了同样一行代码。Git 检测到这种情况，就会做出无法自动合并的结论，并把这个地方作为一个冲突标识出来，等待人工介入。

幸运的是，Git 有好几种解决冲突的方法。知晓其原理，是精通 Git 的重要一步。在本书 5.4 节“冲突处理”（第 64 页）中，将介绍所有相关工具。

有这么多的分支及其合并，你可能会想“如何才能记录和跟踪我所做的合并呢？”

Git 的使用者无须手动做这样的记录和跟踪，因为 Git 提供自动记录和跟踪合并的功能，称为合并跟踪（*merge tracking*）。就像这个名词所体现的，Git 记录和跟踪哪些提交已经合并到一起了，以避免重复合并。

多数传统版本控制系统都不具备 Git 的这种功能，比如 CVS。Subversion 也是不久前才具备这一功能的。<sup>3</sup>如果使用不具备合并跟踪功能的工具，使用者就不得不人工记录哪些提交合并到哪儿了。人工记录此类信息会使得分支的使用变得繁琐。这正是过去很多程序员不喜欢使用分支的重要原因。

---

<sup>3</sup> Subversion 从 1.5.0 版（2008 年 6 月发布）起支持合并跟踪。

## 1.9 锁机制

### Locking Options

一本书从图书馆借出到归还之前，其他人无法从图书馆将它再次借出。

原因很简单：一本书在物理上无法同时给两个人。很多人第一次在版本控制领域听到检出（*check out*）这个词的时候，都想当然地认为，同时只能由一个程序员检出——就像图书馆借书一样。<sup>4</sup>

有些版本控制系统确实如此。当程序员从版本库检出某个文件时，版本库禁止任何其他人修改这个文件，直到该程序员检入（*check in*）为止。

这是一种锁机制，准确地说，是严格锁（*strict locking*）。就像在图书馆，同时只有一个人可以获得该文件。这显然不是一种有效的团队开发方法，并且也与分布式版本控制模型相冲突——在分布式版本控制模型中，程序员们彼此之间是弱连接。

另一种锁机制，乐观锁（*optimistic locking*）<sup>5</sup>，是大多数版本控制系统和所有分布式版本控制系统所使用的。乐观锁允许多个程序员同时修改同一文件。乐观锁机制基于一个假定：大多数时候，这种并发修改不会引起冲突。

下面举例说明：Joe 和 Alice 都克隆了公共版本库到本地，并开始修改代码。他们修改了同一个文件的不同部分。Alice 把她的修改推入公共版本库，随后，Joe 试图把他的修改也推入公共版本库。

Joe 的尝试将会被 Git 拒绝，因为 Git 检测到，在 Joe 克隆公共版本库后，公共版本库上有新的变化。于是 Joe 不得不先把这些新的变化从公共版本库拖入他的本地版本库，解决可能存在的冲突，然后再把他自己的修改推入服务器上的公共版本库。

这听起来颇为复杂，但事实上并不难。实际上，很少有两个程序员同时修改同一个文件，因此，版本合并并不经常发生。即使发生，大多数情况下，Git 也能自动合并。

---

<sup>4</sup> 英文中，从图书馆借出图书和从版本库中检出文件，对应的英文单词都是 *check out*。一译者注

<sup>5</sup> 在计算机科学中，这种锁有时候被称为乐观并发控制。

### 配置管理（Configuration Management, CM）

你可能听说过配置管理（Configuration Management, CM）这个术语，以及配置管理工具。这类工具协助管理应用程序在不同版本中的配置。它们大多基于版本控制系统，支持对配置历史的记录和跟踪，但它们本身并不是版本控制系统。

## 1.10 下一步

Next Steps

本章概览了分布式管理系统及其代表 Git，包括：版本库的概念，版本库里要存储哪些内容（项目相关的几乎全部内容！），工作目录树的概念，文件是如何被记录和跟踪的，版本库之间是如何同步的，以及标签、分支和合并等相关知识。

这些概念能让你对 Git 和分布式管理控制系统有一个基本的理解。接下来，将介绍 Git 的安装和配置。





## 第 2 章

# Git 安装与设置

Setting Up Git

在使用 Git 前，必须先安装并做一些基本设置。这些操作很简单。本章将讲述如下内容：

- 安装 Git。
- 设置 Git。
- Git 图形界面介绍。
- 如何使用 Git 内置帮助。

掌握这些内容后，就可以开始学习 Git 的日常使用了。

### 2.1 安装 Git

Installing Git

对 `Makefile` 很熟悉并经常用 `make` 工具安装新软件的读者，可以轻松阅读本节的大部分内容。对此了解不多的读者也不必担心，本节将详细介绍 Git 安装中的相关内容。

Git 起源于 Linux 世界，因而继承了 Linux 的风格。这意味着，在早期，安装 Git 必须下载它的源代码并编译。近来发行的 Linux 大都自带了 Git 安装包，Mac OS X 上的 Fink<sup>1</sup> 和 MacPorts<sup>2</sup> 也是这样。而在 Windows 上，则有多种安装 Git 的方法。本节“在 Windows 环境中安装 Git”部分（第 17 页）将介绍相应内容。

---

<sup>1</sup> <http://www.finkproject.org/>

<sup>2</sup> <http://www.macports.org/>

## 在 Linux 环境中安装 Git

不同的 Linux 发行版本可能选用不同的安装包管理工具，不论选用哪种安装包管理工具，大多数 Linux 发行版本中都包含 Git 安装包。但要注意，由于 Git 诞生还没几年，它的安装包可能存在于安装包存储库的 development(开发中)或 unstable(不稳定的)类别之下，而非默认的类别之下。

使用这些安装包的唯一问题是，它们的版本可能太老。本书中的例子须要使用 Git 1.6.0 或更高版本。

举例来说，本书写作时，Ubuntu 上有两个与 Git 相关的安装包：`git-core` 和 `git-doc`。然而，它们都只是 1.4.4 版。

因此，建议大家从 Git 网站<sup>3</sup>上直接下载 Git 的最新版本的源代码，然后自己编译。

在 Ubuntu 环境中，首先须要安装 `build-essential` 安装包。此外，Git 还依赖于少量其他安装包，好在它们并不常变化。可以使用下述 `apt-get` 命令来列出所有依赖关系：

```
prompt> sudo apt-get build-dep git-core git-doc
```

该命令列出了 `git-core` 和 `git-doc` 两个安装包所依赖的各个安装包。它们也是源代码编译安装 Git 中所依赖的全部安装包。

当上述依赖安装包安装完成后，就可以编译 Git 源代码并安装 Git 了。首先，从 Git 网站上下载 Git 源代码包，并解压缩到适当路径。然后，在该路径下，运行下述命令：

```
prompt> make prefix=/usr/local all doc
```

这个命令将为该计算机上的所有用户安装 Git。如果只为当前用户安装，则在命令行中去掉 `prefix=/usr/local` 参数。编译 Git 源代码将花费数分钟时间。在编译完成后，运行下述命令安装 Git：

```
prompt> sudo make install install-doc
```

完成安装后，在命令行中运行 `git --version`，可以检查 Git 的安装情况：

```
prompt> git --version  
git version 1.6.0.2
```

---

<sup>3</sup> <http://git.or.cz/>

如果不采用下载 Git 源代码并编译的安装方法，也可以咨询你所使用的 Linux 发行版的网上社区或供应商，以期获得更高版本的 Git 安装包。

## 在 Mac OS X 环境中安装 Git

在 Mac OS X 环境中，可使用安装包管理工具 MacPorts 来安装 Git。使用 MacPorts 能够获得比较新的 Git。命令如下：

```
prompt> sudo port install git-core +svn +doc
```

这个命令安装了 Git 和 git-svn 工具。第 10 章“迁移到 Git”（第 131 页）中将介绍 git-svn。

Google Code<sup>4</sup>上也提供一种 Git 安装包。它不仅介绍了 Git 的安装，还介绍了如何在 Finder 工具里加上一个 Git 的链接图标，以方便在使用 Finder 时调用 Git。

如果采用下载并编译 Git 源代码的方式安装 Git，则须要先安装 Xcode<sup>5</sup>——苹果公司的软件开发工具。此后，编译 Git 源代码的命令与在 Linux 环境中的情况很相似。

不同点在于，在 OS X 中，不建议在 make 命令中使用 doc 和 install-doc 参数。原因是安装包依赖关系方面的复杂情况。

不论是从安装包安装还是从源代码安装，都可以使用命令 git --version 来查看安装情况：

```
prompt> git --version  
git version 1.6.0.2
```

## 在 Windows 环境中安装 Git

Git 开发团队不太重视对 Windows 的支持，原因很简单，Git 开发团队起源于 Linux 核心开发社区。然而，随着 Git 越来越流行，人们开始致力于把 Git 变成完全跨平台的工具。当前，在 Windows 环境中，有如下几种方法安装 Git。

---

<sup>4</sup> <http://code.google.com/p/git-osx-installer/>

<sup>5</sup> 可从 <http://developer.apple.com/tools/xcode/> 免费下载，也可从 Mac 自带的安装盘中获得。

## Cygwin

安装和运行 Git 的“官方”方法是通过 Linux 仿真器 Cygwin。但这会给大部分使用者带来一些难度，包括但不限于 Cygwin 本身的安装和使用。如果确实要使用这种方式，可在 Cygwin 的安装包存储库的 Devel 类别下找到 Git 安装包。

在安装 Git 前，须要先安装其依赖的若干安装包。首先是 Net 类别下的 **openssh**，因为 Git 使用的默认传输协议是 SSH，Git 通过它来与远程版本库通信。

其次，如果计划使用 Git 和 Subversion 的集成方案，则须要安装 Libs 类别下的 **subversion-perl** 安装包。关于 Git 和 Subversion 的集成方案，请参见第 10 章“迁移 to Git”（第 131 页）。此外，如果打算使用 Git 的图形界面，则须要安装 Libs 类别下的 **tcltk** 安装包。关于 Git 的图形界面，请参见 2.3 节“使用 Git 的图形界面”（第 22 页）。

最后，要从 Editors 类别下选择并安装一个文本编辑器，以方便使用 Git 中的一些与提交留言相关的功能。用 Vim、emacs，甚至 Pico 都是可以的。如果不了解什么是 Git 提交留言，也不必担心，下一章将介绍这方面的内容。

## MSys 版的 Git

Git 本身的一个版本分支是，基于 Windows 底层结构的 Git。本书写作时，这一方法仍不是“官方”方法，但是这一版本分支正在合并回 Git 的开发主线。

在 Google Code 上有一个叫“Git on MSys”的项目。基于上面介绍的 Git 版本分支，该项目致力于创建一个 Windows 下易于安装的 Git 安装包。这个安装包可以从该项目的网站上下载。<sup>6</sup>

下载该安装包后，双击鼠标，开始安装。第一个对话框（见图 2.1）显示这个软件的一些信息。点击“继续（Next）”键进入到下一步。

下一步是关于许可证及 Git 的 Windows 版与\*nix 版间不兼容性的说明。不兼容性包括不同的换行符等。

---

<sup>6</sup> <http://code.google.com/p/msysgit/>

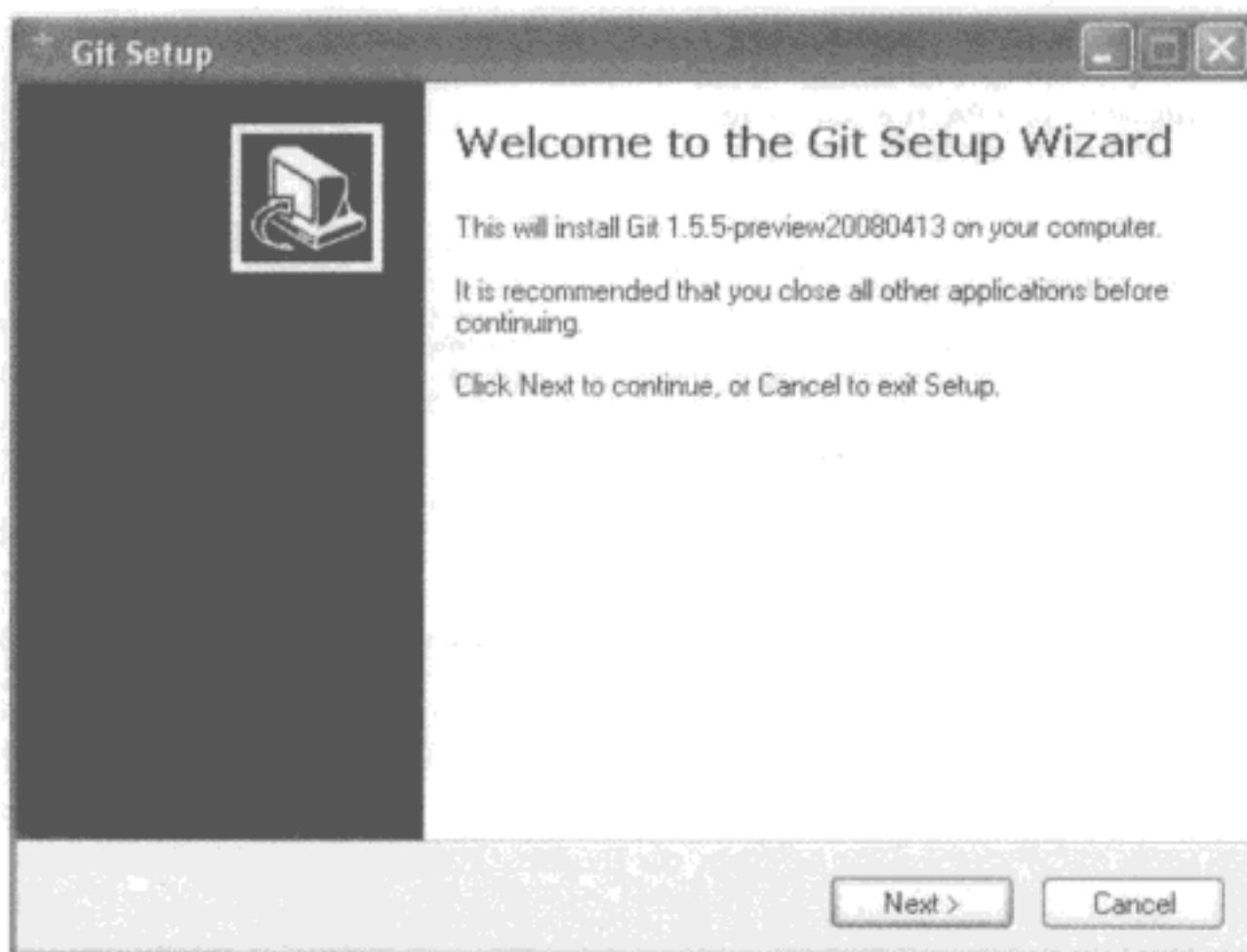


图 2.1 Git 安装的欢迎页面

不要担心这些改换，这都是为了 Git 能在 Windows 环境中更好地运行。<sup>7</sup>

接下来的两步是确定 Git 安装路径，以及确定在开始菜单中 Git 位于哪个程序组的。取默认值就可。

接下来的一步是选择一些增强功能。这些也取默认值就可。

下一步（见图 2.2）是把 Git 添加到 PATH 环境变量中。这一步很有必要，它是为了让 Windows 能够找到 git 命令。

一共有三种选择。一般来说，请选择第二种，以便 Git 可以从 Windows 命令行环境中直接运行，同时又不会因为添加了额外的设置而给 Windows 命令行环境带来潜在的问题。

接下来安装程序开始安装。在不同的计算机上，可能需要几秒到几分钟的时间。当安装完成后，可以从 Windows 命令行窗口中运行命令 `git --version` 来检验安装正确与否。

<sup>7</sup> 较新版本的 Msys 版 Git，在安装时可以选择其支持换行符的不同方式。详见安装时的提示说明。  
—译者注

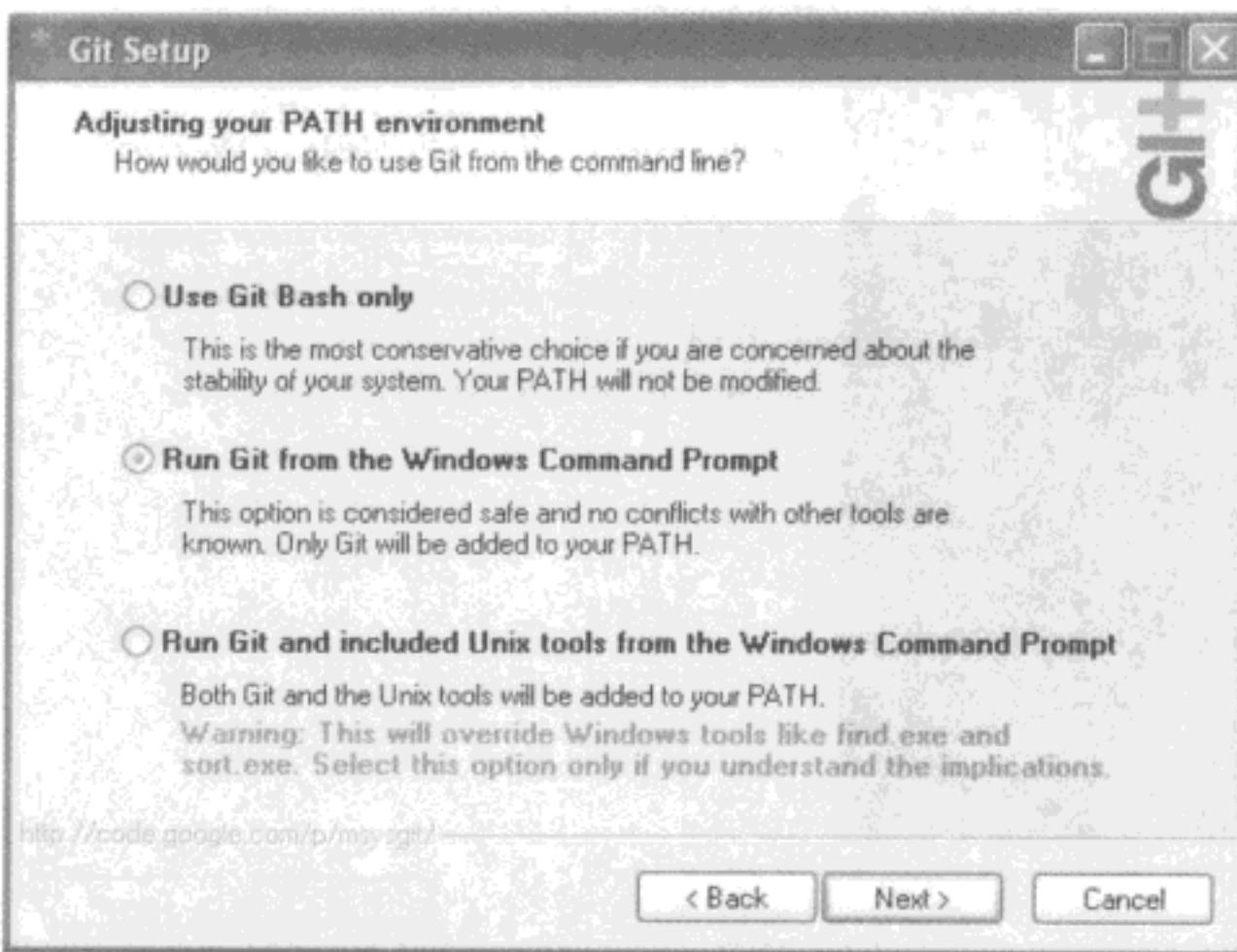


图 2.2 Git 的额外选项

具体做法是，点击“开始”按钮，然后点击“运行”。在弹出的对话框中，输入 cmd，然后点击“确认”按钮。如果一切顺利，Git 将显示它的版本号，如图 2.3 所示。

不论在哪个操作系统上使用 Git，使用前，都要做一些基本的设置。下一节介绍这方面的内容。

## 2.2 设置 Git

### Configuring Git

Git 需要用户提供若干信息。Git 是分布式的，因此不存在向用户询问姓名和邮件地址的中央版本库。通过命令 `git config`，用户可以把此类信息提供给本地版本库。

须要设定一些全局变量的值。“全局”的意思是，你在这台计算机上使用任何 Git 版本库时，这些全局变量的值都起作用。在相关命令中加上`--global`参数可设置全局变量值。

首先设置 `user.name` 和 `user.email` 两个全局变量值。第一个值用来说明版本历史上的一个修改是谁提交的。

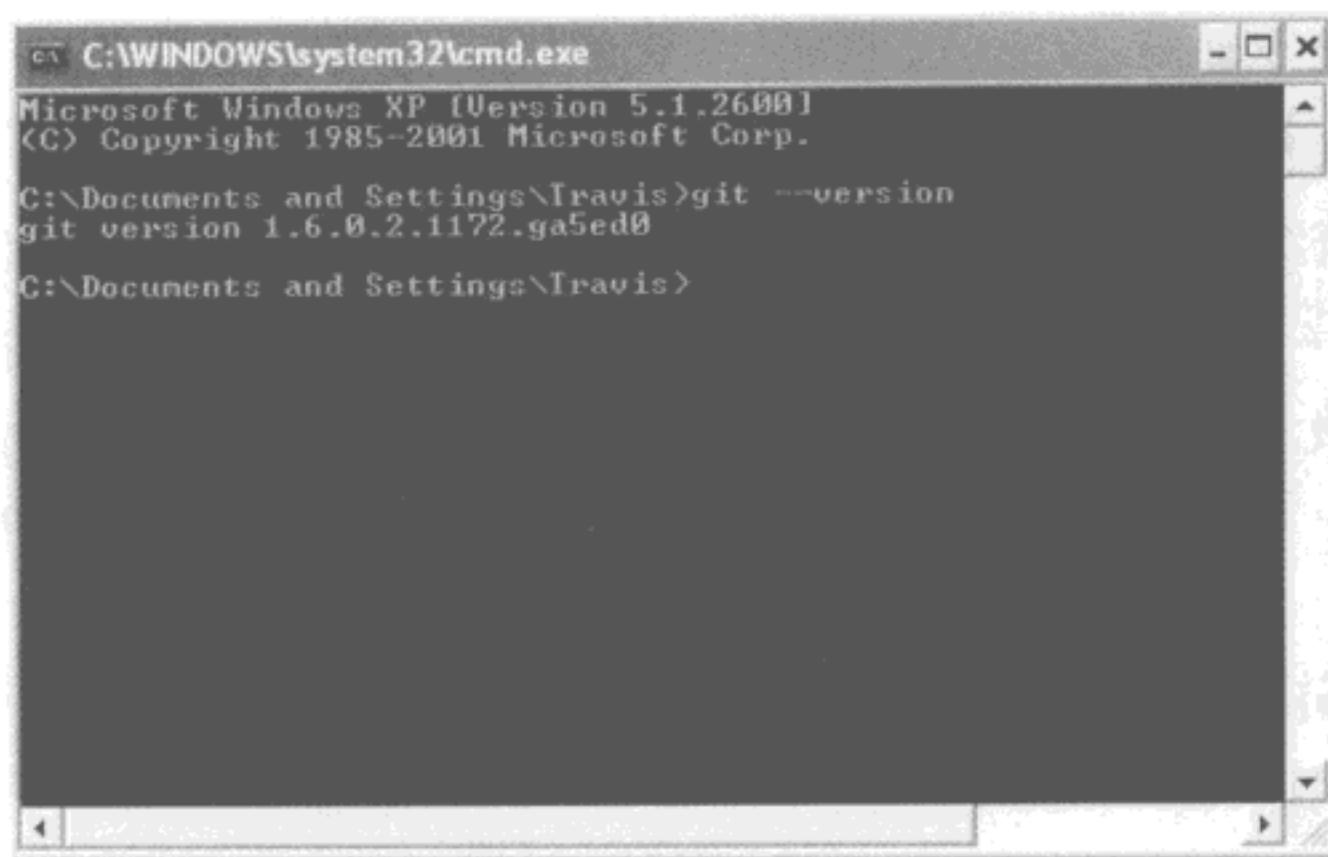


图 2.3 在 Windows 环境中显示 Git 的当前版本

第二个值是邮件地址，以方便联系修改者。

示例如下，当然，实践中你得换成自己的名字：

```
prompt> git config --global user.name "Travis Swicegood"
prompt> git config --global user.email "development@domain51.com"
```

用下列命令可检查上述设置是否成功：

```
prompt> git config --global --list
user.name=Travis Swicegood
user.email=development@domain51.com
```

仅这两个全局变量值是必须设置的。但所有可设置的值超过 130 个。<sup>8</sup>其中大部分都很少用得上。但有一个例外：关于用不同颜色显示 Git 命令的输出。

若想在命令行窗口中使用不同的颜色显示不同类型的内容，请将“color.ui”的值设为“auto”或“always”。<sup>9</sup>

<sup>8</sup> 如果在本机中安装了 Git 文档，则可通过运行命令“git help config”列出所有设置变量值，也可以访问这个网页：[http://www.kernel.org/pub/software/scm/git/docs/git-config.html#\\_variables](http://www.kernel.org/pub/software/scm/git/docs/git-config.html#_variables)。

<sup>9</sup> 建议使用 MSys 版 Git 的用户把该值设为“always”而非“auto”，这样不仅在 Bash 环境中是彩色的，在 Windows 命令行环境中也是彩色的。一译者注

具体命令如下：

```
prompt> git config --global color.ui "auto"
```

很多开发人员喜欢用不同颜色显示——特别是查看版本间差异（Diff）的操作——但也有一些开发人员不喜欢这种方式。因此“color.ui”的默认值是“false”。

这些就是在使用 Git 命令行之前要做的全部设置工作了。在介绍第 3 章“在 Git 中创建第一个项目”（第 25 页）前，本章还将介绍 Git 的图形界面及其相关的设置。

## 2.3 使用 Git 图形界面（GUI）

Using Git's GUI

Git 自带基于 Tcl/Tk<sup>10</sup>的图形界面。在命令行窗口中，在 Git 工作目录树下键入 `git gui`<sup>11</sup> 可启动 Git 图形界面。在一些操作系统的文件管理器中选择了 Git 工作目录树后，可通过在上下文菜单中选择“git-gui”来启动 Git 图形界面。

Git 图形界面如图 2.4 所示。本书中介绍的 Git 命令基本上都可以通过它来完成。但在介绍这些 Git 命令时，不会特别讲解如何使用 Git 图形界面来完成。喜欢使用图形界面的读者，请自行实践。

使用 Git 图形界面可完成提交等操作，但不能完成查看历史信息等操作——那是工具 `gitk` 的职责。在命令行窗口的 Git 工作目录树下键入 `gitk` 就可以启动。<sup>12</sup>

`gitk` 显示版本库中的历史记录。在 `gitk` 命令中添加 `--all` 参数，可以显示全部分支的历史，而不仅是当前分支的历史。

在 Mac OS X 操作系统中，可使用另一个图形界面工具 `GitX`。它由 Pieter de Bie 编写，模仿 `gitk` 功能。在 GitHub 相关网页<sup>13</sup>中可下载该工具。

在介绍 Git 安装时，我们提到了 Git 文档，下一节将介绍如何使用它们。

<sup>10</sup> Tcl/Tk 是一个跨平台的图形界面工具包。

<sup>11</sup> 这条译注写给长期工作于 Windows 环境下的朋友。如果是在类 Unix 环境下——比如在 Cygwin 或 MSysGit 自带的 Bash 下——运行 `git-gui` 或 `gitk` 命令，可以在命令后面加个“&”号。这样，图形界面就会在新进程中运行，不会导致你的命令行窗口“死掉”。一译者注

<sup>12</sup> 在 Git 的图形界面中，版本树总是向上生长的，像真实的树一样。在本书的一些配图中，版本树是从左至右生长的。在一些版本控制工具（如 ClearCase）的图形界面中，版本树是从上向下生长的。一译者注

<sup>13</sup> <http://gitx.frim.nl/>

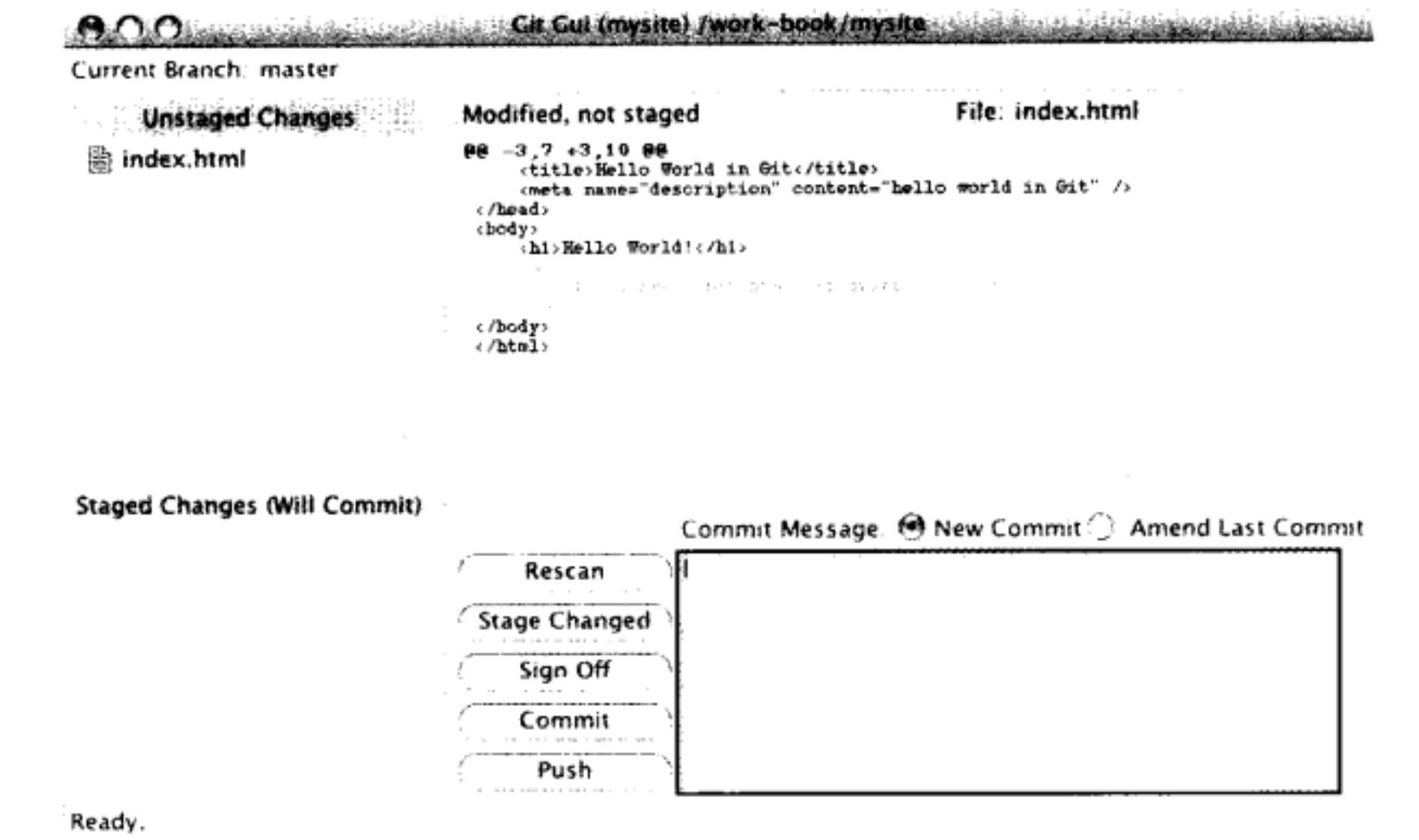


图 2.4 Git 图形界面

## 2.4 获取 Git 内置帮助信息

### Accessing Git's Built-in Help

Git 内置丰富的用户手册，可以从命令行访问，也可以在线访问<sup>14</sup>。在命令行环境中键入：

```
prompt> git help <command>
```

其中，用希望了解的具体命令名称替代<command>。Git 将显示用户手册中该命令的描述。

如果从 Git 源代码安装 Git，默认情况下不会安装 Git 文档。一些安装包管理工具把 Git 文档当作单独的安装包，并称其为 `git-doc`。

如果从 Git 源代码安装 Git，要安装 Git 文档，须把 `doc` 和 `install-doc` 设为 `make` 命令中的“target”。Git 需要 AsciiDoc<sup>15</sup>来把文档格式从纯文本转化为手册格式。

如果在安装 Git 文档时出现问题，不必烦恼，因为可以在线浏览 Git 文档，或

<sup>14</sup> <http://www.kernel.org/pub/software/scm/git/docs/>

<sup>15</sup> <http://www.methods.co.nz/asciidoc/>

者把文档下载<sup>16</sup>到本地。请在该网页上查找以 `git-htmldocs` 开头的文件并下载。

至此已介绍完 Git 的安装和基本设置，可以开始使用 Git 了。下一章将用一个简单项目来演示 Git 的基本功能。

---

<sup>16</sup> <http://kernel.org/pub/software/scm/git/>



# 第3章

## 创建第一个项目

Creating Your First Project

现在是进入 Git 世界的时候了。到目前为止，我们已经介绍了一些 Git 中的概念，以及系统的安装启动方法。

从本章开始，将以一个很小的 HTML 项目为例，使用 Git 来记录和跟踪这个项目。不熟悉 HTML 的读者请不要担心，这个例子中使用的 HTML 语言很简单也很容易理解。

本章将介绍以下内容：

- 创建版本库。
- 添加与修改文件。
- 创建新分支。
- 打标签并整理版本库。
- 克隆版本库。

完成本章的学习后，你将掌握使用 Git 的基本技能。本章是一个概览。在学习本章过程中，会介绍很多概念和命令。

本章仅简要介绍这些概念和命令。欲览详情，请参阅后面的相关章节。

请读者尽量跟随本章及其他章节的示例，动手执行这些命令，在反复实践中学习 Git，无论它多么简单，都请在命令行中输入这些 Git 命令<sup>1</sup>。

<sup>1</sup> 如果你对这种学习方式本身感兴趣，请参见 Andy Hunt 的 *Pragmatic Thinking and Learning* [Hun08]。

## 3.1 创建版本库

Creating a Repository

在 Git 中创建版本库是很简单的，但是对习惯了 Subversion 或 CVS 的读者来说，会感觉有点不适应。在多数版本控制系统中，版本库的一个断面的拷贝（即工作目录树）与版本库本身是分开存放的。而在 Git 中，版本库（`.git` 目录）是与工作目录树并排放在同一个目录中的，见下文。

在 Git 中创建版本库，首先要决定把项目源代码存放在哪里。本例中，要创建一个简单的 HTML 页面，所以给这个项目取名为 `mysite`。首先创建一个同名目录“`mysite`”，并进入到这个目录，然后输入命令 `git init`。整个过程如下：

```
prompt> mkdir mysite
prompt> cd mysite
prompt> git init
Initialized empty Git repository in /work/mysite/.git/
```

创建完成。从现在开始，这个 Git 版本库就可以用来记录和跟踪该项目的代码了。

相比其他版本管理系统，在 Git 中建立一个版本库是很简单的。命令 `git init` 会创建一个 `.git` 目录。这个目录用来存放版本库的全部元数据。`mysite` 目录作为工作目录树，存放从版本库中检出的代码。

## 3.2 代码修改

Making Changes

前面的操作已经创建了一个空版本库，现在该往里添加文件了。现在我们创建一个名为 `index.html` 的文件，并添加标题文本“Hello world”。详细内容如下：

```
<html>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

创建了一个简单的 HTML 文件后，就可以开始跟踪版本了。我们会向这个文件不断添加更多的内容。要想让 Git 跟踪这个文件，须先让它知道这个文件要分两步走：首先使用 `git add` 命令把该文件添加到版本库的索引（`index`）；然后使用 `git commit` 命令提交。

```
prompt> git add index.html
```

```

prompt> git commit -m "add in hello world HTML"
Created initial commit 7b1558c: add in hello world HTML
1 files changed, 5 insertions(+), 0 deletions(-)
create mode 100644 index.html

```

文件或文件列表可以作为 `git add` 命令的参数。本书 4.1 节“添加文件”（第 42 页）将会介绍其他几种添加文件的方法。

`git commit` 命令创建一个提交记录。提交记录是存储在版本中的历史记录，每提交一次创建一个记录，并标记出代码的演进。<sup>2</sup>Git 把提交者的姓名和邮件地址（上一章中我们设置了它们），以及提交留言，都添加到提交记录中。

前面命令中参数`-m` 的作用是，告诉 Git 本次提交的提交留言为 `add in hello world HTML`。对于任何版本控制系统，适当书写的提交留言都是极其重要的。它可以说明提交的原因：新添加的文件是做什么用的？修改那行代码的原因是什么？关于如何书写一个适当的提交留言，请参看 31 页的“Joe 问……”。

现在我们已经提交了一个文件到版本库中。运行命令 `git log` 可以看到这个提交相关的信息：

```

commit 7b1558c92a7f755d8343352d5051384d98f104e4
Author: Travis Swicegood <development@domain51.com>
Date:   Sun Sep 21 14:20:21 2008 -0500

```

```
add in hello world HTML
```

命令 `git log` 运行后输出的第一行显示提交名称，该名称是 Git 自动产生的 SHA-1 码。Git 通过它来跟踪提交。Git 使用该哈希码可以保证每个提交的名称都是独一无二的。这在分布式的环境中非常重要。如果想了解更多的原因，请参看下一页的“Joe 问……”。

命令 `git log` 输出的提交名称的前七位字符和命令 `git commit` 输出的相同，这是因为命令 `git commit` 显示的是提交名称的缩写。显然，要把 40 位的 SHA-1 哈希码都显示出来就有点长了。

用命令 `git commit` 显示的七位字符来表示一个提交，通常已经足够了，没有必要使用完整的 40 位哈希码。为了保持简洁明了，本书将一直使用这种缩写格式。但是命令 `git log` 将显示完整的 40 位哈希码。

---

<sup>2</sup> 在 ClearCase 等工具中，版本是文件级的。而在 Subversion、Git 等工具中，版本是系统级的。也就是说，在一次提交发生后，版本库就会“长”一个版本。而每一个版本，对应于版本库作为整体的一个“断面”。另一个重要区别是，在（Base）ClearCase 等工具中，签入/提交是每个文件单独的，顶多是提交留言相同。而在 Subversion、Git 等工具中，每次提交可以涉及多个文件，它们被当作一个单位来记录和处理。—译者注

// Joe 问……

### 什么是 SHA-1 哈希码？Git 为什么使用它？

在集中式版本控制系统中，一台特定的服务器会为每个提交分配一个数字（通常是不断增长的自然数），用这些数字区分不同的提交。对于分布式版本控制系统，这个方法就行不通了。

当两个人工作在同一套代码上，分别往各自的本地版本库提交时，相同的提交号对应着不同的修改。这样的提交号不再能唯一标识一个提交了。Git 使用 SHA-1 解决了这个问题。

SHA 的意思是安全哈希算法。它由美国国家安全局（NSA）开发，用来从已知的数据中产生一个短的字符串（或称消息摘要），这个字符串几乎不可能等同于由另外一组数据产生的字符串。

Git 使用版本库中的元数据产生提交名称。这些元数据包括提交者的个人信息及提交的时间戳。这些数据产生的哈希码，使不同的提交几乎不可能具有相同的提交名称。

当然，产生相同哈希码的可能性是存在的，但是这种可能性非常低，可以忽略。从数学的角度来讲，这种可能性是 2 的 63 次方分之一，也就是  $1/9\,223\,372\,036\,854\,775\,808$ 。

40 位的哈希码很难记忆。其实大多数情况下，没必要写那么长。7 位或 8 位的哈希码就可以基本保证提交名称具有唯一性。在本书中，主要使用缩写形式，也就是完整哈希码中的前 7 位。事实上，Git 自己主要也是使用缩写形式。

命令 `git log` 输出的第二行是提交者的信息；第三行是提交日期；最后是提交留言。

## 3.3 在项目中工作

Starting to Work with a Project

第一个项目的版本库已经准备好，并且已经开始跟踪文件。下面开始学习怎样处理文件修改。

这个 HTML 文件里还没有<head>和<title>元素。下面为该文件添加这些元素：

```
<html>
<head>
    <title>Hello World in Git</title>
</head>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

修改完毕，Git 可以检测到文件被修改。命令 `git status` 会显示工作目录树的状态，即当前的视图状态。Git 中的工作目录树与 Subversion 和 CVS 中的工作拷贝差不多是一个概念。<sup>3</sup>

```
prompt> git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified: index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

上面的输出结果表明 Git 监测到了修改，但还不知道如何处理它们。修改过的文件在 `Changed but not updated` 下列出来，如果要提交，须要暂存 (*stage*) 修改。

暂存修改，以准备把修改提交到版本库。Git 中有三个地方可以存放代码。第一个地方是工作目录树，编辑文件时可以直接在这里操作。

第二个是索引 (*index*)，也就是暂存区 (*staging area*)。暂存区是工作目录树和版本库之间的缓冲区。第三个，也就是最终的一个，是版本库。暂存区中存放的是准备提交到版本库中的修改，在 4.1 节“添加文件”（第 42 页）中将会详细介绍。

---

<sup>3</sup> 大致对应于 ClearCase 中静态视图 (Snapshot View) 的概念。一译者注

回头看命令 `git add`, 它可以暂存对 `index.html` 刚刚做的修改。它跟前面章节中添加一个新文件时使用的是同一个命令, 只不过, 这次它告诉 Git 要跟踪的是一个新的修改而非新的文件。

```
prompt> git add index.html
prompt> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: index.html
#
```

暂存修改过的 `index.html` 后, 执行命令 `git status` 可以看到, 输出信息中的标题从 `Changed but not updated` 变成了 `Changes to be committed`。如果打开颜色开关, `index.html` 这一行会由红色变为绿色。

使用命令 `git commit` 时, 不要忘记使用带`-m` 的参数, 并在参数后面加上提交留言, 以解释修改的原因。如下所示:

```
prompt> git commit -m "add <head> and <title> to index" \
    -m "This allows for a more semantic document."
Created commit a5dacab: add <head> and <title> to index
1 files changed, 3 insertions(+), 0 deletions(-)
```

请注意这里使用了两个`-m` 参数。Git 可以接受任意多次提交留言的输入, 每次另起一段。命令 `git log` 可以快速浏览提交留言。下面的例子以两段的形式显示了前面 `git commit` 中的内容。

```
prompt> git log -1
commit a5dacabde5a622ce8ed1d1aa1ef165c46708502d
Author: Travis Swicegood <development@domain51.com>
Date: Sun Sep 21 20:37:47 2008 -0500

add <head> and <title> to index

This allows for a more semantic document.
```

这里引入了命令 `git log` 的一个新参数`-1`。可以通过改变该数字来限制命令 `git log` 输出的提交条目的个数。

到目前为止, 我们已经学习了 Git 的基础知识, 包括添加新文件, 修改文件, 查看版本库的历史记录等。接下来让我们把注意力转到分支上来, 学习一些更深入的东西。

1// Joe 问……

### 提交留言应该包含什么内容？

在刚开始使用版本控制工具时，提交留言里应该写些什么，似乎不太好确定。提交留言至少应该体现出进行本次修改的原因。

下面是几个不错的提交留言：

- 为了增强可读性，把 if/elseif 语句改为 switch 语句。
- 删除已被证明无用的测试代码。

精心写出完美的提交留言是一种艺术。这可以通过查看其他成功项目的提交日志来学习。Git 开发项目本身的提交留言就非常好。

在每一个提交上花费一两分钟，总结这些修改，就像给坐在旁边的开发人员解释一样。首先用一句简单的话来概括该提交；然后用几句话全面解释。

另外一个须要注意的事情就是，要清楚谁是这些提交留言的读者。如果只是人类来阅读这些信息，直接写自然语言就可以了。如果是计算机程序来读取这些提交留言，就要保证提交留言包含了所需要的全部信息。有一些工具可以阅读提交留言，并更新第三方工具，比如缺陷/任务跟踪系统，以记录缺陷/任务所对应的 Git 中的提交条目。

### 让 Git 的输出变成彩色的

如果在安装和设置 Git 前已经把颜色开关打开了，那么执行命令 `git status` 时会显示彩色。未暂存的修改，即那些 Git 还不知道如何处理的修改，会显示为红色。一旦用命令 `git add` 暂存修改，文件名将会显示为绿色。

## 3.4 理解并使用分支

### Using and Understanding Branches

就像 1.7 节“用分支创建可选历史记录”（第 9 页）中介绍的那样，分支是维护项目中并行历史记录的方法。创建并行的历史记录当然非常好，但在真实的项目中该如何使用呢？

从 Git 工具的角度讲，“想怎么用就怎么用”。然而在实际应用中，须要根据具体情况来确定。有两种分支比较有用：用来支持项目的不同发布版本的分支，以及用来支持一个特定功能的开发的分支（常被称为“Topic Branch”）。在本节中，将介绍第一种分支。

`mysite` 项目的代码现在几乎可以发布了，但是还须要进行测试等工作，直到确认它达到了预期的功能和质量。而与此同时，借助分支，可以开始下一个版本的新功能的开发了。

分支可以为要发布的代码保留一份拷贝，所以无须停止正在开发的工作。创建分支的命令是 `git branch`，该命令需要两个参数：新分支名称和父分支名称。新创建的分支基于已经存在的父分支。

```
prompt> git branch RB_1.0 master
```

该命令从主分支（`master branch`）上创建一个叫 `RB_1.0` 的分支。主分支是 Git 的默认分支。CVS 和 Subversion 的用户可能已经认出，这对应于 CVS 和 Subversion 中的主干（`trunk`）。<sup>4</sup>

分支名称中的 `RB` 代表发布分支（*release branch*）。该前缀可以让人快速分辨出哪些分支是发布分支。

现在来做一些新的改动。这些改动不影响准备发布的代码。下面给“bio”页面添加一个链接。在`</body>`之前增加如下代码：

```
<ul>
  <li><a href="bio.html">Biography</a></li>
</ul>
```

用如下命令提交这些修改：<sup>5</sup>

```
prompt> git commit -a
... editor launch, create log message, save, and exit ...
Created commit e993d25: add in a bio link
1 files changed, 3 insertions(+), 0 deletions(-)
```

<sup>4</sup> ClearCase 的用户可能已经认出，它对应于 ClearCase 中的 Main Branch。一译者注

<sup>5</sup> 最好加上`-m` 参数及提交留言，不然可能不成功。等学完第 4 章“添加与提交：Git 基础”（第 41 页）你就明白啦。一译者注

请注意，命令 `git commit` 带了一个-a 参数，它告诉 Git 提交全部修改过的文件。

现在主分支上有最新的修改，而发布分支上还是原来的代码。请切换到发布分支，做发布前的最后修改。切换分支的命令是 `git checkout`。

```
prompt> git checkout RB_1.0
Switched to branch "RB_1.0"
```

如果 `index.html` 文件已经打开，你使用的编辑器可能会提醒文件已经被修改。重新载入文件，刚才在主分支上做过的修改就消失了，当然，它们还是很安全地存放在另一条分支上。

在发布前做最后的修改：在`<head>`标记块中添加一些描述性的元标签。代码如下所示：

```
<head>
  <title>Hello World in Git</title>
  <meta name="description" content="hello world in Git" />
</head>
```

保存并提交该修改：

```
prompt> git commit -a
... editor launch, create log message, save, and exit ...
Created commit 4b53779: Add in a description element to the metadata
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在是发布它的时候了。为此，要给这个版本打个标签。

## 3.5 处理发布

Handling Releases

现在准备发布 1.0 版本了，要给它打个标签。给 Git 中的代码打标签，意味着在版本库的历史中（用人类可以理解的名称而非 SHA 号）标记出特定的点，这样将来就容易找到相应版本的代码。因为是要做 1.0 版的发布，所以打一个名为“1.0”的标签：

```
prompt> git tag 1.0 RB_1.0
```

以上命令中的两个参数分别指明了标签的名称和希望打标签的点，它们分别为 1.0 和 `RB_1.0` 分支的末梢（所对应的版本，或者说所对应的提交）。用不带参数的命令 `git tag` 可以查看版本库中的标签列表。目前只显示了刚刚打过的标签 1.0：

```
prompt> git tag
1.0
```

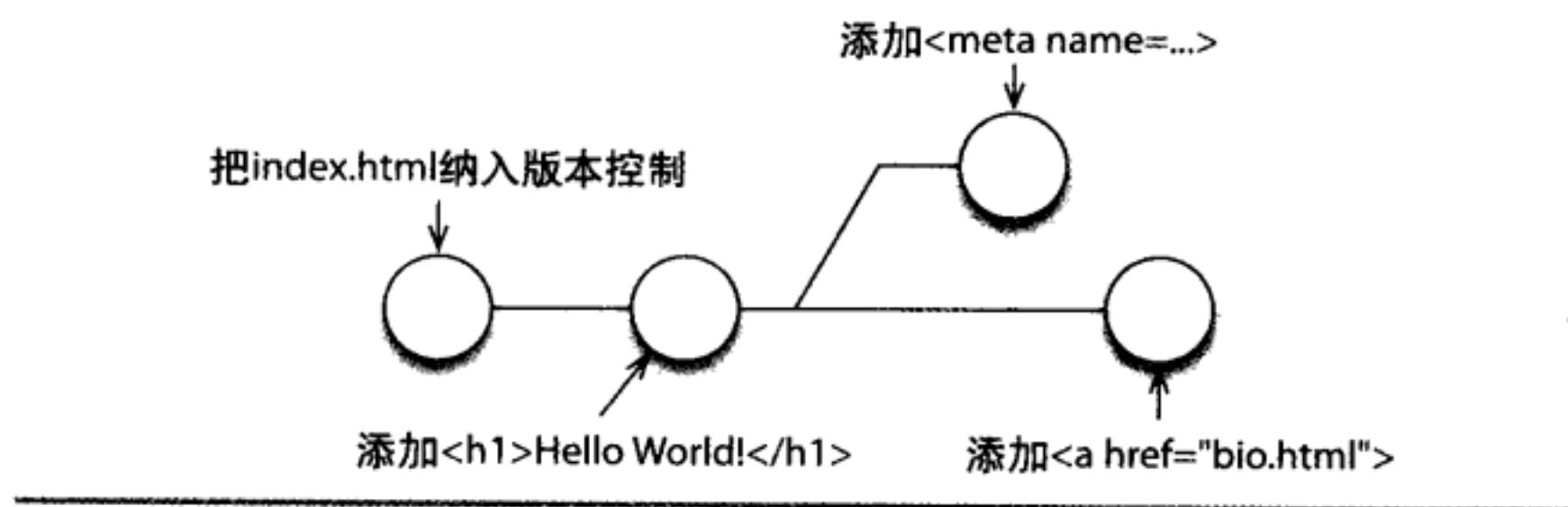


图 3.1 变基前的 mysite 版本库

给代码打过标签以后，须要做一些整理工作。现有的两条分支上有不同的提交，它们并不知道彼此所包含的代码。创建 RB\_1.0 以后，主分支用于版本 2.0 的新功能开发，现在要把 RB\_1.0 分支上所做的修改合并到主分支上来。

变基命令 `git rebase` 可以完成这项工作。变基是把一条分支上的修改在另一条分支的末梢重现。<sup>6</sup>现在的版本树如图 3.1 所示，变基以后的版本树如图 3.2 所示。

首先使用前面讲过的命令 `git checkout` 切回到主分支。

```
prompt> git checkout master
Switched to branch "master"
```

接着运行命令 `git rebase`，后面跟一个参数：希望变基到哪条分支的末梢，就使用哪条分支名称做参数。

```
prompt> git rebase RB_1.0
First, rewinding head to replay your work on top of it...
Applying: add in a bio link
```

现在版本库如下页的图 3.2 所示。`add in a bio link` 结点放入 RB\_1.0 分支最后一个提交的后面。

最后，作为整理工作的一部分，删除发布分支 RB\_1.0。这看上去好像很危险，不过不用担心。刚才打过的标签所指向的结点就是 RB\_1.0 分支的末梢。（只要标签还在，从标签到版本树起点的一连串提交记录就都在。）这时删除该分支只是删除了分支的名字，并不会删除分支上的任何实际内容。

<sup>6</sup> 变基的意思是“改变分支的基底”。不同的版本控制工具，其实现方法不尽相同。假定有分支 A 和分支 B，它们的分叉点是版本 Y。Git 的实现方法是，如果站在分支 B 上告诉 Git “我要变基到 A 的末梢”，那么 Git 会把从版本 Y 到分支 B 的当前末梢之间的所有提交，顺序加到分支 A 的末梢上去，生成新的分支 B 及其末梢。而分支 A 及其末梢则没有任何变化。—译者注

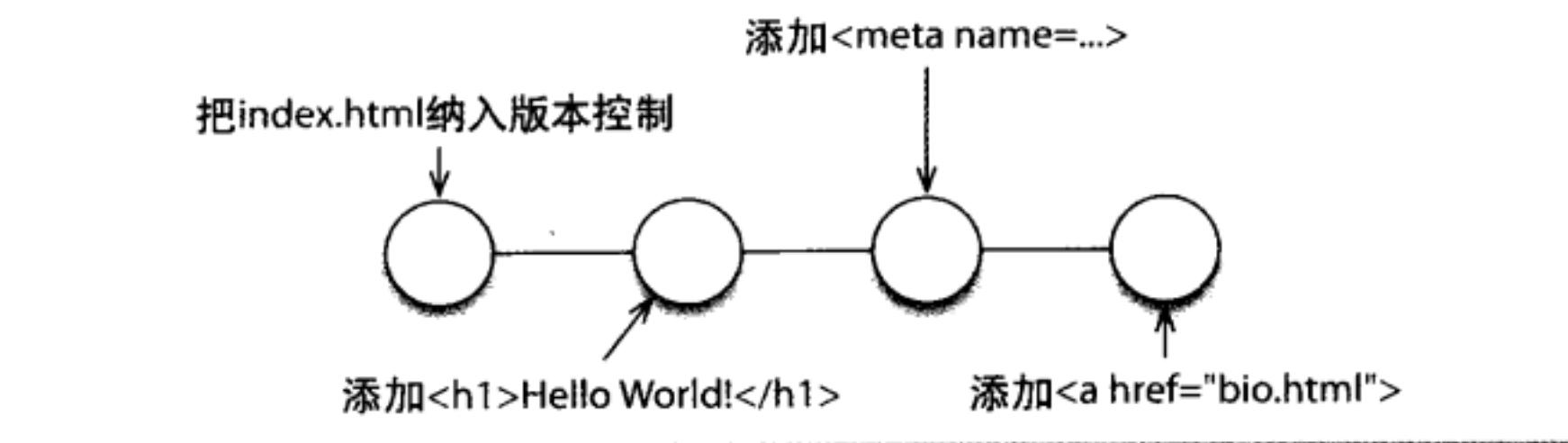


图 3.2 变基后的 mysite 版本库

使用命令 `git branch -d` 可以删除分支：

```
prompt> git branch -d RB_1.0
Deleted branch RB_1.0.
```

如果没有发布分支，将来如何给“1.0.x”分支打补丁呢？很简单，只须从打标签的地方再创建一条分支即可。

还记得前面是如何创建分支的吗？当时，命令的最后一个参数是新分支的父分支名称。现在只须把父分支名称改成发布标签名称即可。命令如下：

```
prompt> git branch RB_1.0.1 1.0
prompt> git checkout RB_1.0.1
Switched to branch "RB_1.0.1"
```

运行命令 `git log` 快速查看历史记录，可以看到在 RB\_1.0 分支上只有 3 个提交（而不包含主分支上的最新提交）。

```
prompt> git log --pretty=oneline
4b53779 Add in a description element to the metadata
a5dacab add <head> and <title> to index
7b1558c add in hello world HTML
```

使用 Git 做的最后一件事是，为代码发布创建归档文件。没有必要总是把历史记录（也就是 Git 版本库）一起发布，通常情况下，将标签对应的版本内容打包成

一个 tar 包或 zip 包就足够了。Git 提供了 `git archive` 命令来做归档处理。

下面就如何为 `mysite` 创建一个 gzip 文件：

```
prompt> git archive --format=tar \
    --prefix=mysite-1.0/ 1.0 \
    | gzip > mysite-1.0.tar.gz
```

该命令中有 3 个参数。第一个，`--format` 指明要产生 tar 格式的输出；第二个，`--prefix` 指明包中所有东西都放到 `mysite-1.0/` 目录下。最后，`1.0` 指明须要归档的标签名称。

如果不习惯 Unix 风格的命令，第三行看起来有一点复杂。该命令把 `git archive` 产生的 tar 文件用管道输出的方法传递给命令 `gzip` 进行压缩，而压缩结果则重定向到 `mysite-1.0.tar.gz` 压缩包里。

命令 `git archive` 也可以产生 zip 文件。创建 zip 文件更简单，因为 `git archive` 的输出是已经按 zip 标准压缩过的。下面这条命令可以生成一个 zip 文件。

```
prompt> git archive --format=zip \
    --prefix=mysite-1.0/ 1.0 \
    > mysite-1.0.zip
```

生成 zip 格式和 tar 格式的命令参数几乎一样，只是改变了传递给`--format` 的参数，并且无须通过命令 `gzip` 管道输出，直接把归档内容保存到归档文件中。

当然，发布一个版本还有更多任务要完成，例如更新 wiki、代码验收测试等，这些任务与项目开发过程中所使用的具体工具和策略相关。

通过打标签，可以使 Git 与现有开发流程交互协作，Git 的归档命令则有助于创建项目源代码的发布包。

以上都是一些基本的命令，但到目前为止它们都是与本地版本库打交道的命令。下面将介绍克隆（Clone）版本库的命令，它可以从其他程序员那里远程获得项目的源代码及其历史记录。

## 3.6 克隆远程版本库

Cloning Remote Repositories

到目前为止，我们已经讲述了不少与本地版本库 `mysite` 打交道的命令，但还没有涉及远程版本库。除了前面的功能之外，Git 也可以与远程版本库打交道，以分享本地的工作成果，或者复制其他的版本库到本地。

与远程版本库打交道，首先要用命令 `git clone` 克隆远程版本库。克隆版本库就像它听上去的那样，会在本地创建远程版本库的完整拷贝。

在 GitHub（一个提供 Git 版本库托管服务的网站）上共享本书中 `mysite` 项目的全部拷贝，可以使用命令 `git clone` 来把它们克隆到本地：

```
prompt> cd /work
prompt> git clone git://github.com/twicelgood/mysite.git mysite-remote
Initialized empty Git repository in /work/mysite-remote/.git/
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 12 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (2/2), done.
```

命令 `git clone` 带有两个参数：远程版本库的位置和存放该版本库的本地目录。第二个参数是可选的，但是在本书的例子中已经有了 `mysite` 目录，所以须要提供第二个参数指定远程版本库拷贝的目标目录。

本书 7.2 节“克隆远程版本库”（第 94 页）还会介绍更多的参数，但大多数情况下这两个参数已足够了。

当然，Git 也可以把本地改动推入到远程版本库，以及从远程版本库取来改动。这里暂时先不介绍这些内容了。在第 7 章“与远程版本库协作”（第 91 页）中将会详细介绍远程版本库。

本章介绍了 Git 的基础内容：命令 `git add`、`git commit`、`git status` 和 `git log`。在此基础上进一步介绍了命令 `git branch`、`git tag`、`git rebase` 和 `git clone`。

本书第一篇快速浏览了 VCS/DVCS 的概念和 Git 本身。本章重点放在了 Git 的基本命令上，而尚未详细解释这些命令背后的原理。在本章的实战中，我们使用 Git 管理了一个简单的网页项目。

随后的章节，将从 Git 基础知识和基本原理开始，详细介绍 Git——这仍将基于本章中创建的“mysite”版本库。



## 第2篇 Git 日常用法

---

Part II Everyday Git





## 第 4 章

### 添加与提交：Git 基础

Adding and Committing: Git Basics

本书第一篇对 Git 进行了概要介绍；第二篇将研究细节：深入学习 Git，并练习 Git 常用命令。

本章深入解析第 3 章“创建第一个项目”（第 25 页）中的内容。通过本章的学习，可以深入了解：

- 添加文件
- 提交修改
- 察看历史
- 管理文件

本章将基于第 3 章“创建第一个项目”（第 25 页）中创建的版本库来继续演示。如果你跳过了前面章节直接进入本章，也不必担心，可以克隆这个版本库：<sup>1</sup>

```
prompt> git clone git://github.com/twicegood/mysite.git
Initialized empty Git repository in /work/mysite/.git/
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 12 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (2/2), done.
```

不管你在学习前面章节的过程中一步步建好了版本库，还是刚从 GitHub 上克隆了一个现成的版本库，现在让我们开始本章的学习。

<sup>1</sup> 你公司的防火墙可能会阻止你克隆这个版本库。—译者注

## 4.1 添加文件到暂存区

Adding Files

添加新文件和修改版本库中的已有文件的内容是常有的事，命令 `git add` 可以完成这两种操作。只须提供要添加或修改一个或多个文件的文件名，Git 就会暂存须要提交的变更。

在继续学习新东西之前，让我们先来弄清一些术语。**暂存的变更** (*stage change*) 就是工作目录树中那些你打算提交到版本库的变更。暂存操作将会更新 Git 的内部索引 (*index*)。大家常把该索引称为**暂存区** (*staging area*)。

通过暂存区，可以设置哪些变更要提交到版本库，哪些先不提交。

“但是这样不就把事情弄复杂了吗？为什么不能直接提交？”

是的，在很多情况下，暂存区没啥好处反而把事情弄复杂了。稍后我们会介绍如何走捷径来简化操作。但是不要忘了暂存区这个概念，它给用户在提交版本库到之前精心选择提交内容的机会。明确指定，哪些变更要提交，哪些不用。这是通过命令 `git add` 来实现的。

正如大多数 Git 命令一样，除了简单调用命令 `git add` 之外，还可以通过指定不同的命令参数来改变添加改动的方式，其中的一些格外有用。

Git 交互添加方式可以用来选择要暂存的文件或文件的部分内容。使用 `-i` 选项将启动该交互方式。

给命令 `git add` 添加 `-i` 选项会启动交互命令提示符，在这种方式下可以交互暂存新文件，暂存对已有文件的修改，甚至只暂存部分修改。接下来修改 `index.html` 文件以便演示。如下所示把 `Biography` 链接改为 `About`：

```
<1i><a href="about.html">About</a></1i>
```

现在请运行命令 `git add -i`，启动交互方式，Git 给出如下提示：

```
prompt> git add -i
      staged  unstaged   path
 1:   unchanged      +1/-1 index.html

*** Commands ***
 1: status   2: update   3: revert   4: add untracked
 5: patch    6: diff     7: quit     8: help
What now>
```

现在有几个选项可以选择。输入 1 显示与该方式启动时相同的输出。如果想要添加文件到暂存区，可以输入 2：

```
What now> 2
      staged      unstaged path
  1:  unchanged      +1/-1 index.html
Update>>
```

这会显示一个可暂存文件列表。在本例中，只有一个文件须要添加，所以输入 1。该文件名称前面出现一个星号(\*)标识，表示该文件将暂存。

```
Update>> 1
      staged      unstaged path
* 1:  unchanged      +1/-1 index.html
Update>>
```

如果想退出该模式，请按下回车键即可退回交互方式的主菜单：

```
Update>>
updated one path

*** Commands ***
1: status    2: update    3: revert    4: add untracked
5: patch    6: diff      7: quit      8: help
What now>
```

这时再次察看工作目录树的情况，可以看到有一个已暂存的修改，而没有未暂存的修改了：

```
What now> 1
      staged      unstaged path
  1:      +1/-1      nothing index.html

*** Commands ***
1: status    2: update    3: revert    4: add untracked
5: patch    6: diff      7: quit      8: help
What now>
```

如果想要取消已暂存的修改，可以使用 revert 模式。revert 模式的用法跟前面介绍的“update”模式用法相似：

```
What now> 3
      staged      unstaged path
  1:      +1/-1      nothing index.html
Revert>> 1
      staged      unstaged path
* 1:      +1/-1      nothing index.html
Revert>>
reverted one path
```

```
*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
What now>
```

选择 `status`，可以看到之前的修改已经变成了未暂存状态：

```
What now> 1
      staged      unstaged   path
1:    unchanged           +1/-1 index.html
```

```
*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
```

如果想要暂存还没有被 Git 跟踪的文件，可以使用第 4 个模式。它的用法跟其他的模式类似，在此就不再演示了。

“patch”模式是交互方式中最有用的模式，进入该模式的方法跟其他模式类似。进入到“patch”模式后，可以选择单个或多个文件。选择后，Git 会显示这些文件的当前内容与版本库中的差异，然后你可以据此决定是否添加这些修改到暂存区。运行情况如下：

```
What now> 5
      staged      unstaged   path
1:    unchanged           +1/-1 index.html
Patch update>> 1
      staged      unstaged   path
* 1:    unchanged           +1/-1 index.html
Patch update>>
diff --git a/index.html b/index.html
index e812d0a..ca86894 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,7 @@
<body>
  <h1>Hello World!</h1>
  <ul>
-   <li><a href="bio.html">Biography</a></li>
+   <li><a href="about.html">About</a></li>
  </ul>
</body>
</html>
Stage this hunk [y/n/a/d/e/?]?
```

可以根据需要决定是否添加该文本块（*hunk*）。文本块由文件中连续的修改构成，文件中不同的区域作为不同的文本块。

根据提示，输入 `y` 表示接受修改，输入 `n` 表示忽略修改，输入“`a`”或“`d`”分别表示添加或放弃剩余的修改。不用担心该如何记住所有这些选项，输入“`?`”将会显示所有选项的帮助信息。

现在输入 `n` 以放弃暂存该文件块，然后输入选项 `7` 退出交互方式：

```
Stage this hunk [y/n/a/d/e/?]? n
```

```
*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
What now? 7
Bye.
```

作为 Git 中所有好用的功能之一，命令 `git add` 有直接进入补丁模式的快捷方式，而无须先启动交互模式。在命令 `git add` 后添加`-p` 参数就可直接进入补丁模式。

例如，可以运行下面的命令直接启动补丁模式：

```
prompt> git add -p
diff --git a/index.html b/index.html
index e812d0a..ca86894 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,7 @@
<body>
    <h1>Hello World!</h1>
    <ul>
-       <li><a href="bio.html">Biography</a></li>
+       <li><a href="about.html">About</a></li>
    </ul>
</body>
</html>
Stage this hunk [y/n/a/d/e/?]?
```

这次输入 `y` 选择文件块，然后该文件就处于暂存状态并准备提交。提交是下一节要讲的内容。

## 4.2 提交修改

Committing Changes

提交是一个相对简单的过程，它将变更添加到版本库的历史记录中，并为它们分配一个提交名称。

## 跟踪空目录

Git 不会单独记录和跟踪目录。曾经很多人反对 Git 跟踪空目录，他们认为如果项目管理得好，通常并不需要空目录。

或许 Git 的未来版本会跟踪目录。在此之前，有一个临时解决方法，就是在想要跟踪的目录里添加一个空文件，并把该文件添加到版本控制中。

这个空文件的文件名可以是任意的名称，但是大家常用于以句点“.”开头的文件名。这是因为大多数文件系统在显示目录内容时，会忽略以句点开头的文件。

注意，这里所说的提交并不是把修改发送到某个中央版本库中，而是提交到本地版本库。但其他人可以从你的本地版本库中拖入修改，或者你从本地版本库推入某个其他的版本库。注意 Git 不支持自动更新版本库。这些知识将在 7.3 节“版本库同步”（第 95 页），以及 7.4 节“推入改动”（第 96 页）中详细介绍。

有多种使用命令 `git commit` 的方式，这些方式都须要输入一段提交留言。对于简短的留言，可以使用参数`-m`，在参数之后输入。留言内容可以是任意有效字符串。如果留言包含多段内容，可以给命令 `git commit` 传递多个`-m`。

对于比较复杂的提交留言，须要用编辑器来输入。如果输入不带`-m` 参数的 `git commit` 命令，Git 将启动编辑器来编辑提交留言。为启动编辑器，Git 会按照以下顺序查找编辑器的设置：

1. 环境变量 `GIT_EDITOR` 的值。
2. Git 的设置 `core.editor` 的值。<sup>2</sup>
3. 环境变量 `VISUAL` 的值。
4. 环境变量 `EDITOR` 的值。
5. 如果上述值均为空，Git 会尝试启动 `vi` 编辑器。

使用编辑器创建提交留言这种方式时，可以添加选项`-v` 把要提交的内容与版本

<sup>2</sup> 比如说，Windows 环境下 MSysGit 的用户可以这么设置：“`git config --global core.editor notepad.exe`”。美中不足的是，如果 Git 把一些内容传到编辑器上（比如正文下一段要讲的），那么这些内容是用 Unix 风格换行符分行，于是在 `notepad` 上，全都挤到一行上去了。如果改用 AkelPad (<http://akelpad.sourceforge.net>)，就没有这个问题了。一译者注

库中版本的比较结果添加到编辑器中。编辑器中显示的信息有很多以符号“#”开始的行，Git 提取提交留言时会忽略这些行。

和 Git 中的几乎所有命令一样，提交操作也可以使用多种不同的方法完成。

在进行新的提交之前，让我们先学习三种提交方法。

第一种方法，先对要提交的文件或修改调用命令 `git add` 来添加到暂存区，例如可以对要提交的修改使用命令 `git add -p`，然后再调用 `git commit` 命令完成提交。该过程如下所示：

```
prompt> git add some-file  
prompt> git commit -m "changes to some-file"
```

上面例子中的提交留言不是一个好的例子，但由于它比较适合本书排版，因此把它作为例子放在这里。在实际提交留言中，不能仅仅提供表面的描述，还应该说明修改的原因。还记得本书第 31 页中“Joe 问……”部分的内容吗？提交留言应详细说明提交的原因，就像与一个开发人员面对面交流一样。

另一种提交的方法是给命令 `git commit` 传递-a 参数，Git 会把工作目录树中当前所有的修改提交到版本库中。注意在这种情况下，命令 `git commit` 只会把已纳入 Git 版本控制的文件添加到版本库中，而不会添加尚未被跟踪的文件。

如果工作目录中只有前面例子对 `some-file` 所作的修改，可以执行下面的命令把 `some-file` 加入暂存区并提交：

```
prompt> git commit -m "changes to some-file" -a
```

最后一种提交修改的方法是指定要提交的文件或文件列表，具体做法是，把要提交的文件列在其他参数的后面。

正如参数-a 一样，它将提取工作目录树中最新的版本，但是它只会提取指定的文件或文件列表。下面举例说明前面对 `some-file` 修改的提交：

```
prompt> git commit -m "changes to some-file" some-file
```

这三个例子都有各自的用途。当要用命令 `git add -p` 提交文件的一部分修改时，用先暂存后提交的方法比较适合；如果修改了多个文件且须要提交其中的一个文件时，可以用指定提交文件的方法来处理。

### 使用与 SVN 简称类似的 Git 别名

Subversion 的用户或许已经适应了常用命令的简写方式：无须输入 `svn checkout` 或 `svn commit`，使用简单的 `svn co` 或 `svn ci` 就可以完成同样的功能。

当你参照本书例子练习时，如果在 Git 中尝试输入一些简写命令，Git 会报错并提示那不是 Git 命令。Git 并没有像 Subversion 那样提供命令的简写方式，但是它提供了一个更好的方法，可以通过 `git config` 添加用户特有的命令别名。

例如，可以将命令 `git commit` 简写为 `git ci`：

```
prompt> git config --global alias.ci "commit"
```

这种给命令起别名的方式适用于任何 Git 命令，所以用户可以按照自己喜欢的方式定制工作环境，只须使用简写别名替换 `alias.` 后面部分就可以了。

对于这三种提交方法：提交暂存后的修改、提交工作目录树中的所有修改，以及提交工作目录树中指定的修改，要牢记一个重要区别，后两种方法把修改直接提交，而第一种方法是先暂存后提交。

也就是说，当在工作目录树中修改了一个文件后，可以先暂存该修改，之后再次修改该文件，然后可以提交刚才已经暂存的修改，而再次修改的内容没有提交，仍然仅在工作目录树中。

可以把暂存区想成一个缓冲区，用命令 `git add` 向缓冲区中添加修改，直到执行 `git commit` 提交缓冲区中的修改。

现在先别忙着提交，让当前在暂存区中的内容多保留一会儿，因为我们要留着它以方便演示下一节中的内容：使用多种方法察看修改情况。

## 4.3 查看修改内容

Seeing What Has Changed

如果添加新的文件或修改文件等工作刚完成，通常应该还记得动了哪些文件。但有时就没有那么幸运了。比如在暂存了修改之后，有人过来拜访，或者要去处理一些紧急情况，回来后恐怕就不那么容易想起离开时的情况了。

使用 Git 的命令 `git status` 和 `git diff`, 可以找出工作目录树中做了哪些修改, 以及是如何修改的。

## 查看当前状态

可以使用命令 `git status` 来查看工作目录树中所有的变动。该命令的输出结果是暂存区内要提交的内容, 工作目录树中未纳入暂存区的改动, 以及尚未纳入 Git 版本控制的新文件。

现在来看看前面已经暂存的修改:

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

注意输出信息中的 `Changes to be committed` (待提交变更) 部分, 列出了哪些文件等待提交。接下来让我们修改另外一处, 在“About”链接后添加一条“Contact”链接:

```
<li><a href="contact.html">Contact</a></li>
```

完成后请保存修改, 然后再执行命令 `git status`:

```
prompt> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified: index.html
#
```

输出结果把 `index.html` 文件显示了两遍。`Changes to be committed` (待提交变更) 部分中的那个对应之前已经暂存的修改; `Changed but not updated` (未更新到索引的变更) 部分中的那个对应还未暂存的修改; 如果设置 Git 时已经把 `color.ui` 开关打开, 那么第一个修改将会是绿色的, 而第二个则会是红色的。关于如何配置 `color.ui`, 请查阅第 2.2 节“设置 Git”(第 20 页)。

这是刚刚才做完的修改，当然还记得改过什么。但等再过些时候，我们就得借助 `git diff` 命令来察看文件的改动了。

## 查看文件改动

使用命令 `git diff`，Git 可以显示出工作目录树、暂存区及版本库之间的差异。<sup>3</sup>

直接调用不带参数的 `git diff`，将显示工作目录树中未被暂存（当然还没有提交）的改动：

```
prompt> git diff
diff --git a/index.html b/index.html
index ca86894..5fdc539 100644
--- a/index.html
+++ b/index.html
@@ -7,6 +7,7 @@
<h1>Hello World!</h1>
<ul>
    <li><a href="about.html">About</a></li>
+    <li><a href="contact.html">Contact</a></li>
</ul>
</body>
</html>
```

从这个例子中可以看到，“Contact”链接所在行行首的“+”号表示该行是新增加行。但是“About”链接行没有显示任何改动。

不添加任何参数而运行 `git diff`，比较的是工作目录树与暂存区之间的区别。前面例子中已经暂存过几个修改，可以使用命令 `git status` 查看已经暂存的东西。如果想比较暂存区和版本库中的区别，可在命令 `git diff` 后添加参数`--cached`：

```
prompt> git diff --cached
diff --git a/index.html b/index.html
index e812d0a..ca86894 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,7 @@
<body>
    <h1>Hello World!</h1>
    <ul>
-        <li><a href="bio.html">Biography</a></li>
+        <li><a href="about.html">About</a></li>
    </ul>
</body>
</html>
```

<sup>3</sup> 在做 `git diff` 时，版本库里的版本、工作目录树、暂存区三者，都被当成版本“断面”来相互比较。比如，在提交了暂存的全部改动后，不是暂存区里没有内容了，而是暂存区与版本库中最新版本之间的差异完全没有了。一译者注

这个例子显示的是在 4.1 节“添加文件到暂存区”（第 42 页）中所做的修改。请注意，在“Biography”所在行的行首有个符号“-”，表示该行被删除了。如果颜色开关是打开的，Git 会将删除的内容标记为红色，添加的内容则为绿色。

然而，这个比较命令并不会显示出没有暂存的修改。在命令 `git diff` 后添加参数 `HEAD`，可以比较工作目录树（包括暂存和未暂存的修改）与版本库中的差别：

```
prompt> git diff HEAD
diff --git a/index.html b/index.html
index e812d0a..5fdc539 100644
--- a/index.html
+++ b/index.html
@@ -6,7 +6,8 @@
<body>
  <h1>Hello World!</h1>
  <ul>
-   <li><a href="bio.html">Biography</a></li>
+   <li><a href="about.html">About</a></li>
+   <li><a href="contact.html">Contact</a></li>
  </ul>
</body>
</html>
```

`HEAD` 关键字指的是当前所在分支末梢的最新提交（也就是版本库中该分支上的最新版本）。如果对分支这一概念还不熟悉，不必担心，后面章节会有详细介绍。

接下来执行提交操作，以便 Git 记录和跟踪变更：

```
prompt> git commit -a -m "Change biography link and add contact link" \
-m "About is shorter, so easier to process" \
-m "We need to provide contact info"
Created commit 6f1bf6f: Change biography link and add contact link
1 files changed, 2 insertions(+), 1 deletions(-)
```

现在我们已经学会了所有常用命令。用这些命令可以添加文件到暂存区，提交变更到版本库，还可以比较所做的修改与版本库中的区别。接下来要学习一些环境整理命令，诸如文件移动、拷贝，以及如何忽略某些文件。

## 4.4 管理文件

Managing Files

目前版本库里非常简单，只有一个文件，但是随着岁月的流逝，文件的整理就很有必要了。

有时需要移动文件，复制代码，忽略垃圾文件等。

## 文件重命名与移动

我们可以一次写成完美无缺陷的代码。真是这样吗？尽管自以为做到了完美，有时还是会出错。例如，给文件起了错误的名字，或者文件放到了错误的目录下。

在 Git 中，可以通过命令 `git mv <原文件名称> <新文件名称>` 来移动文件。该命令告诉 Git 使用原文件的内容来创建新文件，新文件保留原文件的历史修改记录，并删除原文件。

假定前面例子中的“Hello world”网页并不是一个起始网页（`index.html`），下面移动该文件：

```
prompt> git mv index.html hello.html
prompt> git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   renamed:    index.html -> hello.html
#
```

命令 `git mv` 很方便。如果不这么做，Git 也可以监测到文件移动，但是这需要引入更多的步骤：首先必须移动文件，然后调用命令 `git add` 添加新的文件，最后调用命令 `git rm` 把旧的文件从版本库中移除。

现在提交重命名后的文件 `hello.html`：

```
prompt> git commit -m "rename to more appropriate name"
Created commit 9a23464: rename to more appropriate name
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename index.html => hello.html (100%)
```

文件 `index.html` 已经被重命名。接下来介绍文件复制。

## 复制文件——真的要复制它么

已经习惯 Subversion 的用户可能会发现 Git 比较奇怪：它不提供文件复制用的 `git cp` 这样的命令。Git 不提供此命令的原因在于，它不需要这样的复制命令。

Subversion 的复制功能主要是让用户创建分支和标签（有时也用于记录和跟踪复制的代码）。而 Git 提供了创建分支和标签的命令。当然，Git 也能记录和跟踪复制的代码。

Git 跟踪复制的代码让我们想起之前没有进行深入讨论的一点：Git 并不跟踪管理文件，而是跟踪文件内容。见 1.5 节“跟踪项目、目录和文件”（第 7 页）。

Git 这样做是有理由的：文件名只是附在文件内容上的元数据，以便比较容易地组织文件目录树。Git 可以询问文件系统以获得文件名，但 Git 真正关心的是文件本身的内容。

既然 Git 仅跟踪文件内容，那么它是如何管理文件复制的呢？实际上，无须额外告诉 Git 存在文件复制，Git 自己就可以监测到它。

6.5 节“跟踪内容”（第 79 页）将介绍如何查看复制的代码。但是须要注意的是，如果发现自己在复制大量代码，可能意味着软件架构存在问题，应予以重构<sup>4</sup>，以避免代码重复。

## 忽略文件

所使用的编辑器可能造成你运行 `git status` 时看到的输出结果与本书中的稍有不同。这是因为 I 已改变了自己的版本库的一些设置，但还没有告诉你。

如果我没有做这些设置，那么命令 `git status` 后的输出结果就变成：

```
prompt> git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .index.html.swp
nothing added to commit but untracked files present
(use "git add" to track)
```

我所用的编辑器是 MacVim<sup>5</sup>。用它编辑一个文件时，就会产生一个以 `swp` 为后缀的临时文件。这个文件不应该加到版本库中。因此，应把这个文件名加入版本库的 `.gitignore` 文件中，这样，命令 `git status` 输出中就能显示它了。

<sup>4</sup> 重构是一种通过不断少量调整代码而持续优化代码结构的艺术。关于重构的全面介绍，可以参考 Martin Fowler 的《Refactoring》[FBB+99]一书。

<sup>5</sup> <http://code.google.com/p/macvim/>

把每一个要忽略的文件名分别加到`.gitignore` 文件中，实在是没有效率。Git 支持通配符。MacVim 交换文件总以句点开始并以`.swp` 结尾，所以我将字符串`.*.swp` 添加到`.gitignore` 文件中。<sup>6</sup>这样，Git 就会忽略所有符合该通配符的文件：

```
...把“.*.swp”添加到.gitignore文件中...
prompt> git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
nothing added to commit but untracked files present
(use "git add" to track)
```

现在临时文件不见了，取而代之的是一个新的`.gitignore` 文件。在你提交它之后，Git 把它当作一个普通文件记录和跟踪，并且可以随本地版本库里的其他内容一起传播。于是，任何克隆该版本库的人也同样可以忽略那些临时文件。

然而，每个开发人员都有自己喜爱的编辑器，每个编辑器都会用不同的机制来保存备份和标识已打开的文件，所以把个人偏好作为版本库的一部分发布出去并不是一个很好的想法。

如果把要忽略的文件改添到`.git/info/exclude` 文件中，则本地的版本库会忽略这类文件，同时又不会把此设置传播出去。

## 确定设置忽略的方式

是通过`.gitignore` 来设置版本库级别的忽略呢，还是通过编辑`.git/info/exclude` 来设置本地级别的忽略呢？为此，问自己一个简单的问题：每个人的版本库中都需要这类忽略吗？

如果答案是肯定的，就在`.gitignore` 中添加规则来忽略该类文件，并把`.gitignore` 提交到版本库中。如果仅仅是本人需要的忽略，就将它添加到`.git/info/exclude` 文件中。

我不想把对`swp` 文件的忽略添加到`.gitignore` 中，于是在`.git/info/exclude` 文件中添加了一行`.*.swp`。

本章介绍了 Git 的基本操作，其他内容皆基于此：添加（*add*）到暂存区并提交（*commit*）。

下一章将讲述分支。很多版本管理工具把分支当作非核心的、额外的功能。然而在 Git 中，分支如此重要，所以本书尽可能早地介绍它。

---

<sup>6</sup> 注意，要忽略以“`.bak`”为后缀的文件，要用`*.bak`，而不是`.*.bak`，除非你只想忽略那些以点号开头的`bak` 文件。—译者注

# 第 5 章

## 理解和使用分支

Understanding and Using Branches

只使用 Git 中的一条分支，即默认的主分支，就可以获得版本控制的好处：当与其他开发人员合作时，Git 会跟踪历史记录，以避免因偶然因素而删除重要的东西。

通过这种方式使用 Git，就好比买了辆赛车，却用在上下班路上。虽然可以驾驶它到达目的地，但显然并没有发挥它的全部潜力。

分支是 Git 的核心内容之一。本章将学习分支的概念和分支的如下操作：

- 创建新分支。
- 合并分支。
- 处理合并冲突。
- 删除分支。
- 重命名分支。
- 使用发布分支。

本章将继续使用前面创建的版本库。如果你跳过了前面章节直接进入本章，可以从 GitHub 上克隆这个版本库：

```
prompt> git clone git://github.com/twicegood/mysite-chp4.git
Initialized empty Git repository in /work/mysite-chp4/.git/
```

## 5.1 什么叫分支

译者注：本节标题为“什么是分支”，原文为“Why Are Branches Useful?”

实际工作中，我们要为许多任务设定优先级。添加新功能，重构代码以方便维护，或者修正偶然出现的 Bug 等，所有这一切都会占用开发人员的时间和精力，因此要为这些任务的先后顺序作出合理的安排。（甚至有时候，暂时中断正在进行的任务来处理紧急情况。）

如果在 Git 中只使用一条版本演进的轨迹，就难以支持上述情形。因此须要创建和使用多条分支。在 1.7 节“使用分支来跟踪并行演进”（第 9 页）中介绍过，多条分支可以用来记录不同的版本演进轨迹。

事实上，在 Git 里，任何修改和提交都是在分支上完成的。到目前为止，前面所有的操作都是在主分支上进行的。我们可以把这条分支改个名称：

```
prompt> git branch -m master mymaster
prompt> git branch
* mymaster
```

这样，就把主分支改名为 `mymaster` 分支。虽然这两个命令都使用 `git branch` 命令，但使用的参数不一样。

第一个命令有三个参数。`-m` 参数告诉 Git 要执行分支移动（重命名）操作。另外两个参数分别是分支原名称和新名称。

第二个命令 `git branch` 没有参数，因此 Git 就显示本地版本库中所有的本地分支名称。因为目前版本库中仅有一条分支，所以 Git 只显示这条分支名称。

现在把它的名字改回 `master`，以遵惯例：

```
prompt> git branch -m mymaster master
```

在 Git 里，我们总是在某条分支上工作。使用分支很容易，但有些版本控制系统在创建分支时会将所有的文件复制到新目录中。Git 可不这么做，它只把分支创建后的修改记录在这条分支上。

这么说其实并不完全准确。实际上，Git 的分支只记录和跟踪该分支末梢的那个提交。因为沿这个版本回溯，可以找到该分支完整的历史轨迹。<sup>1</sup>

---

<sup>1</sup> 有数学或计算科学背景的读者会意识到，这是有向无环图（Directed Acyclic Graph，DAG）。

使用分支时，最难确定的是何时创建分支，这是一门艺术。至少下面这些情况可以创建分支：

- **试验性更改：**比如想测试新算法，看看是否会更快；或者为某个特别的模式重构部分代码。这时就可以创建新分支来开展工作，与那些须要立即提交和上传的变更分离开来。
- **增加新功能：**为每个新功能的开发创建新分支。完成该功能的开发后，就可以合并回主分支。这时可以选用一般方法合并该分支上所有的历史记录，或者将所有历史记录压合在一份提交中。关于不同的合并策略，将在 5.3 节“合并分支间的修改”（第 59 页）中详细介绍。
- **Bug 修复：**不管是修复未发布代码中的 Bug，还是修复已经标记发布代码中的 Bug，都可以创建新分支来跟踪对该 Bug 的修改。你可以灵活使用不同方法，将 Bug 修改合并回原来的代码中，这与使用分支开发新功能时的情形差不多。当对 Bug 进行试验性修复的时候，这种方法特别有用。

如何创建并使用分支，与具体项目环境密切相关。我使用 Git 来跟踪对本书的修改，并尝试在若干种情况中使用分支。

**beta** 分支中包括了本书准备上市时的最新拷贝，且每章都分属于一条不同的分支。对本章的修改都在该分支上完成，并随后将它们合并到 **beta** 分支上。

本书写作中也使用了不少临时分支。例如，对于勘误，创建了一条“errata”（勘误）分支进行纠正，之后再合并回来，最后再删除该临时分支。

我们已概要介绍了分支的基本功能，下一节开始介绍如何创建新分支。

## 5.2 创建新分支

Version control with Git

创建新分支很简单，只须要使用 `git branch` 命令即可，但要在参数中给出新分支名称。创建分支的命令如下：

```
prompt> git branch new
```

Git 在创建新分支后，并不会向用户明确汇报。你可以再次运行不带任何参数的 `git branch` 命令来检查创建的情况：

```
prompt> git branch
* master
  new
```

结果显示了刚刚建立的名为 `new` 的分支。但请注意，主分支名称前有星号，表示它是当前的检出（*check out*）分支。检出分支就是内容检出到当前工作目录树的分支<sup>2</sup>。

为在新创建的分支下修改文件，只须要检出该分支即可。这样，使用 `git checkout` 命令以检出分支：

```
prompt> git checkout new
Switched to branch "new"
```

切换分支的操作已完成。运行不带参数的 `git branch` 命令查看切换分支后的当前分支：

```
prompt> git branch
  master
* new
```

创建分支后，通常都希望立即检出该分支，以便在该分支上展开工作。Git 为此提供了快捷方法，只须在 `git checkout` 命令后加上参数`-b`，就可以一步完成创建分支并检出该分支。

下面使用 `git checkout -b` 创建另一条分支。要注意的是，每条分支的名称必须是唯一的，因此须要选取一个新名称，避免使用前面的 `master` 或 `new`。

用快捷方法创建 `alternate` 分支的命令如下：

```
prompt> git checkout -b alternate master
Switched to a new branch "alternate"
```

注意命令中的第三个参数 `master`，该参数告诉 Git，不是从当前分支，而是从主分支上创建新分支。（即，基于主分支的末梢创建新分支。）

你也可传给 `git branch` 其他的分支名，这样就可以在指定的任意分支上创建新分支。

现在分支已经创建并且检出，你可以修改和跟踪该分支上的文件了。接下来介绍如何合并分支间的修改。

---

<sup>2</sup> 在其他版本控制工具中，检出有时（比如在 ClearCase 中）是指把某个文件的某个版本检出，有时（比如在 Subversion 中）是指把版本库的某个整体版本检出。在 Git 中，检出是指把版本库的某个整体版本检出。如果命令参数是分支，就把分支末梢的版本检出，同时设定，将来提交改动时，提交到该分支上。  
—译者注

## 5.3 合并分支间的修改

从本章到第 8 章的第 1 节都是关于分支操作的。

分支对组织版本库非常有用，但必须要在分支间共享修改。共享是由分支合并 (*merge*) 来实现的。

合并操作就像这个词听起来的那样，把两条或多条分支合并到一起。实际上有好几种分支合并方法。下面介绍主要的三种。

- 直接合并 (*straight merge*)：把两条分支上的历史轨迹合并，交汇到一起。
- 压合合并 (*squashed commits*)：将一条分支上的若干个提交条目压合成一个提交条目，提交到另一条分支的末梢。
- 捣选合并 (*cherry-picking*)：拣选另一条分支上的某个提交条目的改动带到当前分支上。

### 直接合并

将一条分支直接合并到另一条分支。如果想把一条分支的全部历史提交合并到另一条分支上，可以采用这种方式。

下面将通过在刚创建的 `alternate` 分支中添加一些新修改来介绍这种合并方法。请添加一个名为 `about.html` 的新文件，并加入一些介绍信息。

创建好该新文件后，请将它添加到暂存区并提交到版本库中：

```
prompt> git add about.html
prompt> git commit -m "add the skeleton of an about page"
Created commit 217a88e: add the skeleton of an about page
1 files changed, 15 insertions(+), 0 deletions(-)
create mode 100644 about.html
```

现在就在 `alternate` 分支下包含一个提交，且该提交并不存在于主分支上，可以使用 `git merge` 命令将两条分支合并起来。

首先，请切换到合并操作的目标分支。在上面这个例子中，指的是主分支：

```
prompt> git checkout master
Switched to branch "master"
```

然后就可以使用 `git merge` 命令了。使用 `git merge` 命令最简单的形式，只须要指定想合并到当前分支的源分支名称：

```
prompt> git merge alternate
Updating 9a23464..217a88e
Fast forward
 about.html | 15 ++++++-----+
 1 files changed, 15 insertions(+), 0 deletions(-)
 create mode 100644 about.html
```

这样 `alternate` 分支上的修改就合并到主分支上了。

## 压合合并

功能分支（Feature branch）是 Git 中常见的分支用法。你可以在已有分支上创建新的分支，用于新功能开发或 Bug 修复，然后通过一个压合提交将新分支上的修改合并回来。

压合指的是 Git 将一条分支上的所有历史提交压合成一个提交，提交到另一条分支上。要小心使用压合提交，因为大多数情况下，每个提交都应该作为一个单独的条目存在于历史记录中。如果某条分支上的所有提交都密切相关，应随后作为一个整体记录（在父分支上）时，适合做压合提交。

当想开发一些试验性的新功能或修复 Bug 时，这种合并就很有用，因为此时所需要的并不是（长期）记录和跟踪每个试验性的提交，只是最后的成果。

为了演示压合提交，请从主分支上创建一条名为 `contact` 的分支并检出：

```
prompt> git checkout -b contact master
Switched to a new branch "contact"
```

现在请添加一个名叫 `contact.html` 的文件，并在文件中添加你的电子邮件地址，然后提交对该文件的修改。

```
prompt> git add contact.html
prompt> git commit -m "add contact file with email"
Created commit 5b4fc7b: add contact file with email
 1 files changed, 15 insertions(+), 0 deletions(-)
 create mode 100644 contact.html
```

请再添加第二条电子邮件地址，因而在该文件中有你的两条联系方式，然后提交修改：

```
prompt> git commit -m "add secondary email" -a
Created commit 2f30cccd: add secondary email
 1 files changed, 4 insertions(+), 0 deletions(-)
```

目前在 `contact` 分支中已经有两个提交了，你可以将这两个提交压合成主分支上的一个提交，首先，请切换到主分支上：

```
prompt> git checkout master
Switched to branch "master"
```

然后调用 `git merge` 并给它传递参数`--squash`。参数`--squash`告诉 `git merge` 命令，将另一条分支上的全部提交压合成当前分支上的一个提交：

```
prompt> git merge --squash contact
Updating 217a88e..2f30ccd
Fast forward
Squash commit -- not updating HEAD
 contact.html | 19 ++++++-----+
 1 files changed, 19 insertions(+), 0 deletions(-)
 create mode 100644 contact.html
```

此时，在 `contact` 分支上的两个提交已经合并到当前工作区并暂存，但还没有作为一个提交，提交到版本库中。请运行 `git status` 命令查看当前状态：

```
prompt> git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file: contact.html
#
```

像其他“正常”提交一样，现在把这些改动提交到版本库中。

```
prompt> git commit -m "add contact page" \
    -m "has primary and secondary email"
Created commit 18f822e: add contact page
1 files changed, 19 insertions(+), 0 deletions(-)
create mode 100644 contact.html
```

压合合并与直接合并各有用途。然而，有时候须要合并的仅仅是一个提交，此时适合采用拣选合并。

## 拣选合并

有时候分支间只须要合并一个提交，而不须要合并一条分支上的全部改动，因为该分支上可能含有尚未完成的新功能，或者包含其他一些尚不能合并的改动。

那么，什么时候只须要合并一个提交呢？也许是一个已修复的 Bug，或者一个用于分支间共享的类。使用 `git cherry-pick` 命令可合并单个提交。

正如“拣选”（cherry-picks）这个词的意思那样，它拣选一个提交并将它添加到当前分支的末梢。为演示拣选合并，请在 `contact` 分支中创建第三个提交。

首先，请确认当前分支是 `contact` 分支：

```
prompt> git checkout contact  
Switched to branch "contact"
```

然后，转换到 `contact` 分支，请在 `contact.html` 文件中添加一条新链接，比如你在 Twitter 中的账号，然后提交：

```
prompt> git commit -m "add link to twitter" -a  
Created commit 321d76f: add link to twitter  
1 files changed, 4 insertions(+), 0 deletions(-)
```

使用提交名称 `321d76f` 就可以随时对它进行拣选操作。请记住，提交名称具有全局唯一性特征，因此实际操作中的提交名称与本例中的名称不一样。下面将它拣选合并到主分支上：

```
prompt> git checkout master  
Switched to branch "master"  
prompt> git cherry-pick 321d76f  
Finished one cherry-pick.  
Created commit 294655e: add link to twitter  
1 files changed, 4 insertions(+), 0 deletions(-)
```

当然，该例子不会给人深刻印象，毕竟只从 `contact` 分支上带回一行代码的修改，但它演示了拣选操作的基本步骤。

假定在重构现有代码的过程中，在一条分支下创建了一个新类，该类实现了某个通用模式，比如注册模式，现在想把这个改动拿到另一条分支上。但问题是，如果进行合并操作，这条分支上的全部改动都会带过来，但显然不是你想要的结果。

此时你可使用 `git cherry-pick` 命令来拣选出所需要的改动。Git 默认使用拣选出的提交创建新提交。大多数情况下可以这样，但当新提交包含拣选的多个提交时，该如何操作呢？

要拣选多个提交，须要给 `git cherry-pick` 命令传递参数`-n`。这就告诉 Git 在创建提交前须要连续进行合并操作。下面介绍具体步骤。

首先，请使用 `git reset` 命令恢复前面的修改。该命令前面还未介绍过，但请不必担心，在 6.6 节“撤销修改”（第 85 页）中会详细介绍。

```
prompt> git reset --hard HEAD^
HEAD is now at 18f822e add contact page
```

现在删除了（主干末梢上的）最后一个提交。请再次使用 `git cherry-pick` 命令，并带`-n` 参数：

```
prompt> git cherry-pick -n 321d76f
Finished one cherry-pick.
```

请注意，完成本次拣选操作时 Git 停了下来，而不是立即提交。运行 `git status` 命令可以看到，改动已被暂存，等待提交：

```
prompt> git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: contact.html
#
```

接着可以进行下一个拣选操作。一旦拣选完需要的各个提交，就可一并提交改动，并添加任意的提交留言。

接下来继续进行提交操作，但不要使用`-m` 参数。编辑器会使用刚刚拣选的提交的提交留言，作为现在的提交留言。

```
prompt> git commit
Created commit 0bb3dfb: add link to twitter
1 files changed, 4 insertions(+), 0 deletions(-)
```

合并操作的基本内容就介绍完了。到目前为止，所有的合并操作都相当简单和直接，但合并的时候出现问题该怎么办？请学习下一节中的处理冲突内容。

## 5.4 冲突处理

冲突（Conflict）

如果在两条分支上编辑同一个文件，做不同的修改，然后合并这两条分支，将会发生什么情况？通常情况下 Git 会成功自动合并，但并不总是这样。

Git 不能自动合并时，称之为冲突（*conflict*）。冲突总是发生在对不同分支上的同一文件的同一文本块以不同的方式修改，并试图合并的时候。例如，你可能会在不同的分支中使用不同的变量名称，而 Git 又不能确定该使用哪个变量时，它就会停下来等待人工处理。

下面演示如何处理冲突。首先，创建名为 `about` 的新分支：

```
prompt> git checkout -b about master
Switched to branch "about"
```

然后，添加几行内容介绍你最喜欢的编程语言。保存文件并提交到版本库中：

```
prompt> git add about.html
prompt> git commit -m "a list of my favorite programming languages"
Created commit 01fe684: a list of my favorite programming languages
1 files changed, 7 insertions(+), 0 deletions(-)
```

接着再创建一条名叫 `about2` 的分支，但先不要切换到该分支：

```
prompt> git branch about2 about
```

切换分支前，请在 `about.html` 文件中再添加一行介绍你喜欢的编程语言，然后提交修改：

```
prompt> git commit -m "add Javascript to list" -a
Created commit 9e114ac: add Javascript to list
1 files changed, 1 insertions(+), 0 deletions(-)
```

此时已在 `about` 分支下做此修改，而 `about2` 分支中则没有，因为 `about2` 分支创建于第二次提交之前。

现在请切换到 `about2` 分支，并刷新 `about.html` 文件查看，结果是刚才所做的最后一次修改并没有出现在 `about2` 分支上：

```
prompt> git checkout about2
Switched to branch "about2"
```

下面请在 `about.html` 文件中刚才修改过的同一行位置添加不同的内容。

然后保存并提交修改:

```
prompt> git commit -m "add EMCAScript to list" -a
Created commit b84fffdc: add EMCAScript to list
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在已埋下了潜在的冲突。当然，通常情况下，用户意识不到这一点，直到合并时 Git 报告冲突。下面我们操作一下。

先切换回 `about` 分支，然后将 `about2` 分支合并到 `about` 分支上:

```
prompt> git checkout about
Switched to branch "about"
prompt> git merge about2
Auto-merged about.html
CONFLICT (content): Merge conflict in about.html
Automatic merge failed; fix conflicts and then commit the result.
```

请注意命令输出中 `CONFLICT` 那一行，它表示 `about.html` 文件中有冲突。

`about.html` 文件中的内容是这样的:

```
<ul>
  <li>Erlang</li>
  <li>Python</li>
  <li>Objective C</li>
<<<<< HEAD:about.html
  <li>Javascript</li>
=====
  <li>EMCAScript</li>
>>>>> about2:about.html
</ul>
```

从中可以看到最后一个`<li>`元素中有两个不同的名称。第一个是早先做的修改，第二个是刚刚在 `about2` 分支下做的修改。冲突代码开始于:

```
<<<<< HEAD:about.html
```

来自另一条分支上的冲突代码显示在=====分隔号之后并且其后跟随一行:

```
>>>>> about2:about.html
```

`<<<<<`和`>>>>>`代表两部分内容。第一，`<<<<<`后面跟随的是当前分支中的代码，而`>>>>>`之前的则是另一条分支上的代码；第二，在`<<<<<`和`>>>>>`行文件名之前，是所在分支的名称。本例中是 `HEAD`，请记住，`HEAD` 指向当前分支末梢的那个提交，它和 `about2` 分支有冲突。

对于简单的合并，只须要手工编辑并解决冲突即可。然后保存修改，暂存并提交，就像“正常”提交一样。

对于较复杂的合并，可以使用可视化合并工具，比如 Linux（和 Windows）中的 kdiff3<sup>3</sup> 和 OS X 中的 opendiff<sup>4</sup>，它们都很好使用。在命令行敲入 `git mergetool` 命令时，Git 会启动一个合并工具。

Git 会尽可能地找到一个合并工具。有时它会找到多个合并工具，并提示用户为本次合并选取一个合并工具。

```
prompt> git mergetool
Merging the files: about.html

Normal merge conflict for 'about.html':
{local}: modified
{remote}: modified
Hit return to start merge resolution tool (opendiff):
```

命令 `git mergetool` 查看本地 Git 设置中的 `merge.tool` 的值<sup>5</sup>，然后扫描本地系统中的合并工具。这里说的合并工具可以是：kdiff3、tkdiff、meld、xxdiff、emerge、vimdiff、gvimdiff、ecmerge 及 opendiff。

本例中，只须要按下回车键就可启动 OS X 操作系统的 opendiff 中的 FileMerge 工具，详见下页图 5.1。

每个合并工具都略有差异，全都一步一步讲解是不可能的，但总的来说使用它们得到的结果都差不多，都是显示各冲突区域，供用户选取定夺。

在解决了所有冲突后，Git 自动暂存修改，等待提交。提交时，若不用 `-m` 参数，则 Git 会使用刚才合并操作相关的信息来填充提交留言：

```
prompt> git commit
Created commit f846762: Merge branch 'about2' into about
```

<sup>3</sup> <http://kdiff3.sourceforge.net>

<sup>4</sup> opendiff 是苹果公司“Xcode”中的一部分，可以从 <http://developer.apple.com/tools/> 下载。

<sup>5</sup> 也就是说，用命令 `git config --global merge.tool kdiff3` 可把合并工具设置为 kdiff3。当然，前提是你要安装了 kdiff3，并且它在你的 path 环境变量中。—译者注

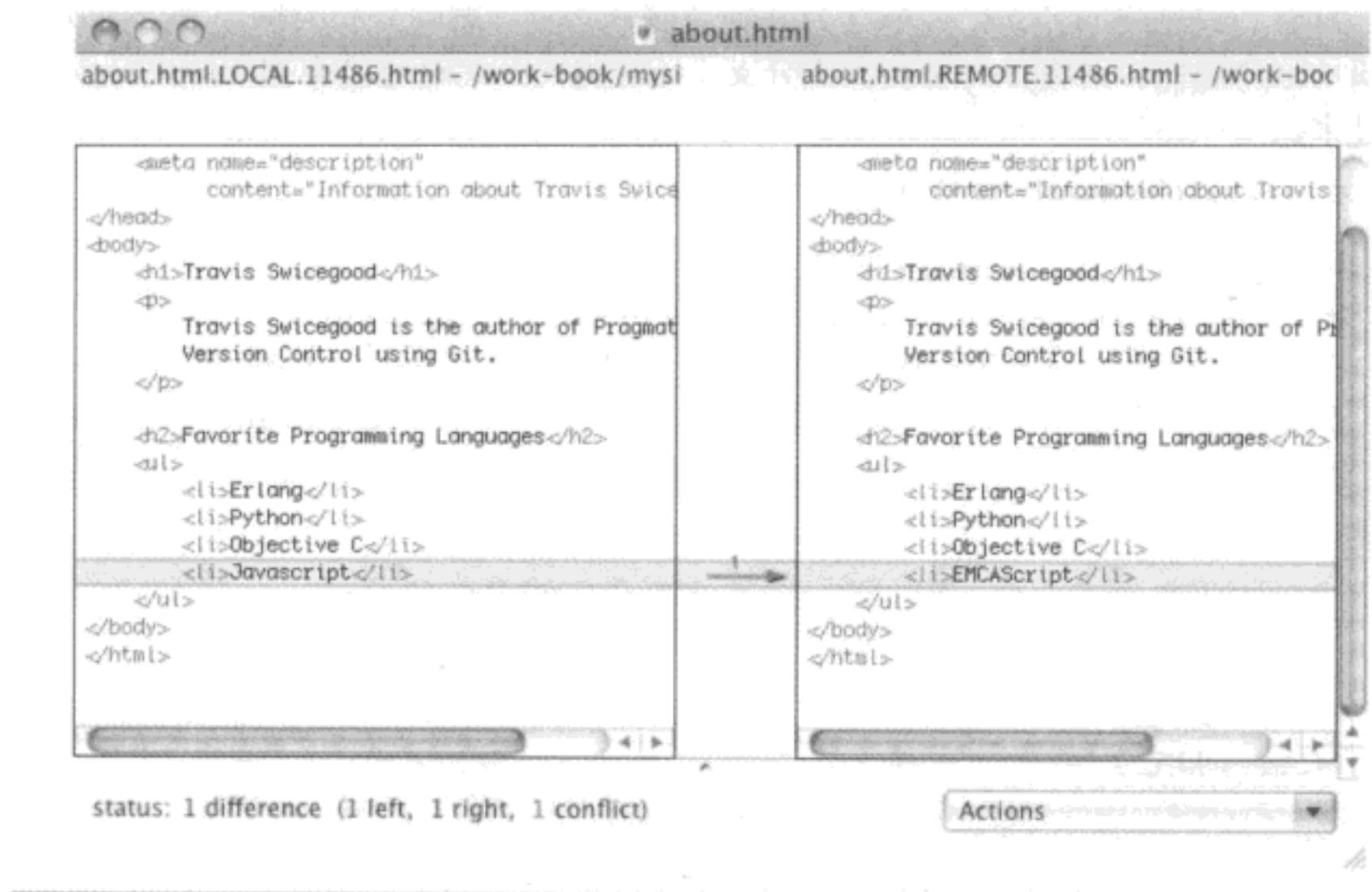


图 5.1 OS X 的 FileMerge 工具

上面介绍了创建分支与合并分支，以及如何解决偶尔发生的合并冲突，这是关于分支的最常见的 3 类操作。下面将学习工作环境清理：删除分支。

## 5.5 删除分支

Deleting Branches

有些分支没必要长期保存，比如分支中的代码已经打了标签并已发布<sup>6</sup>；或者试验分支已成功完成工作或中途废弃，等等。这时应该删除不再使用的分支，以免版本库中的分支过多，难以管理。

将代码合并回当前分支后，就可以使用带参数-d 的 git branch 命令删除旧分支了，例如删除上面创建的 about2 分支：

```
prompt> git branch -d about2
Deleted branch about2.
```

<sup>6</sup> 由于已经打了标签，Git 删除该分支时，从版本树起始到此标签间的全部历史轨迹均会保留，这时删除分支操作只是删除分支本身的名字。因此可以说，该分支“没有必要长期保存”。而在其他版本控制工具中，删除分支通常意味着删除分支上的所有历史轨迹。这时就不能说因为“分支中代码已经打了标签”，该分支就“没有必要长期保存”。一译者注

只有当要删除的分支已经成功合并到当前分支时，删除分支的操作才会成功。为演示这个限制，请将分支切换到主分支。还记得吗，此时还没有把 `about` 分支合并过来。

```
prompt> git checkout master
Switched to branch "master"
prompt> git branch -d about
error: The branch 'about' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D about'.
```

如果确定无须合并，但须要删除 `about` 分支，则运行命令 `git branch -D about`，把参数`-d`换为`-D`，就可以强制删除分支。使用参数`-D`，Git 就无须再检查要删除的分支上的内容是否都已经合并过来了。<sup>7</sup>

在结束本章关于分支介绍之前，我们来看分支的重命名。

## 5.6 分支重命名

### *Renaming Branches*

分支名称通常起提醒功能，告诉开发人员当前正在哪里工作。有时这样的提醒信息须要改变，比如说，里面有一个拼写错误。

就前面的 `contact` 分支而言，开始时其文件中只写了一种联系方法，而现在已经三种联系方法。为了表达更准确，现在将分支名称更改为 `contacts`：

```
prompt> git branch -m contact contacts
prompt> git branch
  about
  alternate
  contacts
* master
  new
```

参数`-m` 不会覆盖已有分支名称，所以新分支名称（也即命令行中第二个分支名称）必须是唯一的。如果使用已经存在的分支名称，Git 会报告如下反馈信息：

```
prompt> git branch -m master contacts
fatal: A branch named 'contacts' already exists.
```

将参数`-m` 改为`-M`，就可以覆盖已有分支名称了。大写“`M`”通知 Git，如果当前分支名称已经存在，则强制覆盖现有分支。这种操作要小心使用。

---

<sup>7</sup> 这里所说的合并指的是直接合并。直接合并时，A 分支和 B 分支的轨迹合并在一起。删除 A 分支时，B 分支还在，因此从 B 的末端回溯到版本树起点的各路径上的所有内容都会保留，也就是说 A 分支上的全部历史记录都还在，删除 A 分支，仅意味着删除 A 分支名称本身，因此“`-d`”就够了。而如果分支还没有直接合并，则删除分支名称后，分支上的历史记录可能没有分支名或标签名来“牵头”了，于是可能会被真的删掉，比较危险。因此这时要用“`-D`”。对于压合合并和拣选合并，由于分支的轨迹并没有合并在一起，因此删除分支必须用“`-D`”而非“`-d`”来删除。—译者注

本章分支已介绍完成。我们学习了创建和维护分支的方法。理解分支如何工作是有效使用 Git 的基础，要扎实掌握。

到目前为止，我们已介绍了如何使用 Git 管理代码的不断向前演进，但对于任何版本控制系统而言，还有一个关键内容是让人们看到曾经做过的工作。这方面的内容，将从第 6 章“查询 Git 历史记录”开始介绍。





忘记历史的人将重蹈覆辙。

► George Santayana

## 第 6 章

# 查询 Git 历史记录

Working with Git's History

对历史的记录和管理是版本管理系统的关键功能。在 Git 中，添加新文件或修改已有文件，都会以提交为单位记录下来，形成历史。

本章将介绍如何进行下列操作：

- 利用 `git log` 命令查看版本库的历史记录。
- 指定提交范围，便于搜索目标。
- 查看提交间差异。
- 逐行查看代码修改历史。
- 跟踪内容，即使内容在文件间移动。
- 撤销所做的改动。
- 改写版本库历史。

检查历史记录可以提供非常宝贵的信息。例如面对问题“这个文件怎么不符合编码规范？”时，快速检查文件历史，可能发现只显示一条“为客户演示准备”的提交，在此之后没有被修改过。

如果想查看更加详细的信息，Git 能够显示谁在何时修改了哪行代码，它还可以进一步显示该文件两个版本之间的差异。

最后，Git 允许你修改版本库中的历史记录。请在使用此功能时务必小心，否则可能造成的后果是：团队其他成员花费一上午的时间想方设法解决分支合并时出现的奇怪问题。

本章将学习如何处理前面所述各项任务。下面从 `git log` 日志查询命令开始。

本章仍然使用贯穿本书始终的版本库，如果没有相应的拷贝，还跟着本章例子一起学习，可以从 GitHub 上克隆此版本库：

```
prompt> git clone git://github.com/tswicegood/mysite-chp5.git
Initialized empty Git repository in /work/mysite-chp5/.git/
remote: Counting objects: 41, done.
remote: Compressing objects: 100% (35/35), done.
remote: Total 41 (delta 14), reused 0 (delta 0)
Receiving objects: 100% (41/41), 4.52 KiB, done.
Resolving deltas: 100% (14/14), done.
prompt> cd mysite-chp5
```

## 6.1 查看 Git 日志

### Inspecting Git Logs

查看提交日志是查看版本库历史记录最常用的方法。Git 可以显示提交的日志项，包括是谁，何时做的提交，以及本次提交的具体代码改动（可选的）等信息。

Git 日志像博客一样是按照时间倒序显示的。你可以通过设置各种各样的参数来过滤日志，让我们从基本的用法开始学习。进入工作目录树，在命令提示符后输入 `git log`：

```
prompt> git log
commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 11:06:47 2008 -0500

    add link to twitter
```

```
commit 18f822eb1044761f59aebaf7739261042ae10392
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 10:34:51 2008 -0500
```

```
    add contact page

    has primary and secondary email
```

如果日志输出超过单屏大小，Git 则显示部分输出结果。你可以上下滚动浏览日志。屏幕底部出现的冒号 “:” 表示还有更多信息等待输出。<sup>1</sup>

---

<sup>1</sup> 这时，键入 “j” 向下浏览，键入 “k” 向上浏览，键入 “q” 退出……这些和 vi 编辑器的命令一样。  
—译者注

在不带参数的 `git log` 命令的输出中，每个提交有四部分信息：提交名称、提交人、提交日期、提交留言。前三项内容在前三行显示，第四项内容可能显示多行。接着是下一个提交的信息。

时隔半年之后再查看以前的提交留言，可能并不完全理解当时的意图。在日志信息外，常常要借助查看代码修改本身，才能逐渐回忆起当时的情景。在 `git log` 命令后添加`-p` 选项，Git 就可以显示版本间的代码差异。

你可以仅查看部分提交而非全部提交。在 `git log` 命令后输入参数`-1`，Git 只显示一条提交的日志；输入参数`-2`，则只显示两条提交的日志，以此类推。例如，`git log -10` 则显示最近的 10 个提交。

你也可以给 `git log` 命令传递一个指定的版本，并以此作为查看日志的起始点（查看从那时起到现在的全部提交）：

```
prompt> git log 7b1558c
commit 7b1558c92a7f755d8343352d5051384d98f104e4
Author: Travis Swicegood <development@domain51.com>
Date: Sun Sep 21 14:20:21 2008 -0500

add in hello world HTML
```

注意，上面例子中使用的是提交名称缩写，即 40 位提交名称的前 7 位字符。不论输入几位（至少 4 位），Git 都会设法匹配。如果匹配不上或匹配到多个结果，Git 就会报错。通常 4 位或 5 位字符就足够匹配了，但 7 位或 8 位字符几乎保证了匹配的唯一性。

以上是 `git log` 的基本用法。下面介绍一些更复杂的方法指定要查看哪个或哪些提交。

## 6.2 指定查找范围

Specifying Revision Ranges

如果不是像历史学家那样善于记忆，要记下历史事件的发生的年月日可真让人头疼。大致了解美国历史容易，记下每个具体日期就难了，建国日除外。

同样，当编程的时候，恐怕记不住所有事情发生的准确时间，而仅仅是“上周一或上周二添加了一个新功能”，“今天上午修复了一个 Bug”等。没关系，Git 具有的丰富功能只凭上述大致时间范围就能查找到你想要的信息。

Git 提供了若干有效的方法来指定查找范围。假如只想查看最近 5 小时内的提交，可以在 `git log` 命令后添加`--since="5 hours"`:

```
prompt> git log --since="5 hours"
commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 11:06:47 2008 -0500

add link to twitter
```

同样，也可以通过添加`--before="5 hours" -1`来查看 5 小时之前的最后一个提交:

```
prompt> git log --before="5 hours" -1
commit 18f822eb1044761f59aebaf7739261042ae10392
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 10:34:51 2008 -0500
```

`add contact page`

`has primary and secondary email`

`--since` 和`--before` 参数接受大多数英文格式的日期。Git 工具本身能够识别诸如`--ince="24 hours"`、`--since="1 minute"`、`--before="2008-10.01"`这样格式的日期，哪怕日期中间既有连字符又有句点。

也可以指定两个版本，用“最老版本..最新版本”这种格式作为查找范围。最老版本在前，最新版本在后:

```
prompt> git log 18f822e..0bb3dfb
commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 11:06:47 2008 -0500
```

`add link to twitter`

乍一看，上面的输出结果似乎是错误的。告诉 Git 查看从 `18f822e` 版本到 `0bb3dfb` 版本的日志，似乎输出结果中应包含 `18f822e` 这个版本的日志，但是在 Git 世界里，输入的版本范围的含义是 `18f822e` 之后的提交开始显示日志（不包括起点，只包括终点）。

设定版本及版本范围时，`HEAD` 代表版本库里当前分支末梢的最新版本。在前面的例子中，用 `HEAD` 替换 `0bb3dfb`，得到相同的输出结果：

```
prompt> git log 18f822e..HEAD
commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 11:06:47 2008 -0500
```

add link to twitter

也可以不输入 HEAD 参数，因为 Git 假定“..”后面省略的值为 HEAD：

```
prompt> git log 18f822e..
commit 0bb3dfb752fa3c890ffc781fd6bd5dc5d34cd3be
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 11:06:47 2008 -0500
```

add link to twitter

当指定查找范围时，也可以用标签名称替换提交名称。这对于查看某一特定标签之后的修改，以及查看两个标签之间的历史都是有用的，如下所示：

```
prompt> git log --pretty=format:"%h %s" 1.0..HEAD
0bb3dfb add link to twitter
18f822e add contact page
217a88e add the skeleton of an about page
9a23464 rename to more appropriate name
6f1bf6f Change biography link and add contact link
4333289 add in a bio link
```

注意这次在 `git log` 命令后添加了参数`--pretty, format:"%h %s"`告诉 Git 显示提交名称（哈希值）的缩写，以及提交留言的第一行，即标题。

另一个更常见的用法是`--pretty=oneline`，但是这个输出格式太宽了，以至于无法完全显示在本书的页面中。在 `git log` 的命令手册里还列出了各种各样的格式可供参考使用。

如果指定了范围而 `git log` 命令没有输出任何结果，则须要检查输入范围是否有效。记住，先要指定旧版本，否则 Git 什么也不显示，连提示都没有。

另一种指定版本的常见方法是指出它和另一版本的关系，此时有两种操作符可供使用：

- ^：一个脱字号作用相当于回溯一个版本。`18f822e^`是指 `18f822e` 之前的那个版本，即父节点。也可以使用多个脱字号：`18f822e^^`是指 `18f822e` 之前版本的之前版本，即祖父节点。以此类推。<sup>2</sup>在 Windows 系统下，如果版本中带有脱字号，则版本外须要添加双引号，例如 `18f822e^` 应表示为"`18f822e^`"，否则不能识别。

<sup>2</sup> 事实上，脱字号比这更玄妙。回溯历史，当遇到某个节点（通常是版本合并后的节点）有并列的多个父节点时，“^1”代表第一个父节点，“^2”代表第二个，以此类推。而“^”是“^1”的简写。一译者注

\* ~N: 波浪字符加数字的操作符是指回溯 N 个版本。借用刚才的例子，`18f822e~1` 是指 `18f822e` 的父节点，`18f822e~2` 是指 `18f822e` 的祖父节点。

可以混合使用这两种操作符。以下三个命令将得到一样的结果：

```
prompt> git log -1 HEAD^^^
... log entry ...
prompt> git log -1 HEAD^~2
... log entry ...
prompt> git log -1 HEAD~1^^
... log entry ...
prompt> git log -1 HEAD~3
... log entry ...
```

也可以在早先提到的查找范围里使用带有脱字号或波浪字符的命令，如下所示：

```
prompt> git log HEAD~10..HEAD
fatal: ambiguous argument 'HEAD~10..HEAD': unknown revision or path
not in the working tree.
Use '--' to separate paths from revisions
```

从上面的例子可见，当指定的版本不存在时，会得到“未知版本”的报错信息。以上我们学习了如何使用指定的版本范围和过滤条件来显示日志，下面将学习如何查看不同版本之间的差异。

## 6.3 查看版本之间的差异

Looking at Differences Between Versions

在 4.3 节“查看修改内容”（第 48 页）中我们学习了用命令 `git diff` 查看工作拷贝和版本库中最新版本（HEAD）之间的差异。下面使用版本库中的其他历史版本：（下面例子显示的是 `18f822e` 这个版本和当前工作目录树间的差异。）

```
prompt> git diff 18f822e
diff --git a/contact.html b/contact.html
index 64135cb..63617c2 100644
--- a/contact.html
+++ b/contact.html
@@ -13,6 +13,10 @@
 <p>
     <a href="mailto:tswicegood@gmail.com">Gmail</a>
 </p>
+
+<p>
+    <a href="http://twitter.com/tswicegood">Twitter</a>
+</p>
</body>
</html>
```

在命令 `git diff` 里，指定版本范围的方法与 `git log` 一样。唯一的差别是 `git diff` 输出的是最老版本与最新版本之间的差异，而不是按提交条目一个一个地显示。

在 `git diff` 命令中使用标签作为参数，是一种获取发布版本之间代码量统计的好方法。可以利用它计算出修改和删除的代码行数。

这是通过一个很酷的参数`--stat` 实现的。在 `git diff` 命令中通过它可以得到变更统计数据：

```
prompt> git diff --stat 1.0
about.html    | 15 ++++++=====
contact.html  | 23 ++++++=====
hello.html    | 13 ++++++=====
index.html    |  9 -----
4 files changed, 51 insertions(+), 9 deletions(-)
```

可以看出，用它可以统计上周改动的代码量，或者最新标签之后改动的代码量。

注意，在这个例子中命令 `git diff` 只传送了一个标签做参数。如果省略第二个参数，则 Git 把 `HEAD` 当作默认值。<sup>3</sup>

我们已经学习了如何查询历史上的提交信息，以及如何获取不同版本之间的差异。下面一起来看看逐行的历史记录吧。

## 6.4 查明该向谁问责

Finding Out Who's to Blame

当查看代码的时候，可能发现有一处（没准一行，没准十行）不该出现在这个位置，还可能发现有一处深层嵌套应该重写，以便理解和维护。

此时查看文件的历史，以及版本之间的差异固然有帮助，但还有一个更强大的命令 `git blame`，可以用来查看特定代码块的历史信息。该命令的输出结果是代码块中的每行代码前附加前缀信息，其中包括提交名称、提交人和提交时间。

例如，对 `hello.html` 文件用 `git blame` 命令，前两行显示输出结果如下所示（这里显示的是姓名的缩写，以便本书页面的排版）：

```
prompt> git blame hello.html
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 1) <html>
a5dacabd index.html (Travis S. 2008-09-21 20:37:47 -0500 2) <head>
```

输出结果里有几条有趣的信息。首先，每行前 8 位字符是提交的哈希值。注意，第一行有个脱字号，表示版本库中第一个提交。

---

<sup>3</sup> 准确地讲，默认值是工作目录树，而非 `HEAD`。注意，工作目录树中，所有纳入版本控制的文件将被比较，而新增的还未被纳入 Git 版本控制的文件则不在其列。一译者注

其次,请注意列出的是文件名 `index.html`。还记得是什么时候创建这个文件的吗?其实文件 `hello.html` 的初始名为 `index.html`。输出结果中第二列显示的是当初提交该行时的文件名,紧随其后显示的是提交人姓名、提交时间、行号,以及该代码行。

当然,要查看文件中全部代码行的相关信息的时候不多。在 Git 中通过给命令 `git blame` 传递参数`-L` 能够轻松地缩小范围。它通过指定范围来告诉 Git 只显示某些特定行的注释,此范围用参数“`<开始>,<结束>`”表示。

`<开始>`和`<结束>`参数可以是数字。例如,下面例子是显示文件 `hello.html` 的第 12 行和第 13 行日志:

```
prompt> git blame -L 12,13 hello.html
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 12) </body>
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 13) </html>
```

`<结束>`这个参数不必非得指定一个数字,可以使用`+N` 或者`-N` 指定范围,使用这种方式生成的结果和前面例子中的结果完全相同:

```
prompt> git blame -L 12,+2 hello.html
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 12) </body>
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 13) </html>
```

使用`-N` 记号可以减 `N` 行。例如可以使用`-L 12, -2` 显示第 12 行和第 11 行日志:

```
prompt> git blame -L 12,-2 hello.html
4333289e index.html (Travis S. 2008-09-22 07:54:28 -0500 11) </ul>
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 12) </body>
```

使用数字作为参数常常很方便,因为在查看文件内容时,大多数时候都会使用带有行号的编辑器打开文件。但 Git 还提供了另外一种指定`<开始>`和`<结束>`参数的方法:正则表达式。<sup>4</sup>

---

<sup>4</sup> Git 使用的是 POSIX 风格的正则表达式(而非 Perl 风格的)。

介绍正则表达式的书很多<sup>5</sup>，所以在这里只举个例子，不做更多介绍了。前面例子中的命令可以使用正则表达式改写如下：

```
prompt> git blame -L "</\body>/",+2 hello.html
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 12) </body>
^7b1558c index.html (Travis S. 2008-09-21 14:20:21 -0500 13) </html>
```

当然，很多时候可能由于编码规范改变了，或者有人正在执行一些清理工作，通过 `git blame` 命令得到的结果就会意义不大。不过可以使用前面 6.2 节“指定查找范围”（第 73 页）中讨论过的任何一种方法去查找想要的结果。例如，查看文件 `hello.html` 在 `4333289e` 之前的提交日志：

```
prompt> git blame -L "</\body>/",-2 4333289e^ -- hello.html
fatal: no such path hello.html in 4333289e^
```

噢，出错是因为在 `4333289e^` 这次提交里 `hello.html` 并不存在，实际当时的文件名仍然是 `index.html`。用 `index.html` 作为文件名再次执行命令。注意，在 `4333289e^` 和实际文件名之间有一个--符号，这是在通知 Git 查询指定文件。

```
prompt> git blame -L "</\body>/",-2 4333289e^ -- index.html
^7b1558c (Travis Swicegood 2008-09-21 14:20:21 -0500 7) <h1>Hello...
^7b1558c (Travis Swicegood 2008-09-21 14:20:21 -0500 8) </body>
```

前面介绍了命令 `git blame` 的基本用法。除了可以跟踪并显示文件重命名外，它还可以跟踪并显示文本块的复制粘贴，这是接下来要学习的内容。

## 6.5 跟踪内容

Following Content

就像在 4.4 节“管理文件”（第 52 页）中讨论的那样，Git 能够跟踪文件里甚至文件间的内容移动。在尽力追查某几行代码的初始提交和原作者时，这项功能非常有用。

在 C 语言里，有的代码行只有一个大括号，这样的代码行，彼此之间容易误配。类似的情况也发生在 Python 语言中的构造函数上。为了尽量避免误配，Git 至少要匹配三行才认为是复制和粘贴的结果。

下面将创建一个新文件来演示这项功能。

---

<sup>5</sup> 《Mastering Regular Expressions》 [Fri97] 是一本非常优秀的书。

创建一个名为 `original.txt` 的文件，并且输入以下三行内容：

```
This is the first line.  
This happens to be the second line.  
And this, it is the third and final line.
```

保存、添加并提交到版本库中：

```
prompt> git add original.txt  
prompt> git commit -m "commit of original text file"  
Created commit b87524b: commit of original text file  
1 files changed, 3 insertions(+), 0 deletions(-)  
create mode 100644 original.txt
```

现在编辑这个文件并复制整个内容，即复制前三行并粘贴为随后的三行，然后保存。执行命令 `git diff` 将得到类似下面的输出结果：

```
prompt> git diff  
diff --git a/original.txt b/original.txt  
index 4d9f742..350f1bb 100644  
--- a/original.txt  
+++ b/original.txt  
@@ -1,3 +1,6 @@  
 This is the first line.  
 This happens to be the second line.  
 And this, it is the third and final line.  
+This is the first line.  
+This happens to be the second line.  
+And this, it is the third and final line.
```

提交修改到 Git 版本库。现在可以利用 `git blame` 命令查看文件中每一行的历史记录了：

```
Line 1  prompt> git blame original.txt  
- b87524b7 (... 1) This is the first line.  
- b87524b7 (... 2) This happens to be the second line.  
- b87524b7 (... 3) And this, it is the third and final line.  
5 222cb821 (... 4) This is the first line.  
- 222cb821 (... 5) This happens to be the second line.  
- 222cb821 (... 6) And this, it is the third and final line.
```

例子的实际输出结果比上面显示的要多，其中“...”省略的是名字和时间戳的信息，为了适应本书排版，将它们省略了。

输出结果中的 8 位字符是提交名称缩写。在这之前是行数（这不是 Git 的输出，而是为讲解本例在写作本书时添加上去的）。第 2 行到第 4 行，也就是文件的第 1 行到第 3 行，它们的提交名称相同。而第 5 行到第 7 行也拥有相同的另一个提交名称，这对应于到目前为止的两次提交。

现在重新运行命令 `git blame`, 但是这次添加参数 `-M`。该参数告诉命令 `git blame` 检测在同一个文件内移动或复制的代码行:

```
prompt> git blame -M original.txt
b87524b7 (... 1) This is the first line.
b87524b7 (... 2) This happens to be the second line.
b87524b7 (... 3) And this, it is the third and final line.
b87524b7 (... 4) This is the first line.
b87524b7 (... 5) This happens to be the second line.
b87524b7 (... 6) And this, it is the third and final line.
```

结果显示所有的提交名称都相同。因为 Git 跟踪的是内容, 它在这里检测到了重复的内容。

Git 也可以跟踪文件之间的复制。下面复制文件 `original.txt` 到一个新文件, 命名为 `copy.txt`。

暂存并提交 `copy.txt`。在命令 `git blame` 中添加参数 `-C -C` 来查看文件之间的复制:

```
prompt> git blame -C -C copy.txt
b87524b7 original.txt (... 1) This is the first line.
b87524b7 original.txt (... 2) This happens to be the second line.
b87524b7 original.txt (... 3) And this, it is the third and final line.
b87524b7 original.txt (... 4) This is the first line.
b87524b7 original.txt (... 5) This happens to be the second line.
b87524b7 original.txt (... 6) And this, it is the third and final line.
```

Git 不仅显示初始的提交名称 `b87524b7`, 而且还有初始的文件名 `original.txt`。类似于 `git blame`, 给 `git log` 命令传递 `-C -C` 参数, 也能显示文件复制信息。

当用 `git log` 命令检测文件的复制时, 还得传递参数 `-p`。这样 Git 除显示常规日志信息之外, 还会显示代码的具体变动:

```
prompt> git log -C -C -1 -p
commit 540ecb73f652a882ad235c85b61ffb657d3d4969
Author: Travis Swicegood <development@domain51.com>
Date: Sat Oct 4 17:08:16 2008 -0500

    copy original to show cross-file blame

diff --git a/original.txt b/copy.txt
similarity index 100%
copy from original.txt
copy to copy.txt
```

从上面的输出信息可以看出，文件 `copy.txt` 与 `original.txt` 百分之百地匹配。

到目前为止已介绍了查看历史记录的基本方法，下面介绍如何改写版本库历史。

## 6.6 撤销修改

### *Undoing Changes*

人们常常事后才发现自己的疏忽。对编码工作来说，也不例外。比如提交了修改，才意识到里面包含一个不该透露的密码。

在集中式版本库中，所有的修改都会提交到中央版本库中，因此难以修正。当然，如果拥有管理员的权限，用一些技巧是可以在版本库中修正的，但这相当危险，可能会导致数据的不一致。

Git 为这类错误操作提供了修正的办法。在 Git 中，所有的修改都是在本地进行的，只有推入到公共版本库才能共享。既然本地版本库是你自己的地盘，那么你可以随意改写版本库中的历史记录，而不会影响到别人。

### 改写历史记录的危险

在介绍这部分内容之前，请注意：小心使用下面的命令，特别是当在流程上使用集中模式，即本地版本库里的每个提交都推入到上游版本库的时候。如果在共享了代码改动之后还修改历史，团队其他成员在同步这些代码改动时就会遇到麻烦。

这些命令中，除了本节“反转提交”部分（第 84 页）介绍的 `git revert` 命令外，其他命令都是用来修改版本库历史记录的。这些操作在集中式版本库中是非常危险的。当提交被转移、重命名甚至删除时，版本库很难跟踪。

完全分布式开发的好处之一就是可以只共享准备好的东西。在推入变更（到其他版本库）之前要确保变更是准备好的。在本地修改和提交，直到准备好才共享，这样就不必担心在改写历史的时候对其他人造成影响。

一旦推入了变更，就不能随意修改相应的历史了，除非不怕团队里所有的人抱怨。

当推入变更之后又去修改历史，并且随后又推入不同的变更时，则会给那些已经拿到了之前的变更的同事带来很大的麻烦。关于推入操作的详细内容，参见 7.4 节“推入改动”（第 96 页）。

预先介绍完可能发生的危险，现在让我们开始学习这些操作。首先，增补提交。

## 增补提交

经常会遇到这样的情况，当用一个自己还不熟悉的语言编写程序时，常常会忘记在每行的行尾加上句号或分号。（而在察觉并修复类似的疏忽之前，就已经把代码提交了。）<sup>6</sup>

在 Git 里，处理这些小问题很简单：修改代码并暂存后，提交时加上`--amend`参数就可以了。<sup>7</sup>

为了演示该操作，请在文件 `contact.html` 中添加一个博客链接或你经常访问的网页，但请在该链接中故意留下输入错误，然后提交本次修改：

```
prompt> git commit -m "add link to blog" -a
Created commit c3531c4: add link to blog
 1 files changed, 4 insertions(+), 0 deletions(-)
```

接下来请修复该 URL 后再次提交修改，并在 `git commit` 命令添加`--amend`参数：

```
prompt> git commit -C HEAD -a --amend
Created commit 45eaf98: add link to blog
 1 files changed, 4 insertions(+), 0 deletions(-)
```

该 `git commit` 命令使用了一个新参数`-C`，即告诉 Git 复用指定提交的提交留言，而不是从头再写一个。在这个例子中指定的是 `HEAD`，事实上还可以指定其他有效的提交名称。如果参数是小写的`-c`，就会打开（预先设置好的）编辑器，以便在已有的提交留言基础上编辑修改。

增补提交只能针对最后一个提交。如果想更正好几个提交之前的某个错误，则须使用 `git revert` 这个命令，详见接下来的内容。

<sup>6</sup> 具有讽刺意味的是，我不得不在写本章的时候用一下`--amend` 命令，因为直到提交之后才发现其中某个版本存在未发现的笔误。

<sup>7</sup> 不加`--amend` 参数，正常提交可不可以？当然可以，只是版本库里又多了一个提交记录，看起来乱糟糟。并且所有人都知道你的马虎了。—译者注

## 反转提交

有些时候程序转不起来，因为它须要借助特定的架构或基于第三方软件才能运行，而团队中相关同事又恰好不在，没法立即解决这个问题。（与其空等，不如先临时性地解决。）如果现在这个状况发生在已经提交之后，此时就要撤销这个提交，或者反转（*revert*）它。

反转已经提交的改动，最简单的方法就是使用 `git revert` 命令。此命令通过在版本库中创建一个“反向的”新提交来抵消原来提交的改动。

通常 Git 会立即提交反转结果，但是也可以通过 `-n` 参数告诉 Git 先不要提交，这对于须要反转多个提交非常有用。运行多个 `git revert -n` 命令，Git 会暂存所有的变更；然后做一次性提交。

做反转操作的时候必须提供提交名称，以便让 Git 知道要反转什么。举个例子，想要反转提交 `540ecb7` 和 `HEAD`，命令参考如下。提示，反转总是按照从新到旧的倒序来操作的，即最后的提交最先反转。这样可以避免在反转多个提交的时候，遇到一些不必要的冲突。

```
prompt> git revert -n HEAD
Finished one revert.
prompt> git revert -n 540ecb7
Removed copy.txt
Finished one revert.
prompt> git commit -m "revert 45eaf98 and 540ecb7"
Created commit 2b3c1de: revert 45eaf98 and 540ecb7
 2 files changed, 0 insertions(+), 10 deletions(-)
 delete mode 100644 copy.txt
```

通常情况下，命令 `git revert` 会启动默认编辑器用来编辑提交留言。默认信息会被加入到编辑器中，即：`Revert "your original log message"`，后面跟着 `This reverts commit <提交名称>`。假如想直接用默认的提交留言，可以在命令 `git revert` 中添加参数 `--no-edit`。

正如其他提交一样，要在提交留言中解释清楚为什么要做本次提交。当然，如果只是反转了一个修改也没有什么好说的，就像刚才举的例子。不过最好要加上诸如“不能运行”或“速度太慢”等类似的提示信息，有助于回忆起当初为什么要做出这样的反转操作，哪怕在半年之后，也依然有迹可循。

## 复位

按下回车键的同时，突然意识到刚刚提交的配置文件里有自己的私人密码。估计不少人经历过这样的情况。如果使用集中式版本控制系统，就只能让系统管理员执行某些操作来清除提交中的所有蛛丝马迹。

Git 想到大家会犯这样的错误，于是提供复位版本库到一个特定版本的功能。命令 `git reset` 是以提交名称作为参数的，默认值是 `HEAD`。

可以用“`^`”和“`~`”作为提交名称的修饰符来指定某个版本。`HEAD^`是指要复位两个提交，`540ecb7~3`是指要复位 `540ecb7` 之前的三个提交。<sup>8</sup>

命令 `git reset` 可以在复位版本库后，暂存工作目录树中因复位产生的与版本库的差异，以便提交。这对于在之前的提交中发现错误并须要更改时非常有用。

这是通过`--soft`参数实现的。它使得 Git 暂存所有的因复位带来的差异，但不提交它。之后，用户可以修改这些内容再提交，或者干脆扔掉这些内容。

最后要介绍的一个选项是`--hard`，要小心使用！该选项会从版本库和工作目录树中同时删除提交。`--hard`就好像版本库中的删除键，并且不可以恢复。

下面简单演示这个选项的用法。前面示例中最后一次提交反转了两个文件的修改，假定这并无必要，下面撤销这个提交：

```
prompt> git reset --hard HEAD^
HEAD is now at 45eaf98 add link to blog
```

这样就将版本库复位到 `HEAD` 之前的那个版本了，就好像之前反转那两个文件的事儿从来没有发生过一样。

## 6.7 重新改写历史记录

Rewriting History

修正主义历史学家会篡改历史，以便符合政治上的需要。

正式的代码评审活动会检查代码改动，并在需要时修正代码，以提高其质量。Git 能更进一步，除了浏览代码改动外，还能改写代码改动本身（哪怕它已经提交到版本库了）。

---

<sup>8</sup> 实际上，`HEAD^`是把版本库复位到当前 `HEAD` 之前的那个版本（父节点）上，把 `HEAD` 这个版本的修改扔到工作目录树中。而 `540ecb7~3` 则是把版本库复位到 `540ecb7` 的曾祖父节点上，并把 `540ecb7`、`540ecb7` 父节点、`540ecb7` 祖父节点上的修改扔到工作目录树中。一译者注

下面是几个须要改写历史的例子：

- **给历史记录重新排序，让它们看起来更合理：**难以准确描述何时需要这么做，但当遇到这样的情况时自然就能明白了。一个或多个提交看起来发生的先后顺序不对。
- **将多个提交压合成一个提交：**在完成多个提交之后才意识到，这些提交其实最好算作一个大的提交，因为它们都是为了完成同一任务。
- **将一个大的提交分解为多个提交：**这与上面介绍的压合相反。浏览这个提交中的代码改动后，发现该提交完成了多个任务，应该把它分成多次提交（以便历史记录看起来更清晰）。

对于使用传统的版本控制系统比如说 Subversion 或 CVS 的读者来说，这像是在谈论巫术。“修改已提交的内容”听起来挺有用，但这在技术上能实现么？重新排序？分解提交？天方夜谭！

而这正是分布式版本控制系统——拥有一个和其他开发人员没有联系的私有的版本库——所带来的优势。在集中式版本控制系统中，比如 Subversion 或 CVS，如果修改版本库的历史，那么其他人用版本库中的代码更新自己的工作区时，就会遇到奇奇怪怪的代码冲突。

Git 的交互式 `rebase` 命令，即 `git rebase -i`，能够完成这类改写历史的工作。用这个命令，可以重新塑造历史。下面是当前版本库里的最后三个提交：

```
prompt> git log --pretty=format:"%h %s" HEAD~3..  
45eaf98 add link to blog  
540ecb7 copy original to show cross-file blame  
222cb82 adding copied lines to showcase git blame
```

在下面一节中，学习如何将“45eaf98”这个提交移到另外两个提交发生之前。

## 重新排序提交

可使用命令 `git rebase` 的人机交互模式改写历史记录。交互模式启动的时候会调用事先设置好的编辑器。在 4.2 节“提交修改”（第 45 页）介绍过 Git 如何找到编辑器。使用变基操作的交互模式，须提供重排提交的起始点作为参数。

在这个例子中，我们想从 HEAD~3（之后的那个提交）起，重排提交：

```
prompt> git rebase -i HEAD~3
... launches configured editor ...
pick 222cb82 adding copied lines to showcase git blame
pick 540ecb7 copy original to show cross-file blame
pick 45eaf98 add link to blog

# Rebase b87524b..45eaf98 onto b87524b
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, 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.
#
~
```

在编辑器中以“#”开头的都是注释，并且都会被 Git 忽略。前面三行是须要重新排序的三次提交。现在把 pick 45eaf98 这一行移到最顶上一行。那么前三行就会是这个样子：

```
pick 45eaf98 add link to blog
pick 222cb82 adding copied lines to showcase git blame
pick 540ecb7 copy original to show cross-file blame
```

保存编辑的内容并退出编辑器，Git 开始变基操作。完成之后，可以通过 `git log` 命令来查看新的顺序：

```
prompt> git log --pretty=format:"%h %s" HEAD~3..
8c764d3 copy original to show cross-file blame
be53bab adding copied lines to showcase git blame
4f7621d add link to blog
```

## 将多个提交压合成一个提交

Git 还能做这件事：把添加 blog 和 Twitter 链接的两个提交压合在一起。为此，再此启动交互式变基操作，这次指定的参数是在 Twitter 中添加链接的提交之前的那个版本，也就是 `0bb3dfb^`：

```
prompt> git rebase -i 0bb3dfb^
... launched configured editor ...
pick 0bb3dfb add link to twitter
pick b87524b commit of original text file
pick 4f7621d add link to blog
pick be53bab adding copied lines to showcase git blame
pick 8c764d3 copy original to show cross-file blame
```

```
# Rebase 18f822e..8c764d3 onto 18f822e
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, 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.
#
~  
~
```

这次把 `pick 4f7621d` 这行移到 `pick 0bb3dfb` 之后的位置，并且把 `pick` 修改为 `squash`。操作完成后，前 5 行应该是这样：

```
pick 0bb3dfb add link to twitter
squash 4f7621d add link to blog
pick b87524b commit of original text file
pick be53bab adding copied lines to showcase git blame
pick 8c764d3 copy original to show cross-file blame
```

保存并退出。在变基操作启动之后，将会自动弹出编辑器，以便为正在压合的两个提交填写提交留言。编辑器中的默认信息如下所示：

```
This is a combination of two commits.
# The first commit's message is:
add link to twitter

# This is the 2nd commit message:
add link to blog
```

可以按照自己喜欢的方式来整合这些信息，之后保存并退出，以便 Git 继续完成变基操作。结束后，运行 `git log` 命令查看新的历史：

```
prompt> git log --pretty=format:"%h %s" HEAD~4..
7509494 copy original to show cross-file blame
8184d47 adding copied lines to showcase git blame
9a750b3 commit of original text file
b02376d add link to twitter and blog
```

现在将把 `b02376d` 分解成两个提交来撤销上述修改。

## 将一个提交分解成多个提交

我们刚重新排序了提交，并且将两个提交压合成一个提交。现在把一个提交分解成两个提交。这会复杂一些，但开始时的操作是一样的。

再次使用命令 `git rebase -i`，并修改编辑器中的内容：

```
prompt> git rebase -i HEAD~4
... launches editor ...
edit b02376d add link to twitter and blog
pick 9a750b3 commit of original text file
pick 8184d47 adding copied lines to showcase git blame
pick 7509494 copy original to show cross-file blame
```

注意第一行是以 `edit` 开头的。保存退出，变基操作启动。当运行到 `edit b02376d` 的时候，Git 会停下来并且给出如下提示信息：

```
Stopped at b02376d... add link to twitter and blog
You can amend the commit now, with
```

```
git commit --amend
```

```
Once you are satisfied with your changes, run
```

```
git rebase --continue
```

运行 `git log -1` 显示最后一个提交，也就是刚压合在一起的那个提交，这个提交正是需要编辑修改的提交。

```
prompt> git log -1
commit b02376dbbb76c356d9d44cf65163293db9147d9a
Author: Travis Swicegood <development@domain51.com>
Date:   Sat Oct 4 11:06:47 2008 -0500

add link to twitter and blog
```

此时变基操作暂停，等待用户修改版本库。现在可以通过 `git reset` 命令分解该提交，这样就撤销了这个提交，并创建两个独立的提交：

```
prompt> git reset HEAD^
contact.html: locally modified
prompt> git diff
diff --git a/contact.html b/contact.html
index 64135cb..c6cffa7 100644
--- a/contact.html
+++ b/contact.html
@@ -13,6 +13,14 @@
```

```

<p>
  <a href="mailto:tswicegood@gmail.com">Gmail</a>
</p>
+
+  <p>
+    <a href="http://twitter.com/tswicegood">Twitter</a>
+  </p>
+
+  <p>
+    <a href="http://www.travisswicegood.com/">Blog</a>
+  </p>
</body>
</html>

```

`git diff` 命令显示所有等待（暂存并）提交的变更。把 `contact.html` 文件中的第二个链接去掉，并且保存该文件。这时文件里仅有 Twitter 的链接。然后提交变更：

```

prompt> git commit -m "add link to Twitter" -a
Created commit 07950f4: add link to Twitter
1 files changed, 4 insertions(+), 0 deletions(-)

```

现在把刚才去掉的链接放回到 `contact.html` 文件里，并且为它创建一个新提交：

```

prompt> git commit -m "add link to blog" -a
Created commit 6c0eebe: add link to blog
1 files changed, 4 insertions(+), 0 deletions(-)

```

就这样，将一个提交分解成了两个提交。输入命令 `git rebase --continue` 继续：

```

prompt> git rebase --continue
Successfully rebased and updated refs/heads/master.

```

再快速检查一下日志，结果显示，原来压合在一起的两个提交现在已经重新分解成两个单独的提交了。

变基命令 `git rebase` 当然还有另一个功能，那才是变基的本意。变基本来是用于分支间的同步和移动，将在 9.3 节“分支变基”（第 118 页）中详细介绍。<sup>9</sup>

至此，我们学习了如何在 Git 中浏览甚至是改写历史。它们是 Git 中极其重要的功能，但却并不是每天都要使用的日常操作。至少在刚开始使用 Git 的时候是如此。

当你越来越熟练使用某些操作（比如变基）的时候，你会发现适合使用它的机会也越来越多。

到目前为止，我们只介绍了本地版本库中的操作。下一章将介绍远程版本库的相关操作，以便将自己的工作成果与大家共享，并获得别人的工作成果。

---

<sup>9</sup> 变基（Rebase）的本意是改变分支的起点，而保留分支上的所有修改。那为什么这里用变基操作来整理版本库的历史呢？想想 Git 实现变基的具体方法：基于某个版本作为起点，重新添加若干个提交。这一章我们用到的也是这个方法，只不过“Base”还是那个“Base”，不变了。当再次添加若干个提交时，通过交互方式，打乱了原先的次序或进行了其他的调整。——译者注

## 第 7 章

# 与远程版本库协作

Working with Remote Repositories

到目前为止，我们已经介绍过如何使用 Git 来完成各类本地操作。作为分布式 (*distributed*) 版本控制系统，它还支持项目成员间的协作。

本章介绍如何使用远程版本库相关的操作来协同工作。这将会涉及以下内容：

- 远程版本库的几种类型。
- 如何克隆远程版本库。
- 如何保持与远程版本库同步。
- 如何把本地更新推入到远程版本库。
- 如何对远程分支进行操作。
- 如何添加新的远程版本库。

本章将详细介绍前面使用过的版本库克隆操作，见 7.2 节“克隆远程版本库”（第 94 页）。而在此之前，我们先讨论连接远程版本库的几种网络协议。

### 7.1 网络协议

Network Protocols

Git 与网络上的远程版本库通信。这里所说的网络可以是内部局域网、虚拟专用网 (VPN)，或者是因特网。Git 提供了三种与远程版本库通信的协议：<sup>1</sup>

- SSH
- git
- HTTP/HTTPS

<sup>1</sup> 由于本章译注较多较长，统一放在本章正文之后（第 99, 100 页）。一编者注此译注请见第 99 页“1”。



图 7.1 SSH URL 组成结构

## SSH 协议

通过 SSH 协议来访问远程版本库，与直接通过文件系统访问非常类似。唯一不同的是，在指定远程版本库路径前，还必须为它指定域名服务器和用户名。

如图 7.1 所示，SSH 的 URL 和文件系统的 URL 唯一不同的是，SSH URL 多了 `git@github.com` 这部分。这部分告诉 Git，通过 SSH 协议，使用用户名 `git` 登录到 `github.com` 服务器上，并且克隆路径是 `tswicegood/mysite-chp6.git` 的版本库。

指定用户名不是必须的（上面例子中的用户名是 `git@`）。如果使用本机登录名作为用户名，在 URL 中就可以不用指定用户名。

很多情况下，项目中的多名成员共用一个用户名来登录到服务器。然后服务器会检测用户提供的证书，以确认用户拥有权限。

## git 自带网络协议

Git 也有自己的通信协议，其优势在于速度，在 Git 中它是最快的网络协议。但比较麻烦的是它须使用 9418 端口，而这个端口的通信一般会受到防火墙的限制。

如下页图 7.2 所示，Git URL 的 `git://` 指定是 git 协议，紧接着 `github.com` 是服务器名，再后面就是版本库的路径了。

在本例中，并没有指定版本库的完整路径。当配置 Git 服务器时，已指定服务器中具体目录来存放远程版本库。当客户端访问服务器上的版本库时，只须指定版本库的名称就可以了。

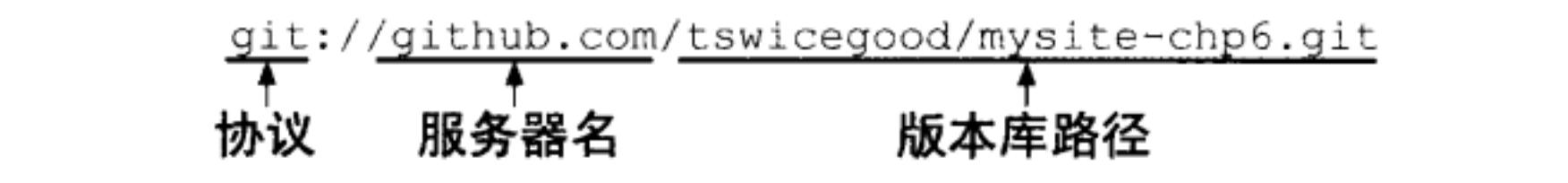


图 7.2 git 自带协议 URL 的组成

git 协议与 SSH 协议之间的主要区别是，git 协议无须加密，且是匿名的。当只希望开放远程版本库的只读权限时，git 协议是个不错的选择。但当想开放远程版本库的写权限时，用 git 协议就比较危险了——通常不希望所有的人都能匿名修改你的版本库吧。只要看到版本库路径中使用了 `git://`，那么在绝大多数情况下，该版本库是只读的。

一种常见的方式是，开发人员使用 git 协议将远程版本库中的更新拖入本地版本库，而使用 SSH 协议来将本地版本库中的更新推入远程版本库中。

## HTTP/HTTPS 协议

HTTP 协议一般是最后才考虑的协议。比起前面介绍的协议，在 Git 中使用 HTTP 协议通信效率最低。但是它的好处在于：能通过严格的防火墙，且易于架设。

继续使用服务器 `github.com` 上的版本库 `mysite-chp6.git` 做例子，如果使用 HTTP 协议来访问该远程版本库，则 URL 路径为：

`http://github.com/tswicegood/mysite-chp6.git`

然而，GitHub 服务器并不支持 HTTP 协议，因此，如果通过它克隆远程版本库，Git 就会提示错误。

## 如何选择网络通信协议

从 git 自带网络协议、SSH 及 HTTP/HTTPS 中选取一种或多种，搭配起来方案就很多。而如果只选取其中一种协议，则：

- 如果希望速度最快，选择 git 协议。
- 如果安全第一，SSH 协议是最好的选择。
- 如果不想更改防火墙限制规则，HTTP 或 HTTPS 协议是唯一的选择。

访问权限也是个问题。git 协议是匿名访问的，所以一旦开放远程版本库的写权限，任何人都可以对该版本库进行写入操作。而 SSH 协议只能是具有相应权限的用户才可以对远程版本库进行操作。对 HTTP/HTTPS 协议，须要架设 WebDAV 服务。

并不是只能单独选用某一个协议；可以混合使用这些协议，以兼顾速度和安全性。

我个人的 Git 版本库中运用了 SSH 协议和 git 协议组合。通过 SSH 协议，授权用户可使用 SSH 密钥对远程服务器进行写入操作，而匿名用户可通过 git 协议来进行只读操作。这种组合设置听起来很复杂，但使用工具 Gitosis 后就变得简单多了。第 11 章“使用 Gitosis 管理 Git 服务器”（第 143 页）将具体介绍 Gitosis。

## 7.2 克隆远程版本库

### *Cloning a Remote Repository*

使用远程版本库，可与其他人共享工作成果。对远程版本库，最简单的操作就是克隆一个现有的远程版本库。克隆操作就是创建远程版本库的本地拷贝。

加入一个正在开发的项目，克隆是一个常见的方法，但不是唯一的方式。如果是自己开始开发一个项目，然后再与他人分享，则可在必要时再设置远程版本库。详细操作请参考第 98 页的提示栏。

通过克隆操作生成的本地版本库，与通过 `git init` 命令创建的本地版本库很相像。唯一不同的是，克隆操作能够得到远程版本库中所有的内容，包括历史记录（截止到克隆操作发生时）。

可以随时使用 `git clone` 命令来克隆远程版本库。在该命令最简单的形式中，只须给出“远程版本库名称”一个参数。

相信到目前为止，你已经很熟悉本书第 2 篇每章开头部分都涉及的 `git clone` 命令。下面再次使用该命令，以获取完成上一章实践后的版本库：<sup>2</sup>

```
prompt> git clone git://github.com/twicelgood/mysite-chp6.git
Initialized empty Git repository in /work/mysite-chp6/.git/
remote: Counting objects: 37, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 37 (delta 10), reused 0 (delta 0)
Receiving objects: 100% (37/37), 4.08 KiB, done.
Resolving deltas: 100% (10/10), done.
```

---

<sup>2</sup> 此译注请见第 99 页“2”。

这个命令把服务器上的版本库下载到本地 `mysite-chp6` 目录下。进入该目录可以看到有 5 个前面章节中创建的文件：

```
prompt> cd mysite-chp6
prompt> ls
about.html contact.html copy.txt hello.html original.txt
```

现在在本地有了一个远程版本库的完整克隆。它既可以跟踪本地的改动，又可以不断从远程服务器上取来别人的改动。

## 7.3 版本库同步

Keeping Up-to Date

克隆操作可以获得远程版本库中直到克隆时的全部历史。然而，在克隆远程版本库之后，项目里其他开发人员仍将不断在此远程版本库中添加新的内容。

这就须要不断取来（*fetch*）远程版本库的改动到本地版本库。使用 `git fetch` 命令来完成这样的操作。

取来操作能够更新本地版本库中的远程分支。<sup>3</sup>我们已经知道，用 `git branch` 命令可以查看本地分支列表。如果后面加上`-r` 参数，Git 就显示远程分支：

```
prompt> git branch -r
origin/HEAD
origin/master
```

可以像检出普通本地分支一样检出远程分支，但是不应修改远程分支上的内容。如果确实要修改，应该先从远程分支创建一个本地分支，然后再进行修改。

命令 `git fetch` 更新远程分支，但它不会把远程分支上的修改合并到本地分支上。使用命令 `git pull` 可以顺序完成两件事情：取来，然后合并。<sup>4</sup>

命令 `git pull` 需要两个参数，一个是远程版本库名称，另一个是须要拖入的远程分支名（无须在分支前指定版本库前缀 `origin/`）。<sup>5</sup>

关于远程分支名前面的前缀 `origin/` 表示远程版本库上的分支名称，用于区别本地分支名称。而 `origin` 是默认的远程版本库别名，即克隆时指定的远程版本库。

---

<sup>3</sup> 此译注请见第 99 页 “3”。

<sup>4</sup> 此译注请见第 100 页 “4”。

<sup>5</sup> 此译注请见第 100 页 “5”。

介绍了从远程版本库取来改动及拖入到本地分支的操作后，下面就来介绍如何将本地改动推入远程版本库。

## 7.4 推入改动

这一节将向你介绍如何将本地改动推入到远程版本库。

前面介绍了如何从上游版本库获得更新，而要与团队中其他成员保持同步协作，这才是一半。另一半是把本地改动推入到上游版本库中，从而实现共享。

正如 1.4 节“代码修改与文件同步”（第 6 页）中描述的那样，推入操作是把本地改动，也就是本地的提交，推入到另一个版本库中。传统版本控制系统中并没有这步操作，因为提交代码时直接提交到中央版本库中了。

调用不带参数的 `git push` 命令时，Git 会推入到默认版本库 `origin` 中，并把本地版本库中当前所在分支的变更推入到远程版本库<sup>6</sup>对应的分支上。<sup>7</sup>

Git 只会推入已检入的改动，而不会推入工作目录树中的变更和暂存区中的改动：

```
prompt> git push
Counting objects: 11, done.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 933 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
To /repos/mysite/.git
  5ef8232..d49d1e5 master -> master
```

可以在 `git push` 命令后添加参数`--dry-run` 来查看推入哪些提交。

与命令 `git pull` 一样，`git push` 命令也可以指定要推入的版本库。语法相同，都是 `git push <repository> <refspec>`。`<repository>` 可以是任意有效的版本库名称。它可以是 7.1 节“网络协议”（第 91 页）中讨论过的任何 URL，包括将在 7.5 节“添加新的远程版本库”（下一页）中介绍的新添加的远程版本库名。

---

<sup>6</sup> 这是默认的情况。当然你也可以指定其他分支进行推入。但是本书中只举这个常见的例子。

<sup>7</sup> 此译注请见第 100 页“7”。

### 在工作目录树中体现推入的改动

这个操作虽然不是很常见，但有些情况需要此操作：当要让他人（甚至是你自己）把改动从一个版本库推入到你的本地版本库中的时候。比如需要从我的一个版本库中的改动推入到另一个虚拟机的版本库上，以便在不同操作系统中测试代码的可靠性。

被推入本地版本库的改动——不论是被别人还是自己——都不会自动反映在这个版本库的工作目录树上。这是为了避免推入的改动与工作目录树中未提交的改动相冲突，引起覆盖问题。

运行 `git checkout HEAD` 命令，可以将版本库中的最新版本（包含所有推入的改动）带入工作目录树中，这样就有机会解决与工作目录树中本地改动之间的冲突。<sup>8</sup>解决冲突的方法详见 5.4 节“冲突处理”（第 64 页）。

参数`<refspec>`可以是任意的简单形式，可以是标签也可以是分支，或者是像 `HEAD` 这样的关键字。用此参数来指定要推入的分支和推入分支到哪个地方。例如，可以使用命令 `git push origin mybranch:master` 将本地分支 `mybranch` 上的提交推入远程版本库的主分支上。

到目前为止，克隆远程版本库并保持本地版本库与其同步的基本方法就介绍完了。接下来介绍最后一个话题：添加新版本库。

## 7.5 添加新的远程版本库

*Adding New Remote Repositories*

只要有相应权限，就可以跟任意远程版本库打交道，进行推入和拖入操作。运行这些命令时，使用版本库的全名是很麻烦的，特别是总跟同一远程版本库同步的时候。

在本地版本库中，远程版本库的别名默认为 `origin`，它是克隆远程版本库时自动生成的。（还可以为更多远程版本库设置别名。）比如，要是工作中你总是从 `Erin` 那里拖入改动。

<sup>8</sup> 此译注请见第 100 页“8”。

### 手动添加远程版本库别名

如果项目已纳入一个公共版本库的版本控制中，那么新加入的开发者可以克隆这个版本库来开始自己的开发工作。但是，如果最初开发者自己在本地用 `git init` 建立了版本库，那该如何才能把本地的改动推入到公共版本库中呢？

须要用本节将要介绍的 `git remote add` 命令，给远程版本库起个别名。可以把推入操作的默认目标库起名为 `origin`。

例如，若想给远程版本库 `git@example.com:/repos/pocus.git` 起个别名，那么命令如下：

```
prompt> git remote add origin git@example.com:/repos/pocus.git
prompt> git push origin master
... output from Git ...
```

如果只是做一次拖入操作，可以直接在 `git pull` 命令中使用版本库的全称，如下所示：

```
prompt> git pull git://ourcompany.com/dev-erin.git
```

然而，这适合做一次拖入操作。如果须要反复进行拖入操作，则须要使用简单的别名来代替远程版本库冗长的全称。可以使用命令 `git remote add <name> <repo-url>` 给远程版本库添加别名。以下命令就是给 `Erin` 的版本库添加别名：

```
prompt> git remote add erin git://ourcompany.com/dev-erin.git
```

现在，进行推入和拖入操作时就可以使用 `erin` 来替代远程版本库的全称了。因此，拖入命令可以像这样简单：

```
prompt> git pull erin HEAD
warning: no common commits
remote: Counting objects: 6318, done.
remote: Compressing objects: 100% (2114/2114), done.
remote: Total 6318 (delta 3899), reused 5680 (delta 3505)
Receiving objects: 100% (6318/6318), 1.03 MiB, done.
Resolving deltas: 100% (3899/3899), done.
From git://ourcompany.com/dev-erin.git
 * branch          HEAD      -> FETCH_HEAD
Merge made by recursive.
... a bunch of output showing the result of the pull ...
```

使用别名替换远程版本库全称是不是很简单！当然，并不一定要使用 `erin` 这个别名，也可以用任何其他的别名。但唯一的限制是所有的别名必须是独一无二的，即只能有一个 `erin` 和 `origin`。

可以通过在本地版本库中调用 `git remote add` 命令，以 `origin` 做别名来标识某个远程版本库。如果在本地先用 `git init` 命令创建版本库，后来又须要把改动送到一个远程版本库里，就可以用这个方法。

`git remote` 命令还有不少其他实用用法。例如，使用不带任何参数的 `git remote` 命令，可以查看本地创建的全部远程版本库别名；再如，使用 `git remote show <name>` 命令可以查看某个远程版本库的详细信息。

如果不再需要某个远程版本库的别名，或者想用这个别名称呼其他远程版本库，则可以使用 `git remote rm` 命令先删除该别名。

远程版本库相关的操作介绍完成。到目前为止，我们要学习的 Git 命令差不多都介绍完了。接下来将介绍 Git 中一些不太常用的命令。

---

1. 此外，Git 还支持“简单”的方式，比如本机文件系统上的相对或绝对路径，比如内部网上共享文件夹的路径。详见“clone”命令手册页中“URLs”部分。一译者注

2. 欲了解克隆命令背后的秘密，请查看 `.git/config` 文件。与源库中该文件相比，它多了一块 [remote "origin"]。这个地方给源库起了个别名叫“origin”（7.5 节具体介绍），以及设置了对该库“fetch”操作（下节具体介绍）时默认的参数。此外，它还多了一块 [branch "master"]，其中说，本库的主分支要追随“origin”库里的“`refs/heads/master`”分支。其实“`refs/heads/master`”就是主分支的全称。至于什么叫“追随”，下节具体介绍。这里只告诉你，Git 在克隆时，把源库里所有的分支都带到了本库，给它们加上前缀改了名字，标识为远程分支（下节具体介绍）。然后基于源库里的主分支（如果克隆的时候主分支是检出状态）所对应的本库中的远程分支，偷偷创建了本地的主分支。一译者注

3. 这里所说的远程分支，是指远程版本库中的分支在本地的复制品。比如源库的主分支在本库的复制品是“`origin/master`”，全称“`remotes/origin/master`”。使用“`git fetch`”命令，可以指定从哪个远程库里，拿哪些分支，复制到本地时，怎么加前缀改名字。如果没有指定参数的时候，Git 就跑到“`.git/config`”文件里去，查看当前检出分支在追随哪个库里的分支（要是查不到就用“`origin`”库）。接着就用“`.git/config`”文件里定义的那个库的“`fetch`”默认参数来操作，把这个库里的内容复制到本地的（一条或多条）远程分支上。更多内容请参见“2”、“4”及“`fetch`”命令手册页。一译者注

4. 这条本地分支就通过远程分支来追随远程版本库里的分支。追随的意思是，总是把远程分支上的改动，本质上是远程版本库里的改动，合并到这条本地分支。本地分支和它追随的远程版本库里的分支，名字常常相同，但也可以不同。一译者注
5. 当不指定这两个参数的时候，Git 就不指定任何参数调用“git fetch”。之后，再把当前检出分支所追随的远程分支上的内容合并到当前检出分支。更多内容请参见“2”、“3”、“4”及“pull”命令手册页。一译者注
7. 不带任何参数调用 git push 时，Git 先要确定跟哪个远程版本库打交道：如果现在的检出分支追随某个库里的分支，就是该库。如果现在的检出分支不追随任何库里的分支，则就是“origin”库。接着 Git 要确定跟哪些分支打交道：挨个考察本库中所有的分支，如果它追随上述远程版本库中与它同名的分支，那它就算一号。然后，Git 考察这些要推入的分支，如果它和它追随的远程版本库里的分支相比较，已经“分叉”了，那 Git 在推入的时候就得合并——一天知道怎么在远程版本库合并，所以这种情况下 Git 干脆就报错退出了。最后，如果都没问题，Git 就把本地分支上的改动推入到远程版本库相应的分支上，同时更新本地版本库中的远程分支。更多内容请参见“2”、“4”及“push”命令手册页。一译者注
8. 操作时请非常小心，仔细检查，以免冲掉了本地未提交的改动或推入的改动。事实上，如果一个版本库经常被用来接受推入，那么它最好不要有本地改动，最好没有本地工作目录树。为此，可以在 git init 命令中加上参数--bare。详见 init 命令手册页。一译者注



## 第8章

### 管理本地版本库

Organizing Your Repository

对任何版本控制工具来说，为用户保留项目的完整历史记录是其主要功能。用户可以快速找到软件的早期版本，并浏览文件历史记录，从而了解文件是如何演进到目前状态的。

但是，这些历史记录信息量非常庞大。为了确保用户能在自己创建的全部历史记录中找到合适的内容，制定一个合理的版本库管理策略非常重要。本章将介绍以下内容：

- 用标签标记项目里程碑。
- 在代码发布过程中处理好发布分支，以便集中精力做好开发。
- 使用类似于目录结构的组织方式对标签或分支分组。
- 用一个或多个版本库来跟踪多个项目。
- 用 Git 子模块功能跟踪外部版本库。

本章将继续使用前面各章中用到的版本库。如果你手头没有该版本库，可以从 GitHub 服务器上克隆：

```
prompt> git clone git://github.com/twicegood/mysite-chp7.git
Initialized empty Git repository in /work/mysite-chp7/.git/
remote: Counting objects: 53, done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 53 (delta 19), reused 0 (delta 0)
Receiving objects: 100% (53/53), 5.72 KiB, done.
Resolving deltas: 100% (19/19), done.
```

## 8.1 使用标签标记里程碑

### Marking Milestones with Tags

随着版本库的不断演进，将会产生一些可部署的里程碑，并在此基础上继续演进。使用标签可以很容易标记这些里程碑，以便于日后回溯。

版本库里的标签就像书签，使用它们可以方便回到版本库上打了标签的点。在 Git 中，可以随心所愿地给任意提交打标签。

标签最常用于给项目代码的发布版本做标识，以便以后在须要修正或功能变更时，可以通过标签回到该发布代码上。

Git 标签是只读的。如果使用过 Subversion，要注意 Git 和它的不同：在 Git 中不能像修改分支的内容一样修改标签内容。但这样才是正道，可以确保在任意时刻取出标签对应的代码，都与创建标签时的代码完全一致，毫无变化。

Git 中关于标签的命令是 `git tag`。该命令与 `git branch` 命令类似，不加参数地调用它，可以查看已存在标签的列表：

```
prompt> git tag  
1.0
```

这个标签是第 3 章“创建第一个项目”（第 25 页）中创建的标签。下面新建一个名为 1.1 的标签。再次调用命令 `git tag`，但这次加上标签名 1.1：

```
prompt> git tag 1.1
```

Git 并没有提示标签是否创建成功，但是可以再次调用不带参数的命令 `git tag` 看到刚刚创建的标签：

```
prompt> git tag  
1.0  
1.1
```

当 Git 标签命令运行成功时，它并不提供反馈信息，但是当有问题发生的时候，它会通知你。例如，当试图使用包含空格的无效名称创建标签时，`git tag` 将会返回错误提示。有效命名将在 8.3 节“标签与分支的有效名称”（第 106 页）中介绍。

```
prompt> git tag "version 1.1"  
fatal: 'version 1.1' is not a valid tag name.
```

如果只使用标签名称做参数调用命令 `git tag`, Git 将基于当前工作目录树中的提交（也就是检出分支的末端版本）创建标签。你也可以添加一个参数指定须要标记的提交。这个参数可以是任何有效的提交名称或分支名称。

例如，可以这样调用 `git tag` 命令，为 `contact` 分支上的最新提交创建标签：

```
prompt> git tag contacts/1.1 contacts
prompt> git tag
1.0
1.1
contacts/1.1
```

你可以使用标签返回标签标记的版本库状态。虽然不能改变标签所对应的代码，但是可以检出标签，就像检出分支一样：

```
prompt> git checkout 1.0
Note: moving to "1.0" 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 4b53779... Add in a description element to the metadata
```

这时，你跑到“三不管地带”去了：不在任何分支上，没法提交，因此不能跟踪改动。如果此时运行命令 `git branch` 查看当前的本地分支，Git 将会提示目前不在任何分支上：

```
prompt> git branch
* (no branch)
  about
  alternate
  contacts
  master
  new
```

你可以使用命令 `git checkout -b` 创建并检出一个新分支：

```
prompt> git checkout -b from-1.0
Switched to a new branch "from-1.0"
```

现在又可以跟踪代码改动了。正如 Git 的其他优点一样，也可以使用其他方法达到相同的目的。你可以将标签作为第二参数调用命令 `git branch` 或 `git checkout -b` 创建新分支：

```
prompt> git checkout -b another-from-1.0 1.0
Switched to a new branch "another-from-1.0"
```

这时使用命令 `git log` 可以看到，当前分支上只有标签 1.0 中的三个提交。

如果要在已经发布的代码上修正 Bug 或做一些小改动，基于标签创建分支将非常方便。下一节将介绍如何处理发布分支。

## 8.2 发布分支的处理

*Handling Release Branches*

发布分支是指即将要发布代码的地方。开发团队一般用它来隔离即将要发布的代码。隔离（*segregate*）的具体含义要视具体情况而定。

至于什么时候使用发布分支取决于开发团队及其开发模式。为了简单起见，这里暂且这样定义：发布分支是指当一个项目的所有功能都已开发完成，但尚未达到发布的质量标准时创建的分支。

发布分支通常只须要少许改动，往往侧重于 Bug 或逻辑修正，很少会添加新功能。有了发布分支，在主分支上继续开发新功能就会很方便，因为发布分支上包含了须要发布的代码，而且不会将新开发的代码包含进来。

发布分支通常以 RB\_作为前缀并包含版本号，比如 1.2 版发布分支可表示为 RB\_1.2，而 1.3 版的发布分支则表示为 RB\_1.3。

发布分支持续时间不长，通常只用于发布代码的最终测试期间。一旦该版本发布，应该使用标签标记项目，然后就可以删除该发布分支了。

不用担心删除分支会造成什么不良影响，因为不用为了保留历史记录而在版本库中保留分支。标签已经标记了发布时的版本（以及从始至此的代码演进轨迹），再保留发布分支，反而有害无益：会把分支列表弄得杂乱无章。

那么，如何修正发布版本中出现的 Bug？可以像上一节介绍的那样，基于标签创建一个新的分支，使用 RB\_做前缀，并指定相应的标签作为分支的基点。

```
prompt> git branch RB_1.0.1 1.0
prompt> git checkout RB_1.0.1
Switched to branch "RB_1.0.1"
```

只要将标签名称作为命令 `git branch` 的第二参数，Git 将会基于该标签所指定的提交创建新分支。

新分支就像是创建标签时的发布分支，它的历史记录与被删除的发布分支的历史记录完全一样。

你可以使用 `git log` 查看：

```
prompt> git log --pretty=format:"%h %s"
4b53779 Add in a description element to the metadata
a5dacab add <head> and <title> to index
7b1558c add in hello world HTML
```

接下来可以修正 Bug 了，当修正完成时，创建一个新的标签：

```
prompt> git tag 1.0.1
```

一旦 Bug 修正结束并打好标签，则采取删除原发布分支一样的方法删除该分支。但是要先回到主分支，因为 Git 不能删除当前检出的分支：

```
prompt> git checkout master
Switched to branch "master"
prompt> git branch -D RB_1.0.1
Deleted branch RB_1.0.1.
```

注意，这里使用的命令是 `git branch -D`，因为该分支没有合并回主分支。我们先这么做，等到 9.3 节“分支变基”（第 118 页）中再深入研究。

发布分支有助于在临近发布前组织团队工作流程，可以将一个即将发布的版本与新功能开发及因此带来的 Bug 相隔离。以确保只有必要的 Bug 修正、客户的最终反馈等，集成到即将发布的版本中去。

这就是 Git 中发布分支的工作原理。难点在于决定什么时候创建发布分支，这取决于团队的开发模式。如果你的团队在不停地添加新功能，最好一旦产品功能开发完毕就立刻创建发布分支；而如果开发流程较慢，则在发布前几天再创建发布分支也不迟。

在 5.3 节“合并分支间的修改”（第 59 页），谈到过在 Git 中，分支合并是容易操作的。因此，在 Git 中早些创建分支并保持同步，并不是件难事。如果认为应该早点创建分支，就应该在拉出分支的提交上创建分支，不必故意拖延。

发布分支和标签可以帮助你组织版本库。唯一要注意的是如何命名这些标签和分支。Git 在允许的范围内比较灵活，但是也有限度。接下来详细介绍。

## 8.3 标签与分支的有效名称

### Using Valid Names for Tags and Branches

在介绍如何在一些特定场景中使用 Git 之前，让我们来看看标签和分支的有效名称。Git 比较自由，但是有些事情应该避免。

首先，虽然可以在标签或分支名中使用反斜杠 “/”，但是不能以反斜杠作为结尾。这样做是为了让用户可以像组织文件目录结构一样组织标签和分支。

在标签名称和分支名称中也可以使用圆点 “.”，但是不能将圆点置于各段名称的开头。例如 `release/1.0` 是合法的标签名，而 `releases/.1.0` 和 `.releases/1.0` 都是无效的。

熟悉 Linux 或“BSD/Mac OS X”文件系统的读者，可能会意识到为何不能这么做：这些操作系统把以圆点开头的文件或目录看做是隐藏项。

你也不能在标签名称或分支名称中使用特殊字符。包括空格、波浪号 “~”、脱字符 “^”、冒号 “:”、问号 “?”、星号 “\*”、方括号 “[” 等，以及所有编码小于 “\040”的 ASCII 控制字符和删除键（ASCII 码 “\177”）均不允许使用。如果你不知道哪些是限制字符，也不明白这些特殊字符从何而来，就不用担心了，因为这也许意味着你永远也不会因为意外而出错的。<sup>1</sup>

命名的另外一条限制是在名称的任何地方均不能出现双圆点 “..”。回忆 6.2 节“指定查找范围”（第 73 页）中表明区间的语法，该区间使用“<第一个提交>..<第二个提交>”来表示。因此为了避免混淆，在标签名和分支名中不允许使用双圆点。

<sup>1</sup> 这条译注写给不了解这些特殊字符，又比较好奇的读者：控制字符是用来实现特定操作的，比如 ASCII 为 7 的字符，BEL，当程序在遇到这个字符时，会触发系统发出 BEEP 一声。

还有一些比较常用的控制字符，比如 ESC、BACKSPACE（删除上一个字符）、Del（删除当前字符）、回车符、换行符等。一译者注

总而言之，任何有效的文件或目录名都可以作为标签或分支名称。也正如文件系统一样，可以像文件、目录结构一样管理标签和分支。下面将介绍一种特殊的标签。

## 8.4 记录和跟踪多个项目

Tracking Multiple Projects

大多数公司都有多个项目。即便是一个产品，也有可能被分割成多个项目，便于团队成员不必翻拣所有的代码，就能找到他们需要的那部分。

在 Git 中有多种方法来管理多个项目。比如，可以将所有项目放在同一个版本库中，也可以为每个项目建立单独的版本库。这两种方法各有利弊，接下来将做详细介绍。

### 多个项目共享一个版本库

第一个方法较为直接，就是多个项目共享同一个版本库。你可以在版本库的一级目录下为每个项目分别创建不同的目录。

用过 Subversion 的用户可能对此较为熟悉，因为这正是 Subversion 在同一个版本库中管理多个项目的方式。当要访问所有项目的时候，只须克隆一个版本库即可，这一点很方便。

当处理的项目需要有统一的历史记录时，这种方法很适用。例如一个由多个组件构成的项目，且这些组件总是同时发布。

例如，内容管理系统、订单录入系统，或者其他类型的系统，都由许多相互独立的组件构成，但它们都被打包在一起同时发布。

请注意，这里强调“同时发布”。当决定一个版本库是包括一个项目还是多个项目时，这是最关键的因素。在 8.4 节“多项目多版本库”（下一页）中，将讨论对多个项目使用多个版本库。

当每一个小项目，或者组件，总是作为整个大项目的一部分来发布时，使用同一个版本库共享历史应该是不错的选择。

这确保版本库中所有的历史记录都围绕同一个大项目。

当这些小项目或组件的发布工作相互独立时，它们可能需要单独的历史记录。这是因为 Git 处理标签和分支方式：总是以整个版本库为单位，而不是对版本库的一部分打标签或拉分支。

如果每个小项目都有自己独立的发布时间表，则须要创建的分支和标签数量将会呈指数增长。好在 Git 的版本库是轻量级的，因此每个小项目创建自己的版本库并不难。

## 多项目多版本库

与多个项目共享同一个版本库相比，还有一种方式是为每一个项目创建一个版本库。这需要多做一些初始化和设置工作，但可以让每个项目都有单独的历史记录。

决定什么情况下应该将项目分割开来并不难：与多个项目共享同一个版本库的规则相反。如果各小项目是单独发布的，它们就应该有自己独立的版本库。

但是规则并不是一成不变的。如果两个项目须要单独发布，但是它们之间关系紧密，不能单独使用，那么可把它们放在同一个版本库中会更为方便，这样只须克隆一次就能得到两个项目。

是的，在 Git 中总是存在着对同一问题的不同解决方法，适应不同的情况。让我们进入下一个话题：Git 子模块。

## 8.5 使用 Git 子模块跟踪外部版本库

Using Git Submodules to Track External Repositories

有时候，须要同时跟踪多个版本库，因为它们关系密切，就好像在一个版本库中一样。这可能是因为项目要依赖的第三方软件库，或者为了管理方便，项目被分割成了多个子项目。

通过 Git 子模块 (*sub-module*) 可以跟踪外部版本库。它允许在某一个版本库中再存储另一个版本库，并且能够保持两个版本库之间的完全独立。熟悉 Subversion 的用户会意识到，它和 `svn:externals` 差不多。

## 添加新子模块

下面演示这个功能。先创建一个新的版本库以容纳子模块版本库。这里创建一个名叫 `magic` 的版本库：

```
prompt> mkdir /work/magic
prompt> cd /work/magic
prompt> git init
Initialized empty Git repository in /work/magic/.git/
```

在这个刚生成的空版本库中，使用命令 `git submodule` 来查看该版本库中的子模块：

```
prompt> git submodule
prompt>
```

因为目前还没有定义任何子模块，Git 没有返回任何信息。添加一个新的子模块很简单，使用 `git submodule add` 命令。有两个必须的参数，第一个是源版本库；第二个是要存储该版本库的路径。

这里先添加一个叫 `hocus` 的子模块版本库，这是我在 GitHub 上为本书创建的一个小版本库。为了明晰起见，放在一个叫 `hocus` 的目录中。当练习的时候可以将下面的命令放在一行，并忽略 “\”，这是为了适合本书页面宽度而添加的。

```
prompt> git submodule add \
          git://github.com/tswicegood/hocus.git \
          hocus
Initialized empty Git repository in /work/magic/hocus/.git/
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 7 (delta 0), reused 7 (delta 0)
Receiving objects: 100% (7/7), done.
```

现在输入命令 `git submodule` 再看看，就会显示出 `hocus`。

```
prompt> git submodule
-20cc9ddc65b5f3ea3b871480c1e6d8085db48457 hocus
```

Git 子模块跟踪了远程版本库的一个特定版本，以一个哈希值来标明，最后显示的是你所起的名字。注意哈希值前面的减号，它表明 `hocus` 子模块还没有被初始化。要初始化它，很简单：

```
prompt> git submodule init hocus
Submodule 'hocus' (git://github.com/tswicegood/hocus.git)
registered for path 'hocus'
```

可在`.git/config` 中添加一项，使 Git 知道 `hocus` 目录是一个子模块。现在所有的内容已经添加并且进行了初始化，可以看看版本库发生了什么变化：

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

其中`.gitmodules` 是一个存储用户所有模块信息的纯文本文件。Git 版本库记录和跟踪这个文件。当别人共享你的版本库时，Git 就可以根据这个文件中的信息来部署它们的子模块。

现在可以提交这些改动了，添加简单的提交留言：

```
prompt> git commit -m "initial commit with submodule"
Created initial commit f24a0d9: initial commit with submodule
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 hocus
```

现在新的子模块可以和其他版本库共存了。克隆包含子模块的版本库稍微有点不同，下面我们试试。

## 克隆含子模块的版本库

在克隆版本库之后，需要一些额外的步骤，来设置好子模块。下面先克隆：

```
prompt> cd /work
prompt> git clone magic new-magic
Initialized empty Git repository in /work/new-magic/.git/
prompt> cd new-magic
prompt> ls
hocus
```

你可以看到 `hocus` 目录了，但进入该目录后，会发现这是个空文件夹。运行命令 `git submodule` 会发现它还没有初始化：

```
prompt> git submodule
-20cc9ddc65b5f3ea3b871480c1e6d8085db48457 hocus
```

记得在上一节中以减号开头的哈希值表明该子模块没有初始化。运行 `git submodule init` 命令进行初始化：

```
prompt> git submodule init hocus
Submodule 'hocus' (git://github.com/tswicegood/hocus.git)
registered for path 'hocus'
```

但是 `hocus` 目录仍然是空的，还须要运行命令 `git submodule update` 把内容从源版本库拿过来。正如其他的 `git submodule` 命令一样，以子模块的名称做参数：

```
prompt> git submodule update hocus
Initialized empty Git repository in /work/new-magic/hocus/.git/
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 7 (delta 0), reused 7 (delta 0)
Receiving objects: 100% (7/7), done.
Submodule path 'hocus': checked out
'20cc9ddc65b5f3ea3b871480c1e6d8085db48457'
```

`hocus` 目录中包含了提交名称为 `20cc9dd` 的版本所对应的全部文件。现在已经介绍了如何快速部署子模块，但是目前它只反映子模块提交所对应的版本的内容，如何才能切换到自己想用的版本呢？这是下面要介绍的内容。

## 改变子模块的版本

Git 子模块并不会自动引用版本库中的最新提交；子模块只反映某个提交对应的版本。当初添加子模块的时候，Git 总是用源库当时的 HEAD，也就是源库中当时检出分支的末梢对应的版本。当须要切换到其他版本的时候，还须要做一些操作。

对于习惯了 Subversion 的用户，这乍看起来比较奇怪。Subversion 跟踪一个版本库，而且，当用户每次执行更新操作的时候，它都会自动拿来最新的提交。当须要引用某个特定的版本时，反而须要进行特殊的设置。

Subversion 方式看似比较方便，但是可能会带来问题（因为引用的内容变得不稳定了，处于不断变化中）。比如，在引用的版本库中，产生了新的缺陷，该怎么办呢？不同的团队成员因为更新自己的版本库的时间不同，而引用了子模块的不同版本，却不易察觉，这样的问题怎么解决呢？明确引用某一个提交版本，就可以避免这些问题。

当用户第一次创建子模块时，Git 记录下所引用的提交；之后它和该版本库间，不会有“偷偷的”通信和同步。事实上，子模块是对检出某个特定提交的版本

库的完整克隆。通过切换到 `hocus` 目录，查看分支列表，就可以看到这一点：

```
prompt> cd hocus/
prompt> git branch
* (no branch)
  master
```

这个版本库只有两个提交，我们从第二次提交切换到第一次提交。检出 HEAD 的前一个提交：

```
prompt> git checkout HEAD^
Previous HEAD position was 20cc9dd... initial commit
HEAD is now at 7901f67... initial commit with README
```

现在应该告诉 Git，当前这个版本正是我们想要的。调用无参数命令 `git submodule` 可以看出，Git 已经检测到了改动：

```
prompt> cd ..
prompt> git submodule
+7901f67feaadeeef755734a92febbc7214fb7871 hocus (7901f67)
```

+表示它不是“容器”版本库所记录的子模块应该在的版本。使用命令 `git status` 也显示出 `hocus` 目录被改动了：

```
prompt> git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified: hocus
#
no changes added to commit (use "git add" and/or "git commit -a")
```

现在要做的工作是添加 `hocus` 目录到暂存区，并提交。表明你希望 Git 引用子模块这个新的版本。

```
prompt> git add hocus
prompt> git commit -m 'update commit to track in submodule'
Created commit fedf2bc: update commit to track in submodule
 1 files changed, 1 insertions(+), 1 deletions(-)
```

查看这个提交的改动，可以看到 Git 所记录的内容。当改变子模块被引用的版本时要格外小心。当使用子模块时，有些细节值得注意。

## 使用子模块时要提防的错误

调用命令 `git add` 的时候，确保结尾处没有斜杠 “\”。Git 目前把结尾的斜杠看做是另外一个请求：将源版本库中的全部文件添加到当前版本库，而不是更新子模块引用的版本。如果犯了这种错误，则在提交前使用命令 `git status` 可以看到。

另一件要注意的事情是，命令 `git submodule update` 是“毁灭性”的，它会覆盖本地子模块中所有没有提交的内容。因此在使用 `git submodule update` 之前，请仔细检查本地子模块的工作目录树，以确保本地该提交的修改已提交到版本库。

另外一个错误发生于直接添加新内容到本地子模块版本库时。要记住，在改动之前，要（在子模块中）检出你想使用的分支，以便将新修改提交到该分支。子模块和“容器”模块里的普通分支是分开的。（在子模块中）调用命令 `git checkout` 很容易就能检出分支。

接下来，一旦（在子模块中）提交修改，必须确保这些改动被送回（子模块的）远程版本库中。这样才能（在本地“容器”版本库中）改变引用的子模块版本。否则，当运行 `git submodule update` 的时候，Git 会因为在（子模块的）远程版本库中找不到对应的版本，而产生不可预期的运行结果。

到此，本章就结束了。总的说来，本章为初学者介绍了如何使用标签，以及如何利用它们来管理版本库中的里程碑；同时也介绍了发布分支，以及在代码发布时如何利用分支来组织版本库结构。

此外，本章还介绍了两种方法来管理多个版本库和多个项目：把所有的项目放在一个大的版本库中，或者为每一个项目建立独立的版本库。在本章结尾，还介绍了 Git 子模块，以及如何在本地版本库中，利用子模块作为“容器”版本库中独立组成部分来引用远程版本库。

通过对本章及前几章的学习，相信你已掌握了如何利用 Git 来管理产品开发的知识，并且逐渐上手。下一章将介绍几个 Git 中不太常用，但有助于拓宽知识视野的功能。



# 第9章

## 高阶功能

Beyond the Basics

Git 是一个很丰富的软件工具包。到目前为止，所涉及的都是 Git 平时最常用的命令。这些只是 Git 的皮毛而已。

Git 目前覆盖了 140 个以上的命令。其中大部分几乎用不到，例如 Git 内部命令 `git check-reformat`，是用来确定一个字符串是否为有效的分支或标签名称的，扩展 Git 功能的脚本或程序可能会用到它。虽然有些命令不经常使用，一旦需要时却又是非常有用的。本章主要介绍这些方面，包括以下内容：

- 压缩版本库。
- 导出版本库。
- 分支变基到新的基点。
- 重现版本库中隐藏的历史。
- 使用二分法在版本库中查找引入 Bug 的时间点。

本章将继续使用贯穿本书的 `mysite` 版本库。如果没有该版本库，可以使用如下命令克隆相应的版本库：

```
prompt> git clone git://github.com/twicelgood/mysite-chp8.git
Initialized empty Git repository in /work/mysite-chp8/.git/
remote: Counting objects: 56, done.
remote: Compressing objects: 100% (49/49), done.
remote: Total 56 (delta 21), reused 0 (delta 0)
Receiving objects: 100% (56/56), 5.92 KiB, done.
Resolving deltas: 100% (21/21), done.
```

## 9.1 压缩版本库

### Compacting Repository History

生活中的每样东西都要适当维护以确保最优运行状态。汽车要换润滑油，地板要打扫，在 Git 中要使用 `git gc` 命令。

Git 版本库里存储了所有的东西，由此带来的问题就是偶尔会留下一些没有用的数据。例如，当人们用 `git commit` 加上`--amend` 参数增补一个提交时，Git 也会记住原来的版本；或者用命令 `git branch -D` 删掉一个试验分支时，Git 会保留该分支上的内容，虽然已经没有跟它相关的分支或标签了。

这就是为什么用 `git gc` 的原因。使用 `git gc` 整理版本库以优化 Git 内部存储历史记录。这种方式挺好用，一个月清理一次，或者每大约 100 个提交清理一次即可。它并不改变历史记录，只改变历史记录的存储方式。

在空版本库中运行 `git gc` 命令不会有什么影响，但是也不会有任何改善。为了给大家一个有意义的例子，这里将使用本书写作所使用的版本库来做演示。

自写作本篇以来，已经好几个星期没有为本书的版本库运行过 `git gc` 命令了。在这几个星期里，已经根据 Beta 版读者的反馈加了 50 个左右的修改。下面是针对本书版本库运行 `git gc` 后的输出结果：

```
prompt> git gc
Counting objects: 3918, done.
Compressing objects: 100% (2052/2052), done.
Writing objects: 100% (3918/3918), done.
Total 3918 (delta 2103), reused 3440 (delta 1852)
Removing duplicate objects: 100% (256/256), done.
```

当然，运行 `git gc` 的输出结果会因每个人的版本库历史的长短而不同。`git gc` 命令的运行的效果取决于版本库当时的实际情况。一旦运行 `git gc` 一次，在版本库有若干新的改动之前都不必再运行它。

因此，版本库优化意味着什么呢？磁盘空间。在版本库里通过运行 `git gc` 节省了大约 20% 的磁盘空间。

这可不是说 Git 日常运行时浪费了磁盘空间。事实上，Git 为了照顾到快速和节约两个目标，它把一些须要费时完成的优化工作，推迟到用户有时间做的时候再做。

在 `git gc` 命令中使用参数`--aggressive`，可使版本库得到进一步优化。这会增加时间，但很值得尝试。

Git 在增量存储单元 (*delta*) 中存储修改。一般情况下，`git gc` 命令运行时会压缩这些增量存储单元，但是不会重新计算它们。如果使用`--aggressive` 参数，则 Git 会重新计算它们。

## 9.2 导出版本库

### Exporting Your Repository

在项目成功完成后，须要发布该软件。如果软件是用现代脚本语言写出来的，例如 Python、Ruby 或 PHP，还要向最终用户提供源代码。

为此可以向大家提供该项目的 Git 公共版本库的访问权限，但是，如果创建一个包含源代码正确版本的发布包，最终用户就无须为使用该软件而先去学习 Git 了。

使用命令 `git archive` 可以方便创建一个版本快照。它可以把代码以 tar 或 zip 的格式导出，也即创建一个版本库中某一版本的拷贝。

命令 `git archive` 有好几个参数。首先，要指定`--format=<格式类型>`参数，以说明想要转换的格式。有效的格式类型是 `tar` 和 `zip`。

其次，要指定一个版本。这个版本可以使用一个提交、一条分支或一个标签的名字来标识。以上两个是必须提供的参数。另外一个有用的参数是`--prefix`，该参数可以指定一个目录名作为父目录，把要发布的代码文件都放入这个目录里。

例如，可在版本库的工作目录树下运行下述命令，从我们一直在使用的 `mysite` 版本库生成压缩包：

```
prompt> git archive --format=zip \
    --prefix=mysite-release/ \
    HEAD > mysite-release.zip
```

第一行包括前面说过的`--format` 参数。第二行是`--prefix` 参数。注意在 `mysite-release` 后面有个斜杠。没有斜杠，`git archive` 会把 `mysite-release` 加

在每个文件的前面，而不是当作父目录。

最后一行由两部分组成。首先，指定版本为 HEAD 对应的版本，随后，用“>”告诉 shell 把 `git archive` 的输出放在 `mysite-release.zip` 文件里。

用类似的方法可以从版本库创建 tar 包，这时须要提供一个额外的命令而不是直接将 `git archive` 输出到文件上。运行以下命令生成 `mysite-release.tar.gz` 文件：

```
prompt> git archive --format=tar \
    --prefix=mysite-release/ \
    HEAD | gzip > mysite-release.tar.gz
```

请注意在 HEAD 后面，用管道（|）传递相应内容到 `gzip` 命令，然后生成 `mysite-release.tar.gz` 文件。如果想使用 `bzip2` 来压缩，可将前面命令中的 `gzip` 换成 `bzip2` 即可。

把代码从版本库中导出为 tar 包或压缩文件的这种方法，能为你带来很多方便，因为很多用户不需要（也不想要）版本库中的历史记录。

## 9.3 分支变基

Rebasing a Branch

使用分支是软件配置管理的重要方法，但是要在各分支之间保持同步却成了分支的“阿喀琉斯之踵”<sup>1</sup>。使用 Git 提供的合并跟踪功能，可大大降低各分支间同步的难度。但是在 Git 中还有另外一种方法。

例如，在 8.2 节“发布分支的处理”（第 104 页）中，在 `RB_1.0.1` 分支上添加了代码，该代码只存在于 `1.0.1` 标签上，而主分支上没有。

现在，可以把主分支变基到这个标签，以便相应的改动也出现在主分支上，成为主分支历史的一部分。

前面 6.7 节“重新改写历史记录”（第 85 页）中介绍过变基操作，在该节中介绍了使用 `git rebase -i` 交互式命令模式来重新改写历史记录，而这里可以使用不带-i 参数的 `git rebase` 命令，让当前分支变基到另一条分支上的某个新版本。你

---

<sup>1</sup> 指致命弱点，阿喀琉斯是希腊神话中的人物。——译者注

可以把这种方式看作是基于一个新的基点，重演分支上发生过的改动。

现在运行 `git rebase`，以 1.0.1 标签作为新的基点：

```
prompt> git rebase 1.0.1
First, rewinding head to replay your work on top of it...
Applying: add in a bio link
error: patch failed: index.html:5
error: index.html: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged index.html
CONFLICT (content): Merge conflict in index.html
Failed to merge in the changes.
Patch failed at 0001.
```

When you have resolved this problem run "git rebase --continue".  
If you would prefer to skip this patch, instead run "git rebase --skip".  
To restore the original branch and stop rebasing run "git rebase  
--abort".

哦，报了一个错误。还记得吗，在 5.4 节“冲突处理”（第 64 页）中介绍过如何解决冲突。`git mergetool` 是处理冲突的常规方法，但是这里手动编辑处理冲突更方便。

这是因为这里的改动没有对错之分，既想要 1.0.1 版的新增段落，也想要 1.0 版之后加上的列表。

手动编辑是最容易的方法。`index.html` 文件的冲突如下所示：

```
<<<<< HEAD:index.html
<p>
  Fixed
</p>
=====
<ul>
  <li><a href="bio.html">Biography</a></li>
</ul>
>>>>> add in a bio link:index.html
```

删掉那三行冲突标志，可使文件内容看起来和正常文件一样。然后保存文件并运行以下命令告诉 Git，冲突已经解决，可以继续了：

```
prompt> git add index.html
```

```
prompt> git rebase --continue
Applying: add in a bio link
... a bunch of additional lines saying Applying: ...
Applying: copy original to show cross-file blame
```

还记得在 4.4 节“管理文件”（第 51 页）中使用命令 `git mv` 将 `index.html` 改名为 `hello.html` 吗？现在检查 `hello.html` 文件，它包括了 1.0.1 版中新增的段落。

这是一个非常简单的例子。它只包括最初的一点冲突，而且这个冲突也很好理解。下面来看几个复杂一点的例子。

假定在 Git 版本库中，使用自己喜欢的语言或框架来开发项目。同时，为了一些定制性的改动，你须要同时在一条维护分支上工作。这时变基可能发生以下几种情况。

首先，对于欲变基的分支上的提交，如果它已经存在于两条分支的共同历史中，则 Git 会意识到这一点，而不会在变基时再次添加此提交。

另外一种可能性就是冲突。处理冲突的流程与我们在刚才的简单例子中演示的一样：先解决冲突，然后调用命令 `git add` 和 `git rebase --continue`。

当然，你也可以调用命令 `git rebase` 时使用参数`--skip` 或`--abort` 来跳过特定提交或完全放弃变基。

参数`--onto` 提供了另外一种有趣的重写历史记录方式。例如，有三条分支：主分支、从主分支拉出来的 `contacts` 分支和从 `contacts` 分支拉出来的 `search` 分支。

这三条分支的历史演进可能类似于这样：先是编写通讯录（即“`contacts`”）相关的代码，并在中途又决定添加关于搜索（即“`search`”）的代码。

当完成 `search` 分支上的代码时，突然意识到不需要任何在 `contacts` 分支上完成的改动，搜索功能就可以运行。

这时就须要用到`--onto` 参数，它需要一个参数，即，希望变基到的分支名称。例如，将 `search` 分支变基主分支，命令如下所示：

```
prompt> git rebase --onto master contacts search
```

这个命令是将 `search` 分支从 `contacts` 分支上脱离，移动到主分支上。如果要合并 `search` 分支上的内容到主分支上，但不需要 `contacts` 分支上的任何东西，可以使用此方法。当然，`search` 分支要完全独立于 `contacts` 分支，尽量避免变基到主分支时出现合并冲突。

还可以同时使用`--onto`参数和提交范围参数，来做其他一些有趣的事情。提交范围在 6.2 节“指定查找范围”（第 73 页）中介绍过。例如可以用 `git rebase` 来抹掉一个版本。为了抹掉倒数第二个提交，可以这样做：

```
prompt> git rebase --onto HEAD^^ HEAD^ HEAD
```

该命令将倒数第二个提交之后的部分（也就是倒数第一个提交）变基到倒数第三个提交上。

当然，无论什么时候，要重新改写分支历史记录，为了防止出错，可以先做些备份。你也可以在正式操作之前，先在试验性分支上试一试。由于在 Git 中很容易创建和管理分支，所以在实施变基之前做做试验并不难。如果出错，还可以使用另外一个方便的工具：重现（reflog）。

## 9.4 重现隐藏的历史

*Using the Reflog*

Linus 为 Git 设定的其中一个重要原则是它一定要安全，这意味着可以回到版本库历史中的任何时刻。为了做到这一点，需要有一些机制可以记录和跟踪历史上的改动。

在通常的版本控制系统中，除了极端情况，版本库中的历史记录足以提供安全保障。然而，Git 允许用户做一些大多数版本控制系统不能做的事，比如改变提交的顺序，将一个提交分成好几个提交，删除提交，添加额外的提交，这些都是其他版本控制系统如 Subversion 或 CVS 做不到的。

伴随这些能力，风险随之而来。如果无意中用 `git branch -D` 删除了一条分支，或者用 `git rebase -i` 重写历史记录时出错，该怎么办？<sup>2</sup>这时就要用到 Git 中的重现（reflog）了。

---

<sup>2</sup> 笔者本人第一次进行交互式变基操作时，就遇到了这样的问题。大约 50 个提交在 5 秒钟内消失了。直到笔者学会重现操作时，它们才重新出现。

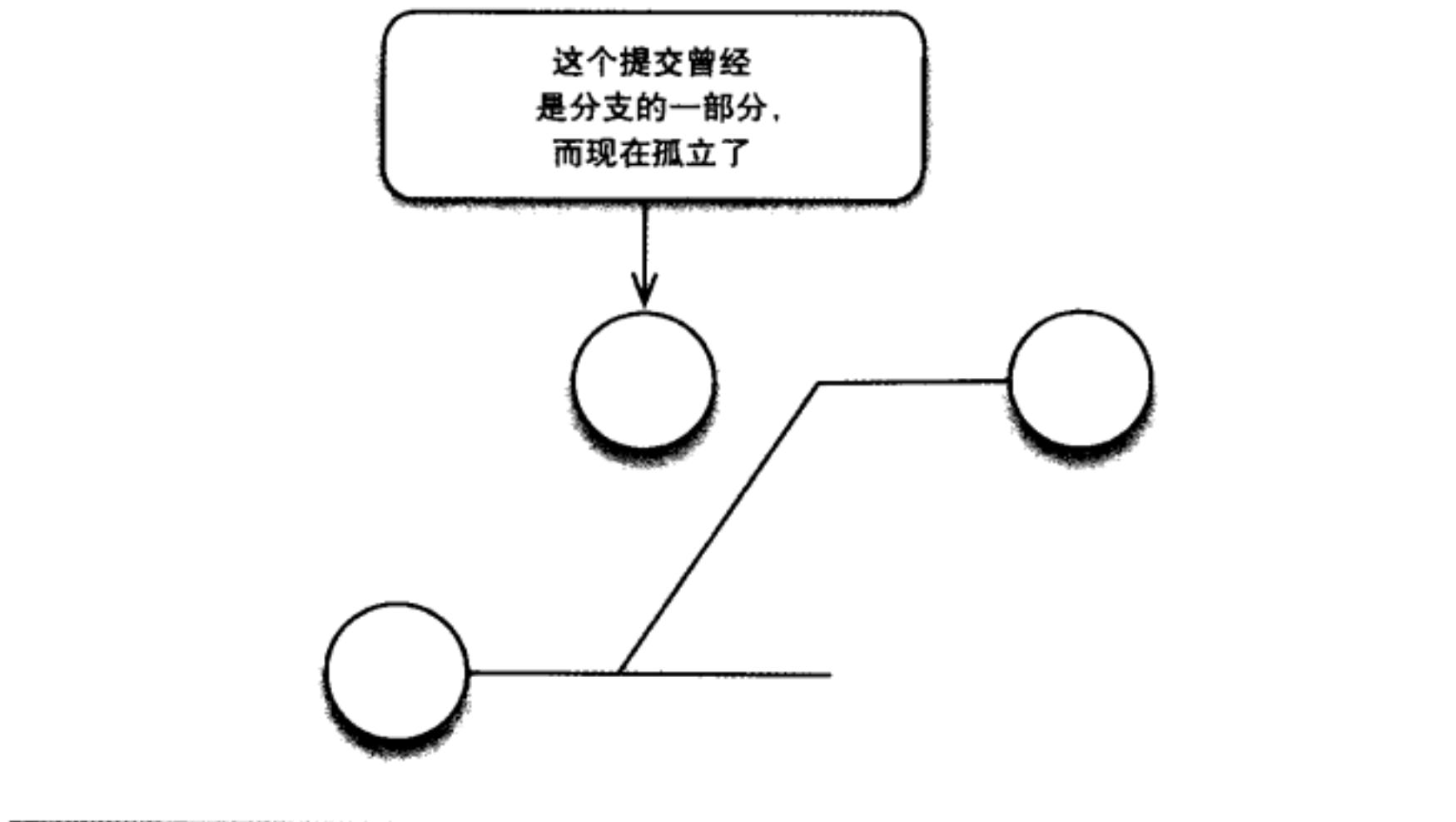


图 9.1 变基操作后，一个提交被孤立了

重现是指记录分支末梢变化的情况，并予以复现。第 5 章“理解和使用分支”（第 55 页）介绍过，分支本质上只是指向最新提交的指针。重现功能可以记录和跟踪所有这些指针的变化。通过使用它，你能找到想要的提交，并据此恢复分支。

例如，让我们先建一个简单的版本库并把它弄乱。创建一个新的版本库，做一次提交，然后新建一条分支，并在其中添加两个新提交。提交中可以包含任意内容，只要按这个顺序即可。

现在，使用 9.3 节“分支变基”（第 118 页）和 6.7 节“重新改写历史记录”（第 85 页）中介绍的 `git rebase` 或 `git rebase -i` 命令，删除第二条分支中的第一个提交。现在的版本树应该如图 9.1 所示。

你可以使用 `git log` 核对版本库历史。

这个历史正如料想的那样，使用 `git rebase -i` 删除一个提交后，看到的历史就只有两个提交。现在用 `reflog` 可以看到更多的提交：

```
prompt> git reflog
1b41334... HEAD@{0}: commit: third commit
5e685de... HEAD@{1}: checkout: moving from reflog to 5e685de...
0cb04ad... HEAD@{2}: commit: third commit
71bc515... HEAD@{3}: commit: second commit
5e685de... HEAD@{4}: checkout: moving from master to reflog
```

这里缩短了命令输出中的第二列，以便适合页面排版。这个输出结果是按时间倒序排列的，就好像 `git log` 的输出那样。注意其中有两个提交的提交留言都是 `third commit`。`0cb04ad` 是第一个这样的提交。`1b41334` 是第二个这样的提交，发生在 `git rebase` 删除 `71bc515` 后。

因此，为了恢复第一个 `third commit`，可检出这个提交：

```
prompt> git checkout 0cb04ad
Note: moving to "0cb04ad" 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 0cb04ad... third commit
```

正如以上信息所示，尽管现在不在任何分支上，但可以使用简单的命令 `git checkout -b` 创建一个。不在任何分支上并不妨碍查看历史记录。运行 `git log` 命令可以看到第二个提交已经恢复了。

```
prompt> git log --pretty=format:"%h %s"
0cb04ad third commit
71bc515 second commit
5e685de initial commit
```

当然，并非必须通过检出分支操作来创建分支。如果不需要在创建分支之前查看历史记录，那么可以简单地调用 `git branch` 命令来创建分支：

```
prompt> git branch reflog-restored 0cb04ad
prompt> git checkout reflog-restored
Switched to branch "reflog-restored"
```

在 9.1 节“压缩版本库”（第 116 页）中介绍过命令 `git gc` 会删除比较旧的重现记录。例如刚才恢复的那个提交，如果没有恢复，通常会在 30 天后被删除。你可以通过修改配置 `gc.reflogExpireUnreachable` 的值来改变有效期。同样，一般来说，除非修改 `gc.reflogExpire` 的值，重现记录本身会在 90 天后过期。这是为了防止重现记录积压过多。

一般很少用到重现功能，当用到的时候，它就像救生员一样。重写历史记录虽然很危险，但 `git reflog` 提供了一张安全网。

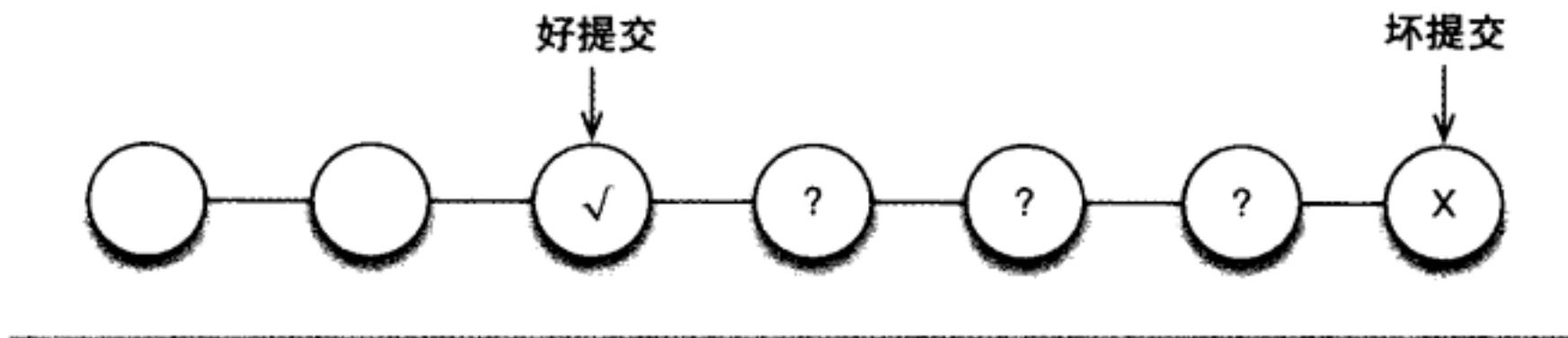


图 9.2 在版本库中，已知一个好提交和一个坏提交

## 9.5 二分查找

Bisecting Your Repository

不管如何努力，开发软件过程中避免不了要出现 Bug。虽然开发过程中进行了单元测试，客户那边也有严格的接收测试，设计和代码实现过程中也进行了深入审慎的思考，但 Bug 仍然不时出现，因为人类的思维本身并非完美。

所以开发过程中，要为忽然冒出来的 Bug 做好准备。使用单元测试、接收测试等工具可以帮助减少 Bug 的出现，然而，一旦 Bug 出现，必须尽快定位是历史上哪个提交引入了这个 Bug。命令 `git bisect` 这时可以帮上忙。

`git bisect` 基于一个已知的坏提交和一个已知的好提交，逐步排查版本库中的历史记录。它带领你在版本库中的历史上标出哪些是好的提交、哪些是坏的提交，直到最后找出那个引进 Bug 的提交。

考虑这个例子：项目版本库中有一个 1.0 版标签，而当前项目代码正朝着 1.1 版演进。此时 Beta 测试人员报告了一个新 Bug，该 Bug 在 1.0 版中尚不存在。这时，在版本库里就有一个好提交和一个坏提交，可以使用命令 `git bisect` 对版本库进行查找，直至分离出引入 Bug 的提交为止。

此时版本库如图 9.2 所示，可以看到有三个提交可能引起这个 Bug。

现在建立一个测试用例来测试该 Bug，先在坏的提交上测试并确定该 Bug 存在；然后在已知的好提交上测试，并证明该 Bug 不存在。

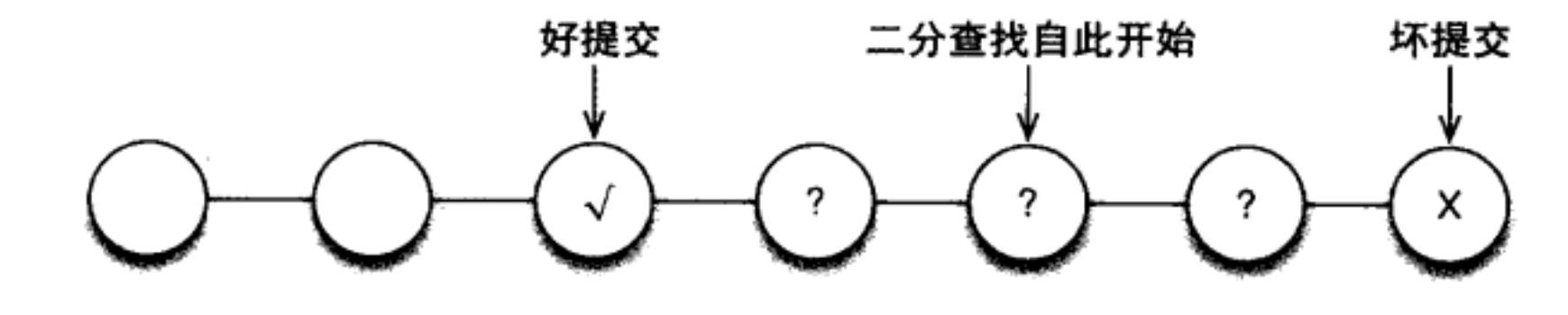


图 9.3 从中间开始二分查找

现在，移到坏的提交，运行 `git bisect start` 来开始二分查找，然后调用 `git bisect bad`，最后调用 `git bisect good <某一个提交或标签>`，执行过程如下：

```
prompt> git bisect start
prompt> git bisect bad
prompt> git bisect good 1.0
Bisecting: 1 revisions left to test after this
[d3289f8ab4916f58134a7e5c813ca91c13dd6a70] <commit message>
```

一旦标出好坏两个点，Git 就会带你到版本库中位于这两个点正中间的那个地方，检出对应的提交。这时的版本库如图 9.3 所示。

再测试一下，确定这个提交是好的还是坏的。在这个例子中，这个提交是坏的，所以使用 `git bisect bad` 标出它：

```
prompt> git bisect bad
Bisecting: 0 revisions left to test after this
[1318fa48089ea36c082d7e69cbd7c04b489821ec] <commit message>
```

现在，在好提交与坏提交之间只剩下一条提交可测试了。从逻辑上讲，如果一条提交之前的所有提交都是好提交，而该提交之后的所有提交都是坏提交，就可以判定该提交就是引入 Bug 的那条提交。快速测试一下，的确如此。

将该提交标识为坏提交后，Git 就知道引入问题的提交已经找到了，于是 Git 将该提交的日志条目显示出来。既然已经把范围缩小到这一个提交，剩下的就是找出该提交中哪部分修改引起了 Bug，然后修复该 Bug。

`git bisect` 命令搜索版本历史时，已经将我们带离了 HEAD，所以在提交新修改之前，必须回到出发点。

此时只须要运行 `git bisect reset` 命令就可以了：

```
prompt> git bisect reset
Switched to branch "master"
```

当然，像 Git 中所有有趣的东西一样，除了这个简单的例子之外，`git bisect` 还可以做其他许多事情。当希望直观地理解代码的演进历史时，能有个可视化历史记录就再好不过了。为此可以使用 `git bisect visualize` 命令。

如果喜欢文本输出，可以使用 `git bisect log` 命令显示有哪些提交被标成好提交或坏提交。这也有助于发现标错的地方。

如果要修改，可以将 `git bisect log` 存储成一个文件；删除该文件中刚才错误的标识操作及其后的所有记录；然后将该文件作为命令 `git bisect replay <某一文件>` 的参数，让 Git 重新执行，直到刚才错误的标识操作之前的一步。

这些功能虽很强大，但都需要许多手工操作，不但增加工作量，而且可能会引入人为错误。自动测试套件可以解决测试本身的自动化，而 `git bisect` 更进一步，可以适时自动运行这类测试套件。

为使用 Git 的这一功能，须要构造一个可以在命令行运行的脚本，且该脚本在测试通过时返回代码“0”，当测试失败时返回一个正数。这个正数通常是“1”，如果要跳过一条提交并让 Git 二分查找自动移向下一条提交，则应使用“125”作为返回代码。一般情况下用不着跳过某条提交，除非判断不出该提交是好提交还是坏提交。

命令 `git bisect run` 可用来自动运行这个脚本。例如，创建一个名为 `run-tests` 的脚本，然后这样调用：

```
prompt> git bisect start HEAD 1.0
Bisecting: 1 revisions left to test after this
[d3289f8ab4916f58134a7e5c813ca91c13dd6a70] <commit message>
prompt> git bisect run /work/run-tests
running /work/run-tests
```

`git bisect` 命令将继续运行，正如手工运行时一样，唯一不同的是，直到 Git 使用脚本 `run-tests` 返回值。每当脚本返回一个非 0 整数时，Git 就将当前的提交作为一个坏提交。

注意该命令中使用的是另外一个目录下的文件（而非当前工作目录树中的文件），这是为了确保 Git 不会改变 `git bisect` 正在运行的脚本文件。使用版本库以外的文件来驱动测试不是必须的，但有助于避免产生问题。

`git bisect` 是不经常使用的命令之一，但是当使用它的时候，就像个救生员，将它和能在命令行中执行的测试套件一起联用，可以快速定位 Bug。

公司中的软件开发团队甚至可以将这种方法与他们的自动编译系统相集成，以帮助定位 Bug。当项目使用持续集成工具时，如果程序员的提交太频繁，持续集成系统难以为每个提交做编译和测试。这种情况下，自动二分查找法将节约大量的时间。

本章是本书第 2 篇的结尾。至此，Git 的基本知识介绍完了。无论你在阅读本书前是不熟悉版本控制的开发人员，还是 Git 高手，现在都应该掌握自如运用 Git 的本领了。

本书第 3 篇将介绍 Git 的系统管理。如果打算从 Subversion 或 CVS 迁移到 Git，可阅读第 10 章“迁移到 Git”（第 131 页）；如果打算架设远程公共版本库，可阅读第 11 章“使用 Gitosis 管理 Git 服务器”（第 143 页）。





## 第3篇 系统管理

---

System Administration





# 第 10 章

## 迁移到 Git

Migrating to Git

那些已经经历了长时间开发的项目，恐怕早已使用 Subversion 或 CVS 存储和管理版本历史了。把这样的项目迁移到 Git 下，并不意味着将丢失所有的历史记录。实际上，Git 提供的一系列工具使得迁移变得很容易。

本章将介绍如下操作：

- 从 Subversion 导入历史记录。
- 与远程 Subversion 保持同步。
- 从 CVS 中导入历史记录。

在此列出了 Subversion 和 CVS 的命令与 Git 命令之间的对应关系。Subversion 与 Git 常用命令对照如图 10.1 所示（下一页）；CVS 与 Git 常用命令对照如图 10.2 所示（第 133 页）。

### 10.1 与 SVN 的通信

Communicating with SVN

目前许多公司使用 Subversion 来管理版本。Subversion 的设计目标是取代 CVS，它取得了很大的成功。Subversion 与 CVS 相比，它有着清晰、容易理解的命令结构，并且，它的原子提交机制（一次提交包括若干个文件的修改）使得记录和跟踪变更变得容易。

Git 的设计目标并非是要取代 Subversion 或其他版本控制系统，但是它强大的功能使得它或将取代 Subversion。

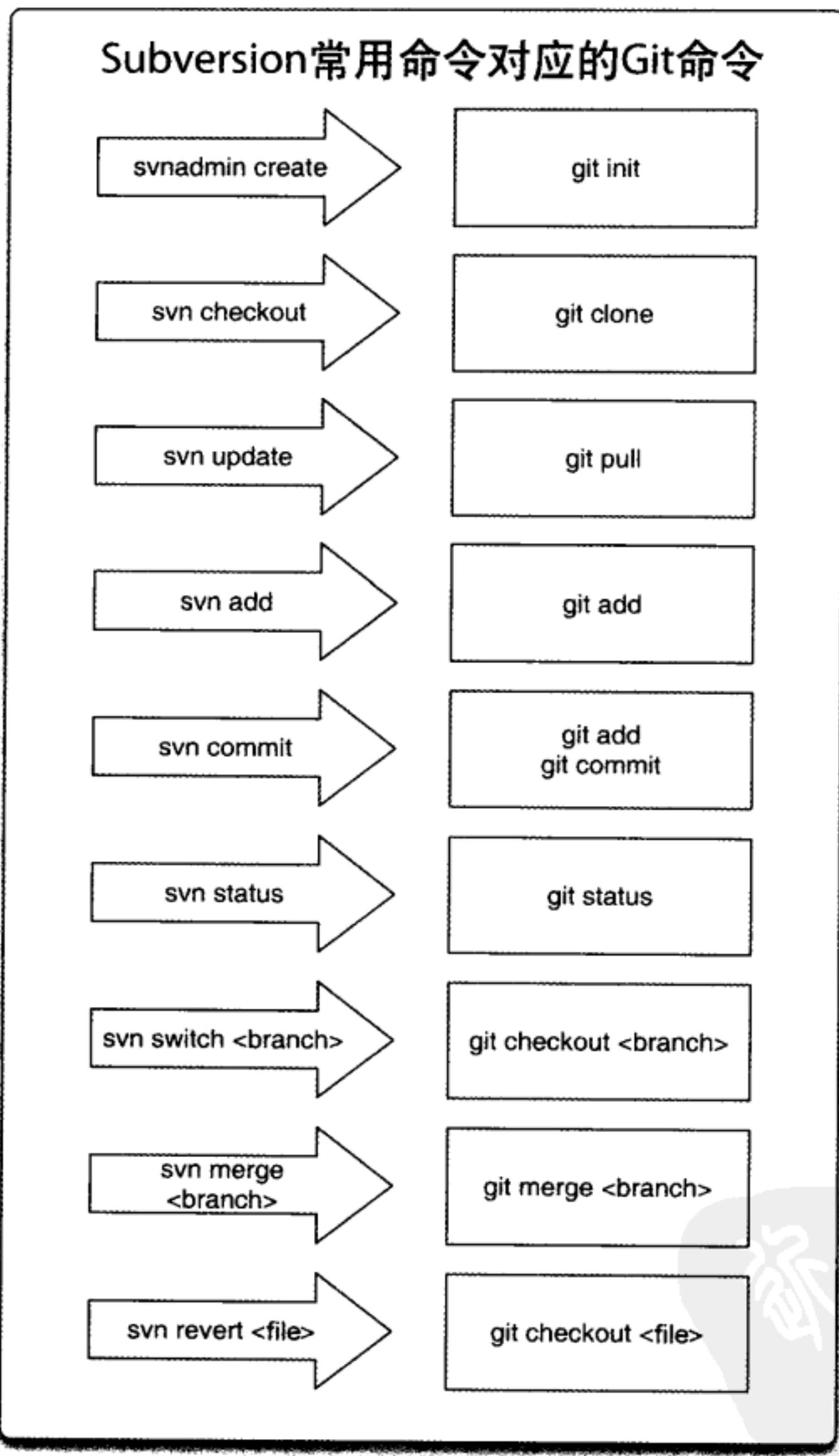


图 10.1 Subversion 与 Git 常用命令对照

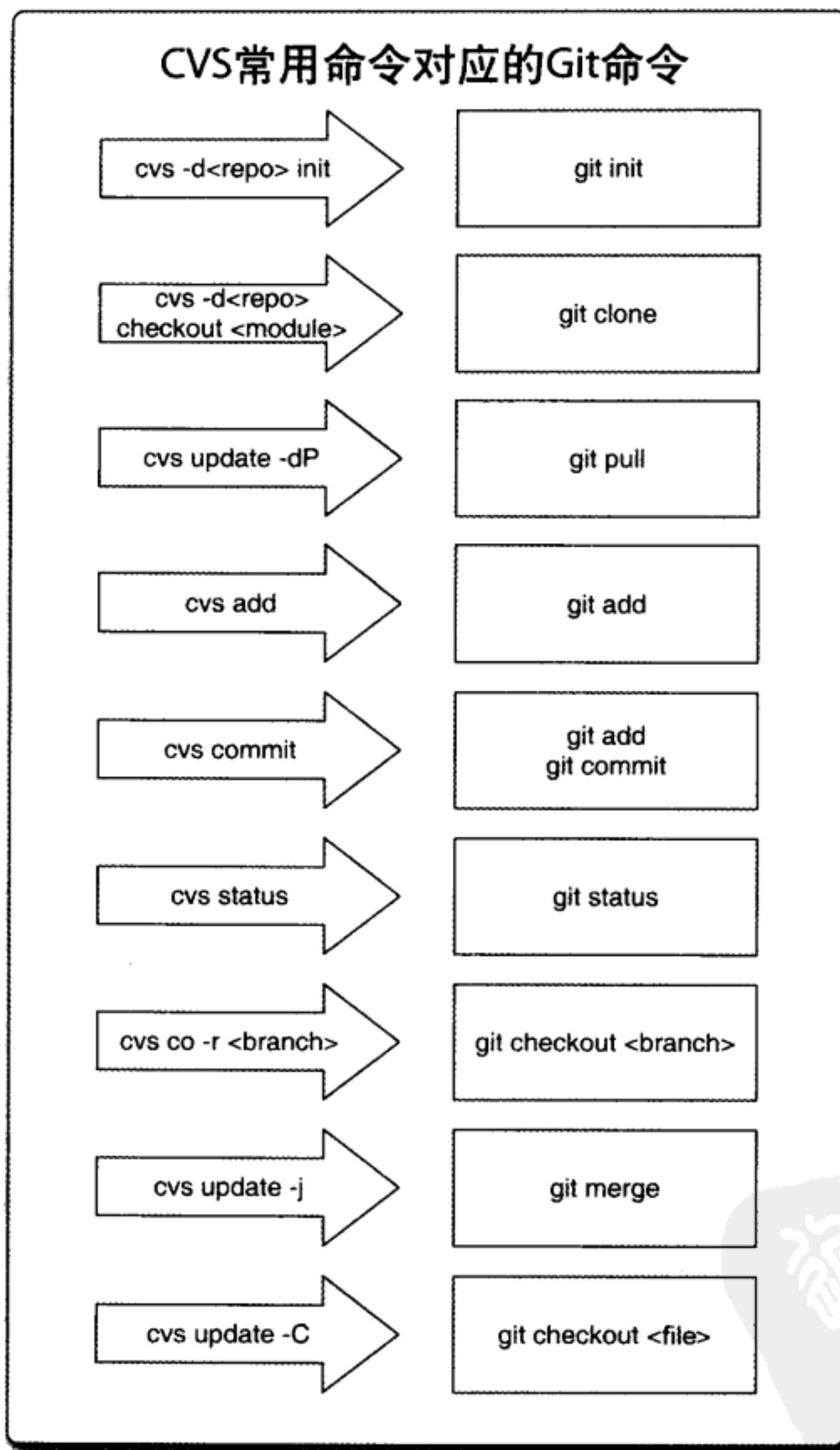


图 10.2 CVS 与 Git 常用命令对照

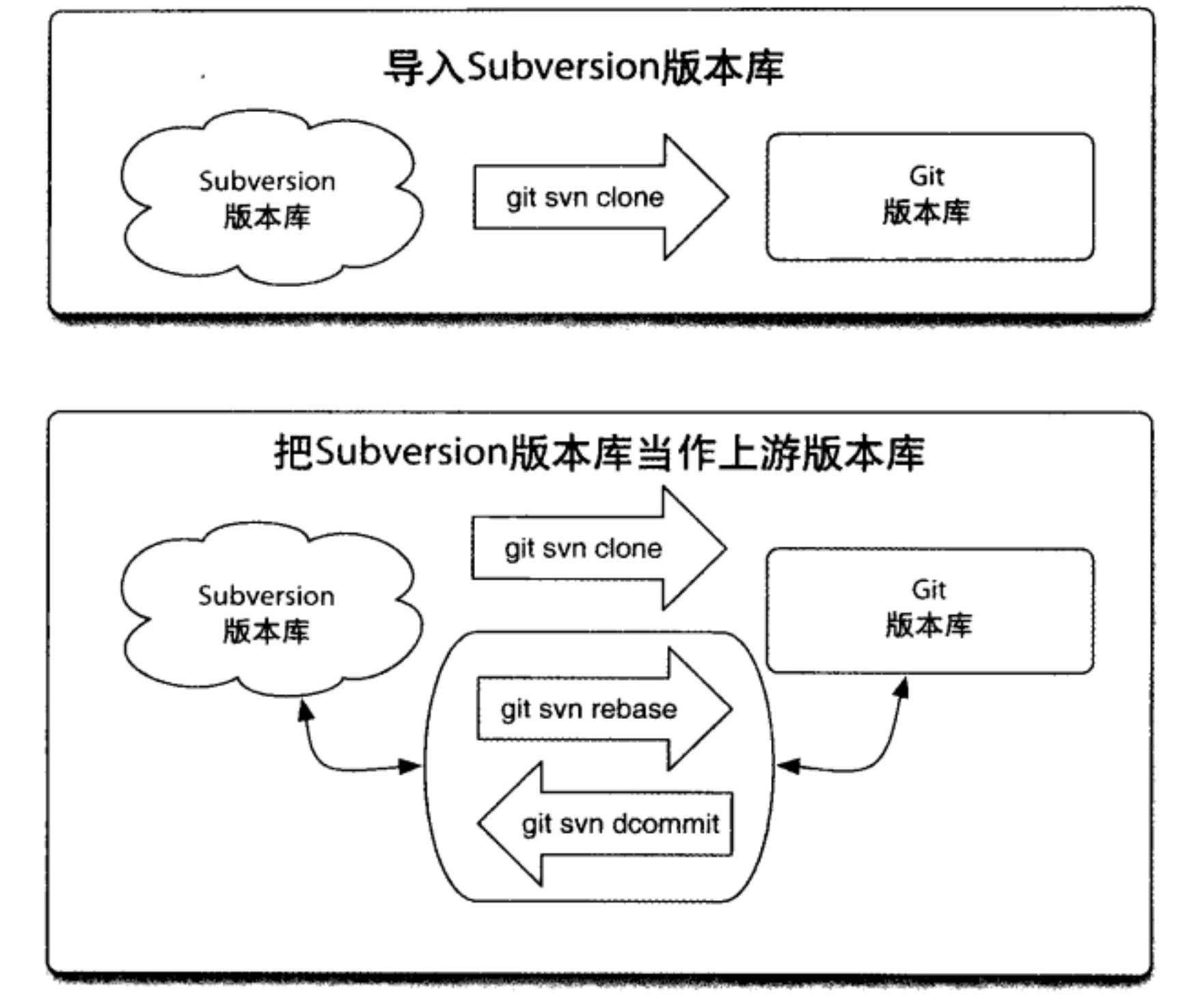


图 10.3 从 Subversion 迁移到 Git 的两种方案

一个简短说明：本章中有的地方写的是“git-svn”，有的地方写的是 `git svn`。前者用来表示这个工具本身，后者是命令行。

从 Subversion 迁移到 Git 有两种方法。第一种方法是从 Subversion 中导入所有历史记录到 Git，然后使用 Git 作为主要的版本控制工具。第二种方法是保持 Subversion 仍作为“官方”方法（而项目中少数人开始使用 Git），这是利用 Git 既能从 Subversion 中导入更改，又能把更改推入（Push）回到 Subversion 去的能力。这两种方法的简明过程如图 10.3 所示。

## 10.2 确保 git-svn 是可用的

Making Sure git-svn Is Available

Git 中有个叫 git-svn 的工具，用于与 Subversion 版本库通信。安装 Git 时，你可能已经安装了 git-svn。可使用命令 `git svn --version` 查看安装与否：

```
prompt> git svn --version
git-svn version 1.6.0.2 (svn 1.4.4)
```

如果输出结果与上面相同，那么万事具备，可以开始使用它了。如果与上面的输出结果不同，则可能出现如下两种错误信息：

- Git 说 `git svn` 不是一个 Git 命令，或者报一个“fatal error”错误。
- Git 说它找不到 `SVN/Core.pm`。

### 非 Git 命令或“fatal error”

当通过像 Ubuntu 的 `apt-get` 或 OS X 中的 MacPorts 这样的安装包管理器来安装 Git 时，容易出现这样的问题。原因是，这些安装包管理器把 Git 拆分成许多个小的包，以便减少安装的依赖性。

要在 Linux 安装 git-svn，须要找到相应的安装包。在 Ubuntu 中，这个包叫 `git-svn`：

```
prompt> sudo apt-get install git-svn
```

如果是按 2.1 节“安装 Git”（第 15 页）中介绍的 MacPorts 安装的 Git，则你大概已经安装过 `git-svn` 了。类似的，如果是用下载并编译源代码的方法安装 Git，则已包含 `git-svn` 了。

### 缺失“`SVN/Core.pm`”库

这是另一个可能出现的错误：Perl 报告找不到 `svn/core.pm` 库文件。这是因为 Perl 没有与 Subversion 绑定。

有几个方法可找到必须的文件。对大部分人来说，最简单的方法是使用包管理器。

在 Ubuntu 中，这个缺失的包叫 `subversion-perl`，要安装它，可键入命令：

```
prompt> sudo apt-get install subversion-perl
```

### 在以 Subversion 为主的公司中使用 Git

我曾经有一段在公司里从事测试和调试应用程序的经历。在找到问题根源之前，我得做很多试验，这些试验广泛分散于不同的领域中。而工程部门使用 Subversion 去跟踪他们的代码修改。

我的开发工作并不总是在一条线索上，一周中常常工作于好几个不同的领域。Subversion 的集中式版本控制方式并不适合我，特别是其中复杂的分支操作，以及缺乏合并跟踪功能。

Git 中方便的分支操作让我工作效率大大提高，不用花费精力和体力去维护足有半打之多的代码拷贝（也就是 Subversion 中的分支）。

还有一个好处，遵从《Practices of an Agile Developer》[SH06]一书的建议，Git 可以让我做到，只有准备好的时候才把工作成果共享给别人。否则别人就可能看到我的尚不成熟的修改，并且还以为它已经是正确完善的。

使用 Git 可以管理多条分支，保持它们跟上时代，而无须人工记录和跟踪合并情况；可以在把若干提交推入版本库之前，先把它们规整好。并且保证只有完成并通过检验的代码才会被别人看到。

如果使用 Mac 系统，MacPorts 中这个安装包的名称是 `Subversion-perlbindings`：

```
prompt> sudo port install subversion-perlbindings
```

如果使用 CPAN 来保持 Perl 库更新，可以通过 CPAN 安装软件包 `SVN::Core`：

```
prompt> sudo cpan install SVN::Core
```

## 10.3 导入 Subversion 版本库

*Importing a Subversion Repository*

使用 `git svn clone` 克隆命令，可将 Subversion 版本库中的历史记录导入 Git 中。其用法和普通 Git 克隆命令 `git clone` 很相像。命令 `git svn clone` 取出整个 Subversion 版本库的历史记录，并将它们存入 Git 版本库中。

### 在 Mac 系统上使用 CPAN 要谨慎

我花了两个下午的时间才解决这个问题：MacPorts 说什么都不缺，可 Git 却找不到 Perl 与 Subversion 的绑定。问题的原因是，我的系统上有两个版本的 Perl 与 Subversion 的绑定：一个是以前安装 Subversion 时安装的老版本，另一个是 MacPorts 上的新版本。

更糟糕的是，在我意识到这个问题之前，还通过 CPAN 安装了另一个 Perl 与 Subversion 的绑定。因此，实际上我已经安装了三个版本！CPAN 上的那个版本可以正常使用，但当我用 Git 调用它时，另外一个依赖问题出现了。当意识到 CPAN 库中并没有包括全部需要的内容时，问题很快就解决了，但在这之前还是花了我好几个小时！

你必须把 Subversion 版本库的组织结构告诉 Git。如果 Subversion 版本库遵循了默认结构<sup>1</sup>，则在命令中加上参数-s，让 Git 按标准组织结构理解 Subversion 版本库。

在 Subversion 版本库中，如果分支和标签地址在不同的位置，则可以在命令中使用-b 和-t 参数分别指定。同样，如果 Subversion 版本库中的主干没有命名为 trunk，则使用-T 参数指定具体名称。

```
prompt> git svn clone --prefix svn/ -s svn://svnrepo/sunshine
Initialized empty Git repository in /work/sunshine.git/
Using higher level of URL: svn://svnrepo/sunshine =>
svn://svnrepo/sunshine/
This may take a while on large repositories
branch_from: /tags => /tags/prototype
Found possible branch point:
svn://svnrepo/sunshine/branch/tswicegood-speed
r1 = 305a06a00d6048fe36ee01bb96b7c770cc30317e (trunk)
      A tests/basic-use-cases.py
r2 = a7f1299c2821f788d64218cc4e366ebe13202105
      A src/sunshine/__init__.py
      A src/sunshine/framework.py
... etc., etc. ...
r50 = c534dfb90a8365842af86fde85065d76baca260a (trunk)
Checked out HEAD:
  svn://svnrepo/sunshine r50
```

<sup>1</sup> <http://svnbook.red-bean.com/en/1.1/ch04s07.html>

### 压缩版本库大小

如果克隆超过 100 个提交的 Subversion 版本库，则克隆结束后，将发现磁盘空间消耗了许多。

根据 Subversion 版本库生成 Git 版本库后，应尽快运行 `git gc` 命令来减少磁盘空间占用。这就好像在几百个提交后须要运行 `git gc` 一样。

如果在生成 Git 版本库后第一件事是使用 `git gc` 来压缩版本库，那么可以安全地使用 `--aggressive` 参数，当然这可能要多花些时间。作为个人经历，我花了超过 14 个 CPU 小时<sup>2</sup> 才压缩完一个刚克隆的、超过 31 000 个提交的新版本库。

Subversion 不知道如何处理诸如“把整个历史记录给我”之类的请求，因此 `git-svn` 只能向 Subversion 每次查询一个提交。对于有成千上万个提交的大型版本库，可能要差不多一整天时间！

注意，前面介绍 `git svn clone` 命令的时候，使用了 `--prefix` 参数，用它来通知“git-svn”添加前缀。例如，为所有当初在 Subversion 中的分支，添加其 Subversion 版本库名作为前缀。如果不添加前缀，远程分支（也就是 Subversion 中的分支在本地 Git 版本库中的映像）和本地分支就可能重名。添加此参数并不是必须的，但添加之后，使用命令 `git branch -a` 查看所有分支时，结果就容易理解多了。

如果某些旧提交不重要，可以使用参数 `-r` 来指定从哪个版本开始克隆。事实上，指定一个版本，Git 只要把这个版本克隆过来即可。随后使用另一个命令 `git svn rebase`，把这个版本之后的修改也拖过来。下一节也会用到该命令，在这里它是用来在初次导入之后，把上游 Subversion 版本库中的导入点之后的改动拖过来。

## 10.4 与 Subversion 版本库保持同步更新

*Keeping Up-to-Date with a Subversion Repository*

无论是与一个活跃的 Subversion 版本库保持同步，还是使用 `git svn clone -r ...` 命令创建新的 Git 版本库，都可以使用两种方式从 Subversion 版本库中拖入变更。

<sup>2</sup> CPU 小时 (CPU hour) 指用小时为单位来描述 CPU 处理特定事务所需时间的累积值。—译者注

1// Joe 问……

这是什么错误？

如果从 Subversion 版本库中克隆一个项目到 Git，且该项目不在该版本库的初始版本里<sup>3</sup>，则克隆过程中可能会出现类似下面的出错提示：

```
W: Ignoring error from SVN, path probably does
not exist: (175002): RA layer request failed:
REPORT request failed on '/test/!svn/bc/100':
REPORT of '/test/!svn/bc/100': Could not read
chunk size: Secure connection truncated
(svn://svnrepo)
W: Do not be alarmed at the above message
git-svn is just searching aggressively for
old history.
```

正如第二个警告提示（以“W:”开头的行）所说的，不必担心。当 git-svn 克隆 Subversion 版本库的时候，它会从初始版本开始，并持续向前搜索，直到你指定的版本。

该警告提示，在初始版本里，找不到该项目。但它会继续沿历史搜索，直到找到第一个包含该项目的版本为止。

第一种方式是使用 `git svn fetch` 命令取来所有远程修改到远程分支，但是不会合并到本地分支。这也是用来从 Subversion 中获取新分支的命令。

第二种方式是使用 `git svn rebase` 命令，一般常用这种方式。这种方式类似于先运行 `git svn fetch` 命令，然后运行 `git rebase` 命令。它从上游 Subversion 版本库中取来全部新提交到本地 Git 版本库中的远程分支，然后变基当前本地分支到该远程分支。

正如 `git rebase` 命令一样，对于已经同步回 Subversion 的本地 Git 库中的提交，变基时不会再次添加该提交。同样，类似于 `git rebase`，如果在本地工作目录树中存在未提交修改，`git svn rebase` 命令不能运行。

<sup>3</sup> 一个 Subversion 版本库可能包含若干个项目，每个项目是一个子目录。该版本库的初始版本，不一定包含了所有的项目。一译者注

### 克隆 Subversion 版本库的另一个方法

克隆 Subversion 版本库有可能是个耗时的过程。我曾经克隆过一个很大的版本库，耗费了 20 多个 CPU 小时。所以，如果想要克隆很大版本库，克隆之前最好先看看有没有引导版本库。

引导版本库是一个常以 tar 包形式存在的 Git 版本库，它是克隆 Subversion 版本库生成的。在开源社区，正常情况下，大约每周会更新一次引导版本库。

在获得引导版本库后，唯一要做的就是使用 `git svn rebase` 命令来获得 Subversion 版本库中所有的新版本，也就是最近更新引导版本库之后的改动。这比从头克隆 Subversion 版本库的全部历史记录快多了！

## 10.5 将修改推入 SVN

Pushing changes to SVN

如果你想从 Subversion 完全迁移到 Git，随后废弃 Subversion 不用，就可以跳过本节的内容。

但是，与其他流行的分布式版本控制系统相比，Git 还有一个好处，它不仅可以从 Subversion 版本库持续拖入修改，还可以将 Git 中的修改推回 Subversion 版本库中。

将修改推入 Subversion 版本库的命令是 `git svn dcommit`。它从本地版本库抓取每个提交，并将它们依次提交回 Subversion 版本库。

在这个过程中，该命令会执行变基操作，让本地提交基于 Subversion 版本库中已有的内容。就像 `git svn rebase` 命令一样，这样做的目的是让 git-svn 更容易确定已经同步了哪些内容。

可以使用 `-n` 选项告诉 Git，只是做个演练（dry run）。在这种模式下，Git 会仔细检查历史记录并列出所有向 Subversion 版本库发送的提交。

## 10.6 从 CVS 导入

[Importing from CVS](#)

同 Subversion 相比，从 CVS 导入 Git 稍显麻烦。实际上，从 CVS 版本库导入到 Git 最可靠的方法是，先使用 `cvs2svn` 命令<sup>4</sup>把 CVS 版本库转换成 Subversion 版本库，然后由 Subversion 版本库导入到 Git。`cvs2svn` 是一个把 CVS 版本库转换成 Subversion 版本库的工具。

事实上，Git 工具包中有一个称作 `git cvsimport` 的工具，用于导入 CVS 版本库。但是由于使用 `cvs2svn` 命令要比它稳定得多，这里就重点介绍 `cvs2svn`。

首先，要从 CVS 版本库中得到修改控制系统（RCS）文件。一旦有了这些，就可以使用 `cvs2svn` 工具将它们转换成 SVN 转储文件（Dump File）：

```
prompt> cd /path/to/cvs-rcs-files
prompt> cvs2svn --dumpfile=svndump
```

使用类似 `svndumpfilter`<sup>5</sup>的工具创建转储文件可以过滤掉不需要的数据。新的 SVN 转储文件准备好后，须要创建一个 Subversion 版本库将它导入，这可以使用 `svnadmin create` 命令来完成：

```
prompt> cd ..
prompt> svnadmin create ./tmpsvn
prompt> svnadmin load ./tmpsvn < /path/to/cvs-rcs-files/svndump
```

如果将历史记录装载入 `tmpsvn`（一个临时的 Subversion 版本库）过程中没有出现问题，就可将它导入 Git 中了。其过程与 10.1 节“与 SVN 的通信”（第 131 页）中介绍的一样：

```
prompt> git svn clone file:///path/to/tmpsvn
```

本章讲解了如何从 Subversion 和 CVS 中导出历史记录并导入到 Git 中，甚至可以把 Subversion 版本库当作上游版本库，将本地修改推回去。

下一章将介绍如何通过 Gitosis 来管理 Git 上游版本库。

---

<sup>4</sup> 可以从 <http://cvs2svn.tigris.org/> 下载 `cvs2svn` 源代码，也可以通过常用的安装包管理软件获得，比如 Linux 上的一些安装包管理软件，或者 OS X 上的 MacPorts。

<sup>5</sup> <http://svnbook.red-bean.com/en/1.1/ch05s03.html#svn-ch-5-sect-3.1.3>



## 第 11 章

# 使用 Gitosis 管理 Git 服务器

Running a Git Server with Gitosis

Gitosis 是一个彰显 Git 荣耀的工具。它表明，Git 不仅仅是一个分布式版本控制工具。一些在底层需要文件版本控制功能的应用程序，也可以把 Git 作为其基础。

具体来说，Gitosis 是一个远程管理 Git 服务器及其上的各个版本库的工具。它的实现机制是，使用一个特别的 Git 版本库（Gitosis 配置版本库）来存储服务器上各版本库的配置信息。要更新配置，只须将新的配置信息推入这个特别的版本库即可。

Gitosis 利用 Git 的钩子脚本（hook script）来实现这样的功能。这些钩子脚本存储在（Gitosis 配置版本库的）`.git/hooks/` 目录下。不同事件的发生会触发相应的钩子脚本。比如，提交操作前会触发 `pre-commit` 脚本，提交操作后会触发 `post-commit` 脚本。

Gitosis 利用这些脚本来更新它所管理的各个版本库的配置。包括创建新版本库并设置其读写权限等。

Gitosis 目前仍然是个诞生不久的工具。因此，安装和配置工作还有点麻烦。具体安装步骤如下：

1. 确定 Gitosis 所依赖的程序已经安装。
2. 安装 Gitosis。
3. 配置 Gitosis 服务器。
4. 创建管理员 SSH 证书。
5. 初始化 Gitosis。
6. 配置 Gitosis。

对于不熟悉配置服务器、运行后台程序（Daemon）这类任务的读者来说，本章可能不太好懂。尽管本章尽量写得浅显易懂，然而它探讨的内容确实博大精深。

不熟悉搭建服务器的读者也不必烦恼。有几种办法来解决这个问题。首先，可以想办法向系统管理员求助，并把这章的内容给他看。

而如果找不到系统管理员，也可以考虑使用互联网上的 Git 托管服务。GitHub 和 Gitorious 两个网站都对外提供免费的托管服务，在附录 B.3 “Git 版本库托管服务”（第 169 页）中有详细介绍。

## 11.1 确定 Gitosis 所依赖的程序已经安装

从 GitHub 上下载了 Gitosis 的源代码并解压后

Gitosis 依赖于若干其他程序。首先，Gitosis 是用 Python 语言编写的，因此在安装 Gitosis 前必须先安装 Python。Python 很流行，因此你可能已经安装了。确定安装与否的最简单方法是运行如下命令：

```
prompt> python --version
Python 2.5.1
```

如果已经安装 Python，就会看到类似的运行结果，尽管显示的版本号可能不同。

接着还须安装 Python Enterprise Application Kit (PEAK) 提供的 EasyInstall 包。EasyInstall 用于安装用 Python 语言写成的软件，它能够帮助处理这些软件间的依赖关系。EasyInstall 也很流行，因此可能也已经安装了。运行如下命令确定安装与否：

```
prompt> python -c "import setuptools"
```

如果 EasyInstall 及其 `setuptools` 模块都已经安装好，那么上述命令不会显示任何输出信息。而如果运行上述命令得到类似于下面例子中给出的警告信息，就须安装 EasyInstall。为此，请阅读操作系统自带的文档，获得关于如何安装 EasyInstall 的信息。或者，阅读 EasyInstall 网站<sup>1</sup>上的内容。

```
prompt> python -c "import setuptools"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named setuptools
```

---

<sup>1</sup> <http://peak.telecommunity.com/DevCenter/EasyInstall>

## 11.2 安装 Gitosis

安装 Gitosis 的源代码

Gitosis 还在不断演进中，我们需要从其源代码安装。你可以通过克隆它自身的 Git 版本库来取得其源代码：

```
prompt> git clone git://eagain.net/gitosis
Initialized empty Git repository in /work/gitosis/.git/
remote: Counting objects: 599, done.
remote: Compressing objects: 100% (168/168), done.
remote: Total 599 (delta 422), reused 599 (delta 422)
Receiving objects: 100% (599/599), 92.57 KiB | 59 KiB/s, done.
Resolving deltas: 100% (422/422), done.
```

克隆完成后，进入工作目录树并运行 `setup.py` 脚本：

```
prompt> cd gitosis
prompt> sudo python setup.py install
running install
数百行的输出……
```

现在 Gitosis 已经安装好了，接下来要在该服务器上创建一个供 Gitosis 使用的账户。在此之前，要为管理员创建 SSH 证书。

## 11.3 创建管理员 SSH 证书

生成 Gitosis 管理员的 SSH 公共密钥

须要生成 Gitosis 管理员的 SSH 公共密钥。SSH 公共密钥是 SSH 协议用来验证身份的，它不同于用密码验证身份的方式。

如果你（作为 Gitosis 管理员）已经有了一个 SSH 公共密钥，就不必重新创建，直接使用它就可以了。如果没有，就须创建一个这样的密钥。密钥可以通过在命令行窗口中使用 `ssh-keygen` 命令来创建。

运行该命令的本地计算机，可以与 Gitosis 所使用的服务器连接。通过下述命令生成密钥：

```
prompt> ssh-keygen -t rsa
```

程序运行时将询问在何处存储新生成的密钥，按回车键接受默认的位置存储即可。接下来程序会提示输入口令，这个口令是使用密钥时必须的，就好像一般的密码一样，请谨慎选择口令并妥善保管，以免泄露。

`ssh-keygen` 程序创建了新的密钥，并提供了新的密钥的相关信息。如果按照默认设置存储位置和用户名信息，公共密钥现在应该位于这个存储位置中：

`/home/<username>/ .ssh/id_rsa.pub`

请将这个文件拷贝到服务器上，后面初始化 Gitosis 时会用到该文件。

## 11.4 配置 Gitosis 服务器

Configuring the Server for Gitosis

要在服务器运行 Gitosis，需要具备两个条件：在服务器上 Gitosis 有它自己的账户；在服务器上有一个存储各 Git 版本库的目录。

每个操作系统都有其特定的添加账户的方法。在大多数 Linux 系统中，可使用 `adduser` 命令来完成。

要为 Gitosis 创建一个新的账户。该账户的家目录（home directory）就是 Gitosis 存储各 Git 版本库的地方，其中也包括那个 Gitosis 配置版本库，它存储着各版本库的配置信息。在大多数 Linux 系统中，可通过命令 `adding --home /path/to/home` 来指定该目录。

该目录可以位于系统中的任意位置。按 Gitosis 约定，它位于 `/srv/example.com/git`（请把 `example.com` 换成实际的域名），但不是必须遵守的。

Gitosis 账户的名称也是任意的，本书中我们使用 `git` 做它的账户名。若使用其他名称，请在使用本章中的命令时，用该名称代替 `git`。

```
prompt> sudo adduser --shell /bin/sh \
           --group \
           --disabled-password \
           --home /srv/example.com/git
git
```

对于熟悉 `adduser` 命令的读者来说，上面例子不言自明。该命令创建了一个名为 `git` 的账户，它的家目录是 `/srv/example.com/git`；不能以它的名义使用输入密码的方式登录服务器；添加一个名为 `git` 的组，并把 shell 类型设置为 `/bin/sh`。

设置好了 Gitosis 账户，以及管理员 SSH 证书后，现在就可以初始化 Gitosis 了。

## 11.5 初始化 Gitosis

### Initializing Gitosis

使用 Gitosis 中的 `gitosis-init` 程序初始化 Gitosis。`gitosis-init` 无需参数，但它要运行于上一节中创建的账户下。如果该账户名是 `git`，则初始化命令如下：<sup>2</sup>

```
prompt> sudo -H -u git gitosis-init < /path/to/id_rsa.pub
Initialized empty Git repository in
/srv/example.com/repositories/gitosis-admin.git/
Reinitialized existing Git repository in
/srv/example.com/repositories/gitosis-admin.git/
```

现在已经初始化了 Gitosis 和 Gitosis 配置版本库，接下来将介绍 Gitosis 配置。如果在本节介绍的初始化过程中出现了 `OSError`，就必须继续阅读本节下面几段的内容，否则可以跳过，直接阅读下一节。

如果 Git 的安装路径不在该账户的 `$PATH` 环境变量里，那么会出现错误信息。这是因为 `gitosis-init` 命令找不到 Git。

要解决这个问题，必须让 `git` 账户能够使用 Git 命令。为此，可以修改 `git` 账户的 `$PATH` 环境变量，也可以在 `$PATH` 环境变量所包含的某个路径中，创建指向 `git` 运行程序的符号链接。

## 11.6 配置 Gitosis

### Configuring Gitosis

现在克隆 Gitosis 配置版本库到管理员本地目录中。Gitosis 配置版本库存储 Gitosis 所管理的各个版本库的配置信息。管理员必须在刚才生成 SSH 公共密钥的计算机上执行克隆操作，并把这个 Gitosis 配置版本库从 Gitosis 服务器上克隆到本地（以便修改）。此时，这个 SSH 公共密钥是 Gitosis 知道的唯一公共密钥。（但 Gitosis 很快就会存储更多用户的公共密钥了，见下文。）

如果服务器的域名是 `example.com`，那么克隆命令如下：

```
prompt> git clone git@example.com:gitosis-admin.git .
Initialized empty Git repository in /work/gitosis-admin/.git/
remote: Counting objects: 5, done.
remote: Compressing objects; 100% (4/4), done.
remote: Total 5 (delta 1), reused 5 (delta 1)
Receiving objects: 100% (5/5), done.
Resolving deltas: 100% (1/1), done.
```

运行上面的命令后，相应的 `gitosis.conf` 文件和 `keydir` 目录将会出现在刚才克隆得到的版本库里。

---

<sup>2</sup> 注意，如果按 Gitosis 约定设置 `git` 账户的家目录为 `/srv/example.com/git`，则 Gitosis 配置版本库 `gitosis-admin.git` 应位于 `/srv/example.com/git` 目录，而非 `/srv/example.com/repositories` 目录。下同。—译者注

`gitosis.conf` 是 INI 格式的文件。该文件以中括号来区分不同段落，并且用“名称=值”的格式记录信息。默认配置如下：

```
[gitosis]
[group gitosis-admin]
writable = gitosis-admin
members = travis
```

看看`[group gitosis-admin]`后面的部分。它的第一行说明该用户组对 Gitosis 的配置版本库 `gitosis-admin` 有写权限；第二行说明 `travis`（也就是本书作者）是这个用户组的成员，准确地讲是唯一成员。

一个用户组可以有多个成员，成员之间可用空格分隔。每个成员都需要在 `keydir` 目录下有一个对应的公共密钥文件，该文件的文件名是成员名加上`.pub` 后缀。例如，成员 `travis` 对应的文件是 `keydir/travis.pub`。

如果要添加新用户，就须获得这些用户的 SSH 公共密钥。每个希望访问 Gitosis 管理下的版本库的用户，都要为它生成密钥。可使用前面介绍过的命令生成该密钥，但请注意，必须在该用户平时所使用的计算机上运行该命令。每个密钥只对应一台特定计算机。

获得公共密钥，并将其添加到 `keydir` 目录后，再修改 `gitosis.conf` 配置文件，把用户名（通过用户组）与允许其访问的版本库相关联。当你在本地的 Gitosis 配置版本库中提交了这一修改后，还要把这些提交使用命令 `git push` 推入 Gitosis 服务器上（Gitosis 配置版本库中才会触发相应脚本，使配置生效）。

## 11.7 添加新版本库

Adding New Repositories

Gitosis 最初只创建了一个版本库，也就是它的配置版本库 `gitosis-admin`。之后，可以把其他任意多个版本库纳入 Gitosis 管理之下。尽管可以在配置文件 `gitosis.conf` 的 `gitosis-admin` 组下添加更多的版本库，并给予写权限，但并不推荐，因为这意味着任何访问其他普通版本库的用户也可以访问 Gitosis 的配置版本库 `gitosis-admin`。更安全的解决方案是，创建两个用户组：一个专用于管理，另一个供开发人员日常使用。下面是一个简单的示例：

```
[gitosis]
[group gitosis-admin]
writable = gitosis-admin
members = travis
```

```
[group team]
writable = mysite
members = travis susannah
```

认得出前面创建的第一个用户组吗？这里添加另一个名为 `team` 的用户组，赋予该组对 `mysite` 版本库的写权限。接着添加 `travis` 和 `susannah` 为该组成员，因此他们就可以修改 `mysite` 版本库了。

当新创建版本库时，要在配置文件中新添加一行来设置该版本库为可写。再次提醒，对配置文件的修改要提交到本地 `gitosis-admin` 版本库中，并推入 Gitosis 服务器上的远程 `gitosis-admin` 版本库。

可将任意本地版本库中的内容传送到新创建的远程版本库中。当然，本地版本库中应包含至少一条提交，否则无内容可送。在本地版本库的工作目录树中，使用命令在本地版本库中为远程版本库创建一个别名，以方便引用，命令如下所示：

```
prompt> git remote add origin git@example.com:mysite.git
```

该命令将名为 `mysite` 的远程版本库添加到本地版本库的配置中，并将它称为 `origin`。显然，实际使用时，需要将 `example.com` 换成真实的服务器域名，并将 `mysite.git` 换成真实的版本库名称。

接下来只剩下将本地版本库内容推入到远程版本库中了。命令格式很简单，但要注意，在命令中要指出打算推入的分支的名字，因为远程版本库中还是空的，尚无该分支存在。

```
prompt> git push origin master
Initialized empty Git repository in
    /srv/example.com/git/repositories/mysite.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 214 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@example.com:test.git
 * [new branch]  master -> master
```

现在已完成了。其他对该版本库有写权限的用户已经可以克隆该远程版本库，并在完成代码修改后使用推入命令送回变更了。

一般来说，开源项目的版本库应该能够公开访问，以方便读取开放源代码。另一方面，公司内部的版本库也可能要对内开放读取权限，而无须了解具体是哪个员工何时读取了版本库的内容。

使用 Git 后台程序 `git daemon` 可支持匿名只读访问，将在下节详细介绍。

## 11.8 设置公共版本库

### Setting Up a Public Repository

最后一项配置任务是为版本库设置（匿名）只读访问入口。只有当希望其他开发人员能够看到某个版本库中的内容，而又不想为他们逐个添加访问账户时，才需要为该版本库设置（匿名）只读访问入口。这在开源软件开发中是最常用的方法，以便任何人都可以匿名读取版本库中的内容。

这种设置只需要两步：首先，把该版本库纳入 Gitosis 的管理中。其次，运行 Git 后台程序 `git daemon` 来支持匿名只读访问。

具体来讲，在第一步中，须要编辑 `gitosis.conf` 配置文件。在该文件中添加一个新的段落，其名称为 `repo` 再加上版本库名称。例如，如果版本库名为 `web`，则须添加：

```
[repo web]
```

表示在 INI 文件格式中创建了一个新的段落，该行下面的内容都属于此段落。接着，添加一行 `daemon = yes`，这样 Git 后台程序就知道此版本库被设置为允许匿名只读访问，如下所示：

```
[repo web]
daemon = yes
```

这就是全部配置步骤。完成后，提交该配置文件的修改，并推入 Gitosis 服务器。接下来设置 Git 后台程序。

使用前面创建的 `git` 账户运行命令 `git daemon`。最简单的方法是在 `git daemon` 命令前加上 `sudo -u git`。此外，`git daemon` 还需要`--basepath` 参数，该参数指出版本库所在的目录，比如`/srv/example.com/repositories/`。

运行 `git daemon` 命令如下：

```
prompt> sudo -u git git daemon \
--base-path /srv/example.com/repositories/
```

可以克隆一个有匿名只读访问入口的版本库来验证前面的命令运行是否生效。例如，克隆名为 `web` 的版本库的命令，如下所示：

```
prompt> git clone git://example.com/web.git
```

设置完毕，但请注意，Git 后台程序只在用户尚未退出登录时运转。另外，当服务器重启时，它也不会自动运行。该问题的解决方法随操作系统的不同而有所不同。

如果想让 Git 后台程序在用户退出登录后仍然运行，但不需要它在系统重启时自动启动，则可使用如下命令：

```
prompt> nohup sudo -u git git daemon \
--base-path /srv/example.com/repositories &
```

命令结尾的`&`符号使得该命令脱离当前进程而运行于单独的进程。`nohup` 前缀则表示该命令脱离当前用户而运行。

最后要提醒的是：虽然 Gitosis 很稳定，但它仍是新生事物，就好像 Git 一样。作为我个人经验，使用 Gitosis 管理版本库一年，没有出现过任何稳定性方面的问题。

由于 Gitosis 与 Git 密切相关，不同的 Git 版本和 Gitosis 版本间可能存在兼容性问题。其中之一是，当 Git 的版本从 1.5.x 到 1.6.0 时，出现了与 Gitosis 不兼容的问题。这一问题已经解决了。但以后仍有可能出现类似问题。

如果你在安装 Gitosis 时遇到问题，可以试着登录本书网上论坛<sup>3</sup>。若你喜欢 IRC，也可以登录 Freenode 上的`#git` 频道，那里有一些对 Gitosis 有经验的人，包括 Gitosis 的开发者 Tv。

## 11.9 结束语

Closing Thoughts

本书正文部分到这里就要结束了。稍后附录里给出了一些补充说明信息。附录 A（第 155 页）列出了 Git 常用命令。附录 B（第 165 页）给出了若干 Git 相关的工具和资源。最后，附录 C（第 173 页）列出了本书的所有参考书目。

<sup>3</sup> <http://forums.pragprog.com/forums/64>

作为笔者，我期待本书带给大家愉快的阅读体验，如同我写作此书时的心情。在本书写作过程中，我学到了很多东西，获得了很多乐趣，期待这些都体现在了本书中。如果你有任何相关问题，请到本书的网上论坛<sup>4</sup>逛逛，在那儿给我留个小纸条。

---

<sup>4</sup> <http://forums.pragprog.com/forums/64>



## 第4篇 附录





# 附录 A

## Git 命令快速参考

Git Command Quick Reference

本附录为 Git 常见命令快速参考。每节介绍一种操作类型。

这里会列出很多命令，而相应的解释却不多。对于还不熟悉 Git 的读者，可回头翻阅第 1 章“Git 的版本控制之道”（第 3 页）。

### A.1 安装和初始化

Setup and Initialization

在使用 Git 之前，须要先进行配置。在使用一个新的版本库之前，须要先初始化。本节介绍与 Git 设置和初始化相关的命令。

#### 配置全局用户名和电子邮件地址

```
prompt> git config --global user.name "Your Name"  
prompt> git config --global user.email "you@example.com"
```

#### 为特定的版本库配置用户名和电子邮件地址

注意：你可以为每个版本库单独设置用户名和邮件地址。这使得用户可在不同项目中使用不同的用户名和/或不同的邮件地址。

```
prompt> cd /path/to/repo  
prompt> git config user.name "Your Name"  
prompt> git config user.email "you@example.com"
```

#### 在命令行中使用不同颜色显示不同内容

```
prompt> git config --global color.ui "auto"
```

### 初始化新版本库

```

prompt> mkdir /path/to/repo
prompt> cd /path/to/repo
prompt> git init
Initialized empty Git repository in /path/to/repo/.git/
prompt>
... create file(s) for first commit ...
prompt> git add .
prompt> git commit -m 'initial import'
Created initial commit bdebe5c: initial import
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 <some file>
```

### 克隆版本库

```

prompt> git clone <repository url>
Initialize repo/.git
Initialized empty Git repository in /work/<remote repository>/.git/
```

### 将目录中的内容纳入 Git 版本控制

```

prompt> cd /path/to/existing/directory
prompt> git init
Initialized empty Git repository in /path/to/existing/directory/.git/
prompt> git add .
prompt> git commit -m "initial import of some project"
```

### 在本地版本库中设置远程版本库的别名

```

... from within the repository directory ...
prompt> git remote add <remote repository> <repository url>
```

## A.2 日常操作

本节列出 Git 日常操作命令，对应于第 4 章“添加与提交：Git 基础”（第 41 页）。

### 添加新文件或暂存已有文件上的改动，然后提交

```

prompt> git add <some file>
prompt> git commit -m "<some message>"
```

### 暂存已有文件上的部分修改

注意： [...] 表示可选参数。

```

prompt> git add -p [<some file> [<some file> [and so on]]]
选择要提交的文本块.....
```

使用交互方式添加文件

```
prompt> git add -i
```

暂存已纳入 Git 版本控制之下的文件的修改

```
prompt> git add -u [<some path> [<some path>]]
```

提交已纳入 Git 版本控制之下的文件的所有修改

```
prompt> git commit -m "<some message>" -a
```

清除工作目录树中的修改

```
prompt> git checkout HEAD <some file> [<some file>]
```

取消已暂存但尚未提交的修改的暂存标识

```
prompt> git reset HEAD <some file> [<some file>]
```

修复上一次提交中的问题

改动相关文件，并暂存……

```
prompt> git commit -m "<some message>" --amend
```

修复上一次提交中的问题，并复用上次的提交注释

```
prompt> git commit -C HEAD --amend
```

## A.3 分支

分支是 Git 的强项之一。本节介绍关于分支的各个命令。详细内容见第 5 章“理解和使用分支”（第 55 页）。

列出本地分支

```
prompt> git branch
```

列出远程分支

```
prompt> git branch -r
```

列出所有分支

```
prompt> git branch -a
```

基于当前分支（的末梢）创建新分支

```
prompt> git branch <new branch>
```

### 检出另一条分支

```
prompt> git checkout <some branch>
```

### 基于当前分支创建新分支，同时检出该分支

```
prompt> git checkout -b <new branch>
```

### 基于另一个起点，创建新分支

你可以从版本库中的任何一个版本开始创建新分支。这个起始点可以用一条已有的分支名称、一个提交名称，或者一个标签名称来表达。

```
prompt> git branch <new branch> <start point>
```

### 创建同名新分支，覆盖已有分支

```
prompt> git branch -f <some existing branch> [<start point>]
```

### 移动或重命名分支

#### 只有当<new branch>不存在时

```
prompt> git checkout -m <existing branch name> <new branch name>
```

#### 如果<new branch>已存在，就覆盖它

```
prompt> git checkout -M <existing branch name> <new branch name>
```

### 把另一条分支合并到当前分支

```
prompt> git merge <some branch>
```

### 合并，但不提交

```
prompt> git merge --no-commit <some branch>
```

### 拣选合并，并且提交

```
prompt> git cherry-pick <commit name>
```

### 拣选合并，但不提交

```
prompt> git cherry-pick -n <commit name>
```

### 把一条分支上的内容压合到另一条分支（上的一个提交）

```
prompt> git merge --squash <some branch>
```

### 删除分支

#### 仅当欲删除的分支已合并到当前分支时

```
prompt> git branch -d <branch to delete>
```

不论欲删除的分支是否已合并到当前分支  
**prompt> git branch -D <branch to delete>**

## A.4 历史

### History

这些命令用来显示版本库的历史信息，包括代码曾在哪里、谁在何时做了什么、修改的内容及其统计信息。详见第 6 章“查询 Git 历史记录”（第 71 页）。

#### 显示全部历史记录

**prompt> git log**

#### 显示版本历史，以及版本间的内容差异

**prompt> git log -p**

#### 只显示最近一个提交

**prompt> git log -1**

#### 显示最近的 20 个提交，以及版本间的内容差异

**prompt> git log -20 -p**

#### 显示最近 6 小时的提交

**prompt> git log --since="6 hours"**

#### 显示两天之前的提交

**prompt> git log --before="2 days"**

#### 显示比 HEAD（当前检出分支的末梢）早 3 个提交的那个提交

**prompt> git log -1 HEAD~3**

或者……

**prompt> git log -1 HEAD^^^**

或者……

**prompt> git log -1 HEAD~1^^**

#### 显示两个版本之间的提交

下面命令中的<start point>和<end point>可以是一个提交名称、分支名称、标签名称，或者它们的混合。

**prompt> git log <start point>...<end point>**

显示历史，每个提交显示一行，包括提交注释的第一行

```
prompt> git log --pretty=oneline
```

显示改动行数统计

```
prompt> git log --stat
```

显示改动文件的名称和状态

```
prompt> git log --name-status
```

显示当前工作目录树和暂存区间的差别

```
prompt> git diff
```

显示暂存区和版本库间的差别

```
prompt> git diff --cached
```

显示工作目录树和版本库间的差别

```
prompt> git diff HEAD
```

显示工作目录树与版本库中某次提交版本之间的差别

<start point>可以是一个提交名称、分支名称或标签名称。

```
prompt> git diff <start point>
```

显示版本库中两个版本之间的差别

```
prompt> git diff <start point> <end point>
```

显示差别的相关统计

```
prompt> git diff --stat <start point> [<end point>]
```

显示文件中各个部分的修改者及相关提交信息

```
prompt> git blame <some file>
```

显示文件中各个部分的修改者及相关提交信息，包括在该文件中复制、粘贴和移动内容等方面的情况。

```
prompt> git blame -M <some file>
```

显示文件中各部分的修改者及相关提交信息，包括在文件间移动内容方面的情况

```
prompt> git blame -C -C <some file>
```