

注意，本文来自 <http://www.open-open.com/lib/view/open1328069609436.html>，讲解的非常好，在此处复制粘贴而已，对原作者表示崇高的敬意，所有版权归原作者所有。本人再次表示这些劳动者的分享精神值得推崇，同时表示香蕉很好吃，也表示我的妞很可爱。

起步

本章介绍开始使用 Git 前的相关知识。我们会先了解一些版本控制工具的历史背景，然后试着让 Git 在你的系统上跑起来，直到最后配置好，可以正常开始开发工作。读完本章，你就会明白为什么 Git 会如此流行，为什么你应该立即开始使用它。

1.1 关于版本控制

什么是版本控制？我真的需要吗？版本控制是一种记录若干文件内容变化，以便将来查阅特定版本修订情况的系统。在本书所展示的例子中，我们仅对保存着软件源代码的文本文件作版本控制管理，但实际上，你可以对任何类型的文件进行版本控制。

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本（这或许是你非常渴望拥有的功能）。采用版本控制系统（VCS）是个明智的选择。有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态。你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而导致出现怪异问题，又是谁在何时报告了某个功能缺陷等等。使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改删的删，你也照样可以轻松恢复到原先的样子。但额外增加的工作量却微乎其微。

本地版本控制系统

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。这么做唯一的好处就是简单。不过坏处也不少：有时候会混淆所在的工作目录，一旦弄错文件丢了数据就没法撤销恢复。

为了解决这个问题，人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异（见图 1-1）。

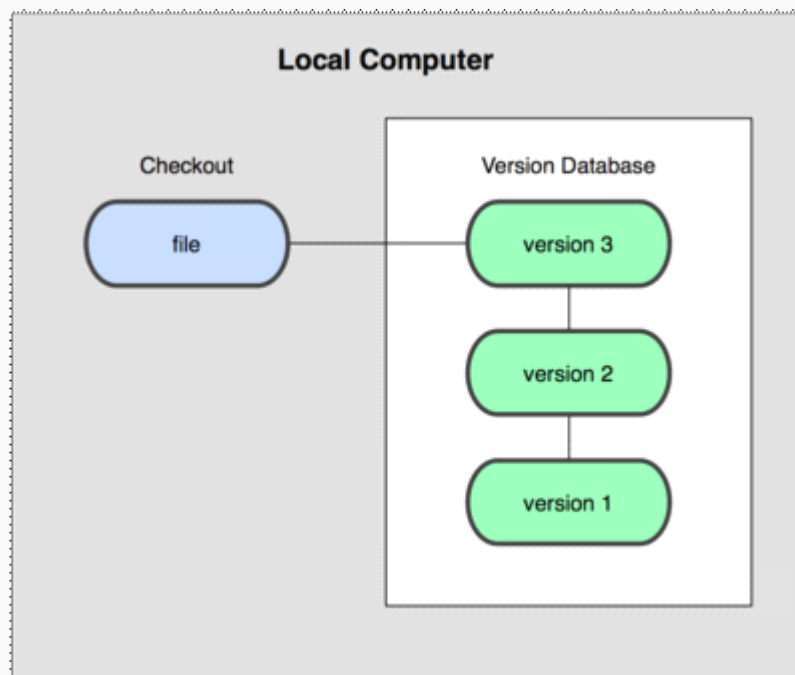


图 1-1. 本地版本控制系统其中最流行的一种叫做 **rcs**，现今许多计算机系统上都还看得到它的踪影。甚至在流行的 **Mac OS X** 系统上安装了开发者工具包之后，也可以使用 **rcs** 命令。它的工作原理基本上就是保存并管理文件补丁（**patch**）。文件补丁是一种特定格式的文本文件，记录着对应文件修订前后的内容变化。所以，根据每次修订后的补丁，**rcs** 可以通过不断打补丁，计算出各个版本的文件内容。

集中化的版本控制系统

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？于是，集中化的版本控制系统（**Centralized Version Control Systems**，简称 **CVCS**）应运而生。这类系统，诸如 **CVS**，**Subversion** 以及 **Perforce** 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。多年以来，这已成为版本控制系统的标准做法（见图 1-2）。

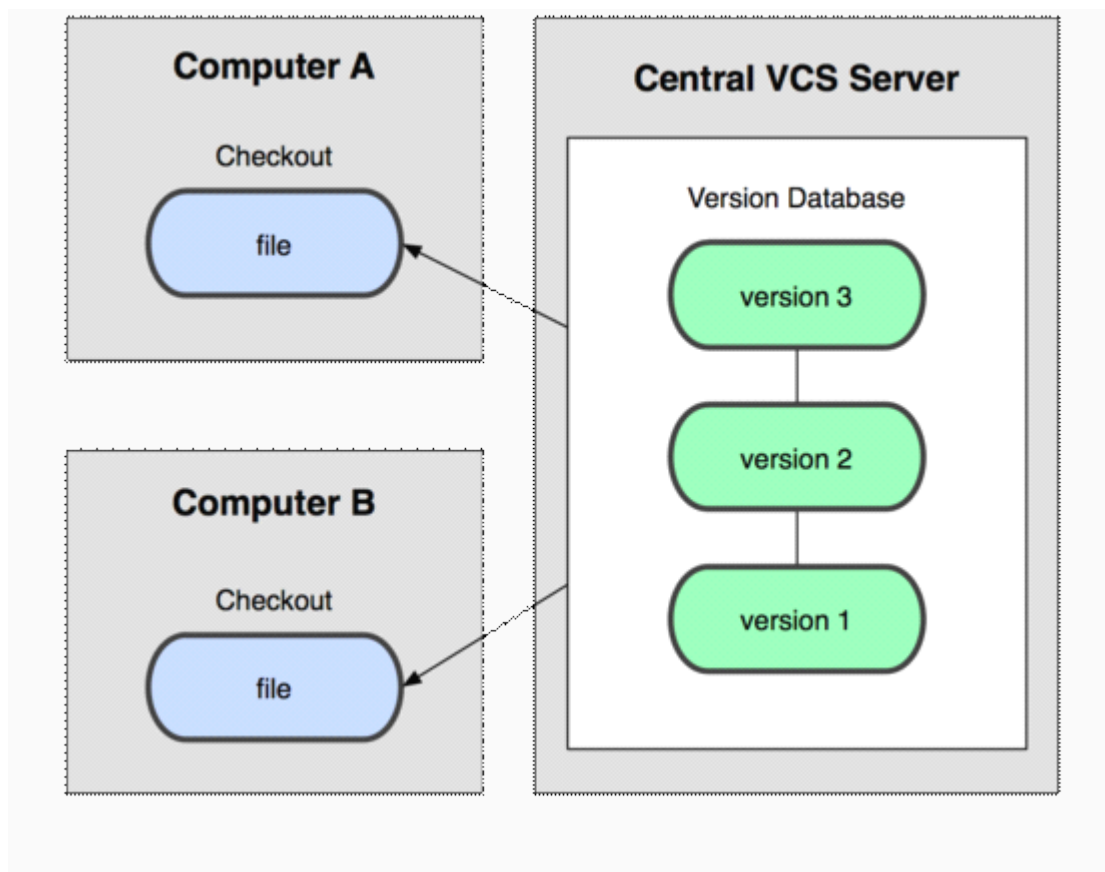


图 1-2. 集中化的版本控制系统这种做法带来了许多好处，特别是相较于老式的本地 VCS 来说。现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么。而管理员也可以轻松掌控每个开发者的权限，并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易。

事分两面，有好有坏。这么做最显而易见的缺点是中央服务器的单点故障。如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。要是中央服务器的磁盘发生故障，碰巧没做备份，或者备份不够及时，就还是会有丢失数据的风险。最坏的情况是彻底丢失整个项目的历史更改记录，而被客户端提取出来的某些快照数据除外，但这样的话依然是个问题，你不能保证所有的数据都已经有人事先完整提取出来过。本地版本控制系统也存在类似问题，只要整个项目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

分布式版本控制系统

于是分布式版本控制系统（Distributed Version Control System，简称 DVCS）面世了。在这类系统中，像 Git，Mercurial，Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份（见图 1-3）。

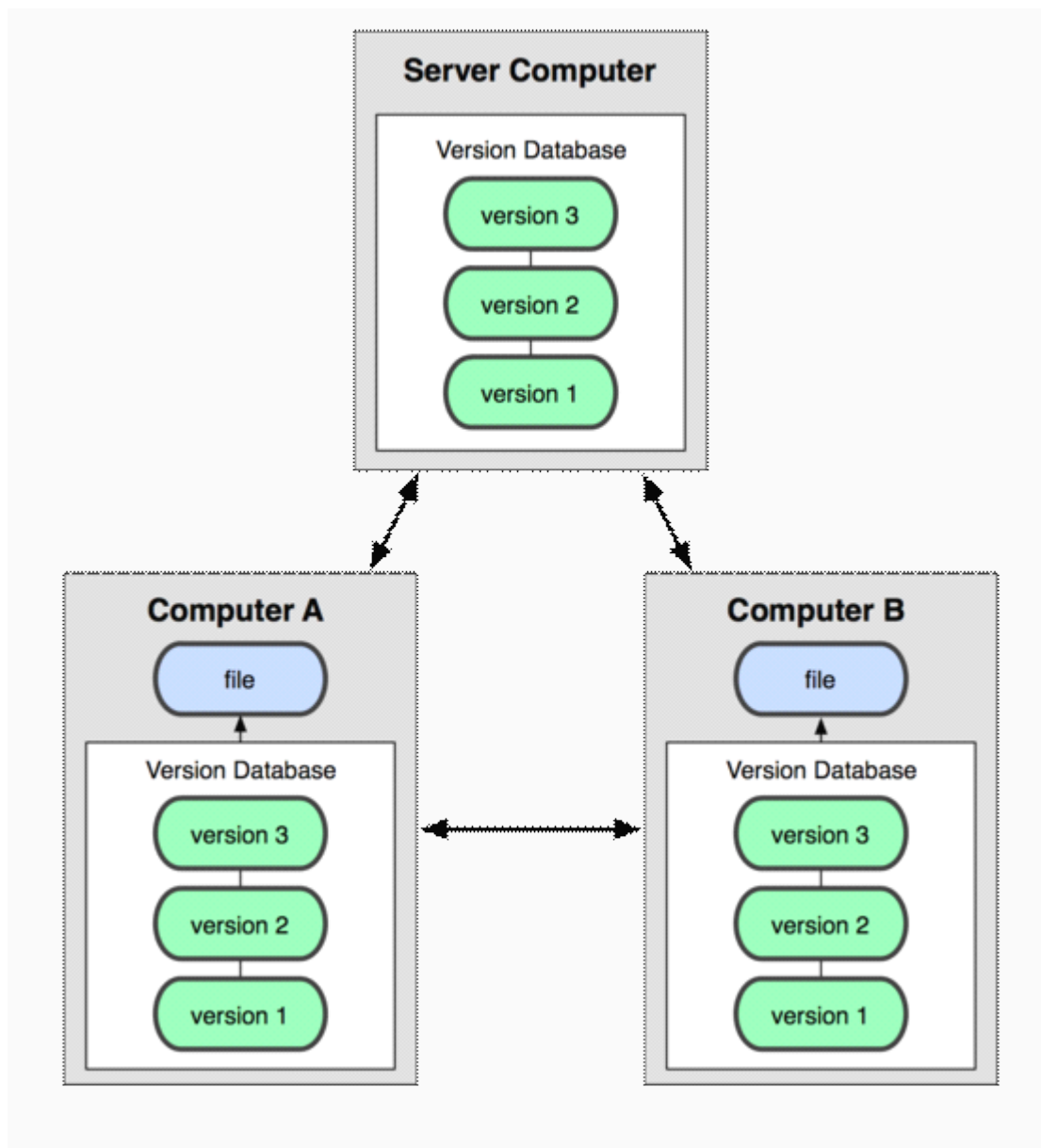


图 1-3. 分布式版本控制系统更进一步，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

1.2 Git 简史

同生活中的许多伟大事件一样，Git 诞生于一个极富纷争大举创新的年代。Linux 内核开源项目有着为数众广的参与者。绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991—2002年间）。到 2002 年，整个项目组开始启用分布式版本控制

系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了免费使用 BitKeeper 的权力。这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。他们对新的系统制订了若干目标：

- * 速度
- * 简单的设计
- * 对非线性开发模式的强力支持（允许上千个并行开发的分支）
- * 完全分布式
- * 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，它还有着令人难以置信的非线性分支管理系统（见第三章），可以应付各种复杂的项目开发需求。

1.3 Git 基础

那么，简单地说，Git 究竟是怎样的一个系统呢？请注意，接下来的内容非常重要，若是理解了 Git 的思想和基本工作原理，用起来就会知其所以然，游刃有余。在开始学习 Git 的时候，请不要尝试把各种概念和其他版本控制系统（诸如 Subversion 和 Perforce 等）相比拟，否则容易混淆每个操作的实际意义。Git 在保存和处理各种信息的时候，虽然操作起来的命令形式非常相近，但它与其他版本控制系统的做法颇为不同。理解这些差异将有助于你准确地使用 Git 提供的各种工具。

直接记录快照，而非差异比较

Git 和其他版本控制系统的主要差别在于，Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。这类系统（CVS，Subversion，Perforce，Bazaar 等等）每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容，请看图 1-4。

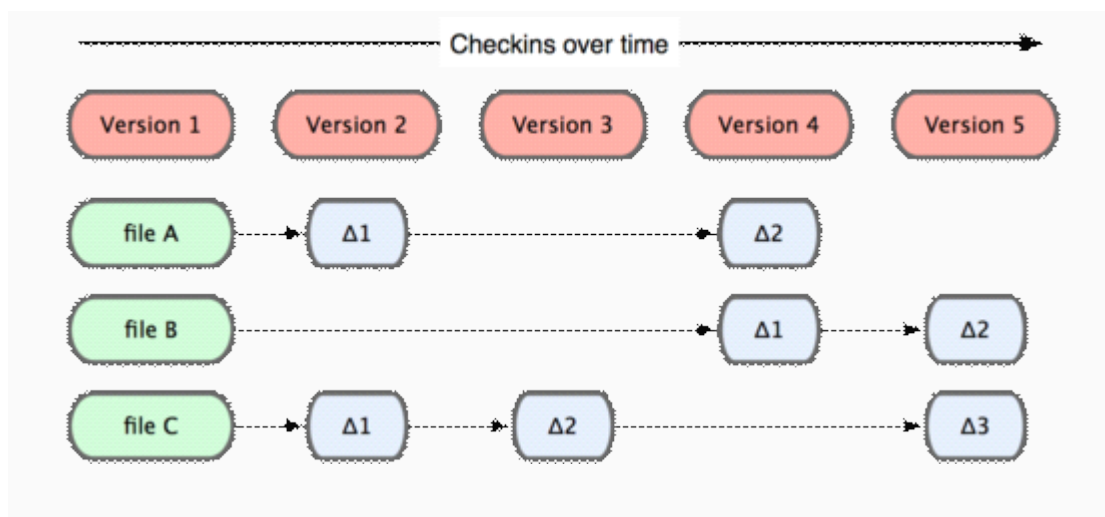


图 1-4. 其他系统在每个版本中记录着各个文件的具体差异 Git 并不保存这些前后变化的差异数据。实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照 的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一链接。Git 的工作方式就像图 1-5 所示。

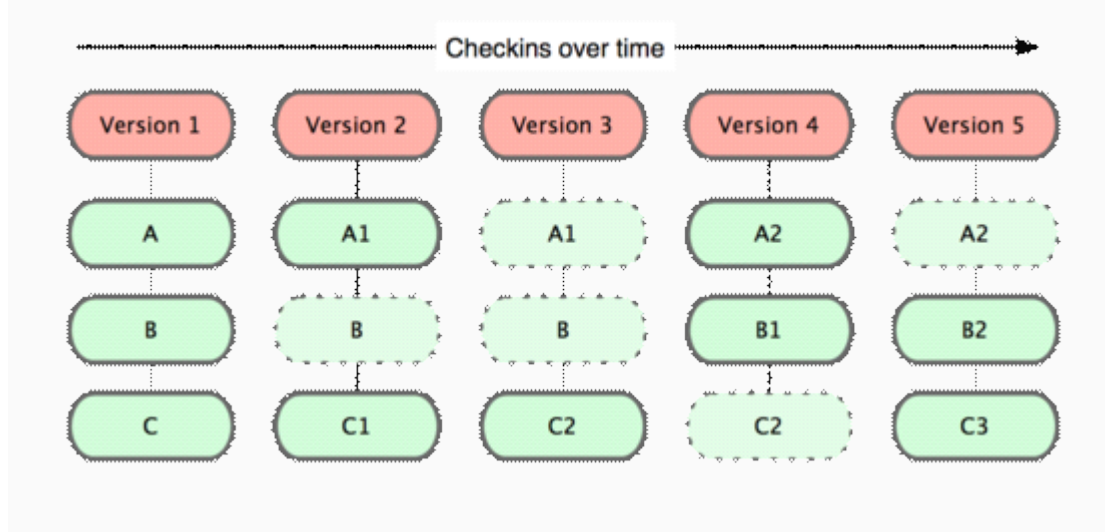


图 1-5. Git 保存每次更新时的文件快照这是 Git 同其他系统的重要区别。它完全颠覆了传统版本控制的套路，并对各个环节的实现方式作了新的设计。Git 更像是个小型的文件系统，但它同时还提供了许多以此为基础的超强工具，而不只是一个简单的 VCS。稍后在第三章讨论 Git 分支管理的时候，我们会再看看这样的设计究竟会带来哪些好处。

近乎所有操作都是本地执行

在 Git 中的绝大多数操作都只需要访问本地文件和资源，不用连网。但如果用 CVCS 的话，差不多所有操作都需要连接网络。因为 Git 在本地磁盘上就保存着所有当前项目的历史更新，所以处理起来速度飞快。

举个例子，如果要浏览项目的历史更新摘要，Git 不用跑到外面的服务器上去取数据回来，

而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。如果想要看当前版本的文件和一个月前的版本之间有何差异，Git 会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。

用 CVCS 的话，没有网络或者断开 VPN 你就无法做任何事情。但用 Git 的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程仓库。同样，在回家的路上，不用连接 VPN 你也可以继续工作。换作其他版本控制系统，这么做几乎不可能，抑或非常麻烦。比如 Perforce，如果不连到服务器，几乎什么都做不了（译注：默认无法发出命令 `p4 edit file` 开始编辑文件，因为 Perforce 需要联网通知系统声明该文件正在被谁修订。但实际上手工修改文件权限可以绕过这个限制，只是完成后还是无法提交更新。）；如果是 Subversion 或 CVS，虽然可以编辑文件，但无法提交更新，因为数据库在网络上。看上去好像这些都不是什么大问题，但实际体验过之后，你就会惊喜地发现，这其实是会带来很大不同的。

时刻保持数据完整性

在保存到 Git 之前，所有数据都要进行内容的校验和（checksum）计算，并将此结果作为数据的唯一标识和索引。换句话说，不可能在你修改了文件或目录之后，Git 一无所知。这项特性作为 Git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git 都能立即察觉。

Git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值，作为指纹字符串。该字符串由 40 个十六进制字符（0-9 及 a-f）组成，看起来就像是：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git 的工作完全依赖于这类指纹字符串，所以你会经常看到这样的哈希值。实际上，所有保存在 Git 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

多数操作仅添加数据

常用的 Git 操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，都会使回退或重现历史版本变得困难重重。在别的 VCS 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 Git 里，一旦提交快照之后就完全不用担心丢失数据，特别是养成定期推送到其他仓库的习惯的话。

这种高可靠性令我们的开发工作安心不少，尽管去做各种试验性的尝试好了，再怎样也不会弄丢数据。至于 Git 内部究竟是如何保存和恢复数据的，我们会在第九章讨论 Git 内部原理时再作详述。

文件的三种状态

好，现在请注意，接下来要讲的概念非常重要。对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。已提交表示该文件已经被安全地保存在本地数据库中；已修改表示修改了某个文件，但还没有提交保存；已暂存表示把已修改的文件放在下次提交时要保存的清单中。

由此我们看到 Git 管理项目时，文件流转的三个工作区域：Git 的工作目录，暂存区域，以及本地仓库。

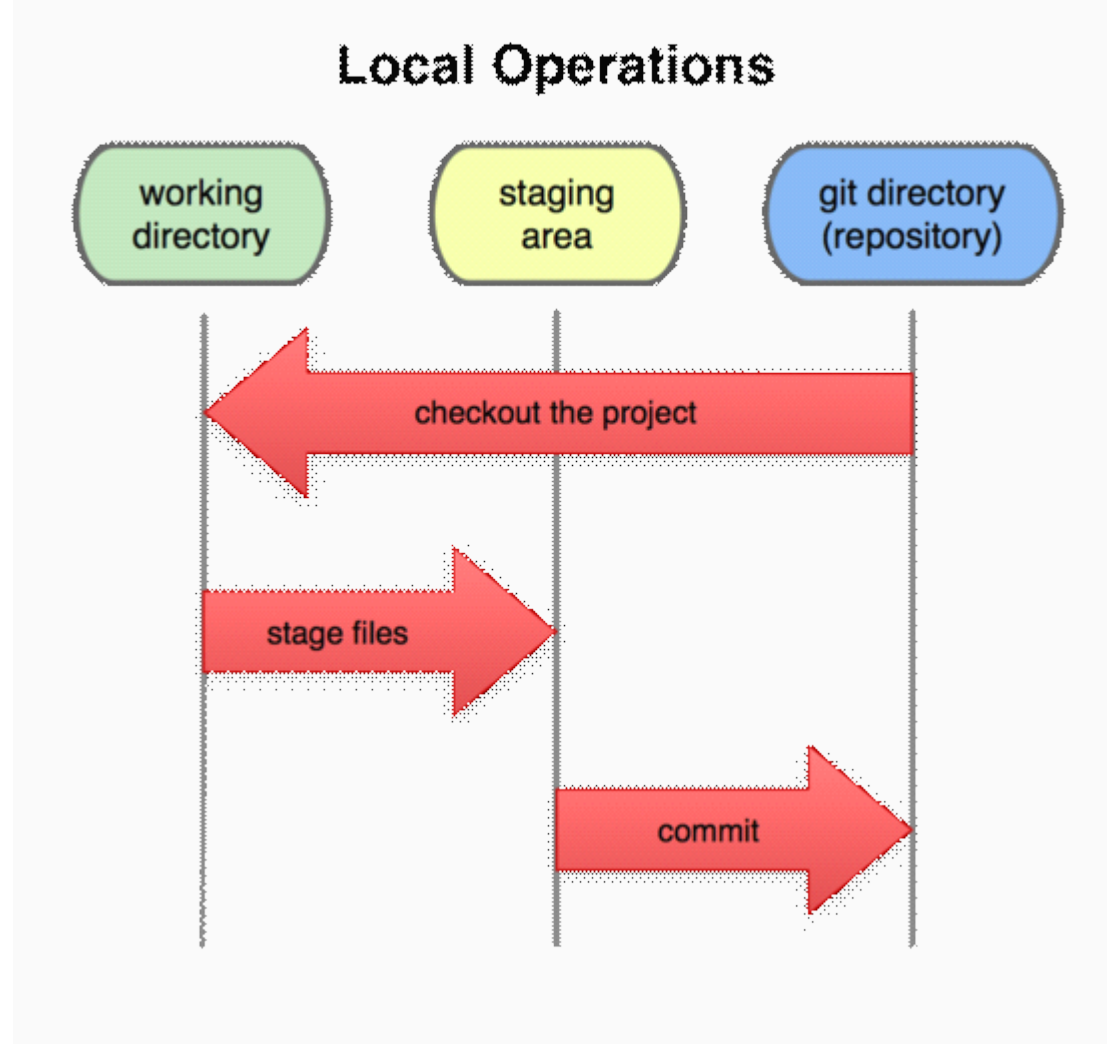


图 1-6. 工作目录，暂存区域，以及本地仓库每个项目都有一个 Git 目录（译注：如果 git clone 出来的话，就是其中 .git 的目录；如果 git clone --bare 的话，新建的目录本身就是 Git 目录。），它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做工作目录。这些文件实际上都是从 Git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

所谓的暂存区域只不过是个简单的文件，一般都放在 Git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改某些文件。2. 对修改后的文件进行快照，然后保存到暂存区域。3. 提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中。

所以，我们可以从文件所处的位置来判断状态：如果是 Git 目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。到第二章的时候，我们会进一步了解其中细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。

1.4 安装 Git

是时候动手尝试下 Git 了，不过得先安装好它。有许多种安装方式，主要分为两种，一种是通过编译源代码来安装；另一种是使用为特定平台预编译好的安装包。

从源代码安装

若是条件允许，从源代码安装有很多好处，至少可以安装最新的版本。Git 的每个版本都在不断尝试改进用户体验，所以能通过源代码自己编译安装最新版本就再好不过了。有些 Linux 版本自带的安装包更新起来并不及时，所以除非你在用最新的 distro 或者 backports，那么从源代码安装其实该算是最佳选择。

Git 的工作需要调用 curl, zlib, openssl, expat, libiconv 等库的代码，所以需要先安装这些依赖工具。在有 yum 的系统上(比如 Fedora)或者有 apt-get 的系统上(比如 Debian 体系)，可以用下面的命令安装：

```
$ yum install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel
```

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
    libz-dev libssl-dev
```

之后，从下面的 Git 官方站点下载最新版本源代码：

```
http://git-scm.com/download
```

然后编译并安装：

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
```

```
$ sudo make prefix=/usr/local install
```

现在已经可以用 `git` 命令了，用 `git` 把 `Git` 项目仓库克隆到本地，以便日后随时更新：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

在 Linux 上安装

如果要在 `Linux` 上安装预编译好的 `Git` 二进制安装包，可以直接用系统提供的包管理工具。在 `Fedora` 上用 `yum` 安装：

```
$ yum install git-core
```

在 `Ubuntu` 这类 `Debian` 体系的系统上，可以用 `apt-get` 安装：

```
$ apt-get install git-core
```

在 Mac 上安装

在 `Mac` 上安装 `Git` 有两种方式。最容易的当属使用图形化的 `Git` 安装工具，界面如图 1-7，下载地址在：

```
http://code.google.com/p/git-osx-installer
```

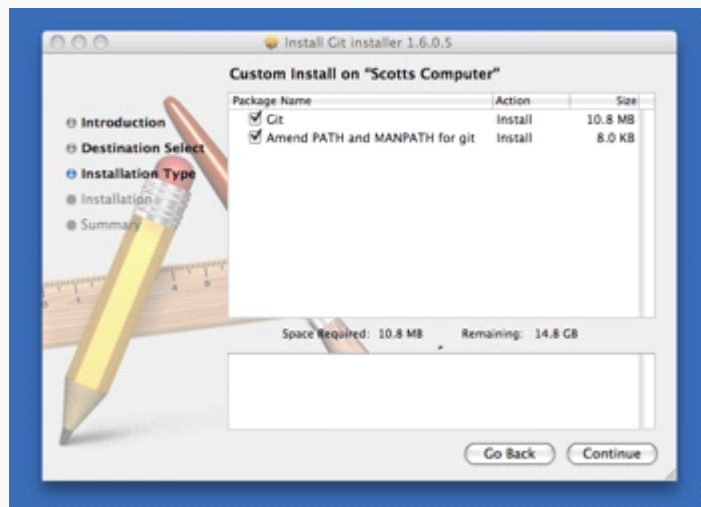


图 1-7. `Git OS X` 安装工具另一种是通过 `MacPorts` (<http://www.macports.org>) 安装。如果已经装好了 `MacPorts`，用下面的命令安装 `Git`：

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

这种方式就不需要再自己安装依赖库了，`Macports` 会帮你搞定这些麻烦事。一般上面列出的安装选项已经够用，要是你想用 `Git` 连接 `Subversion` 的代码仓库，还可以加上 `+svn` 选项，具体将在第八章作介绍。（译注：还有一种是使用 `homebrew`

（<https://github.com/mxcl/homebrew>）：`brew install git`。）

在 Windows 上安装

在 Windows 上安装 Git 同样轻松，有个叫做 `msysGit` 的项目提供了安装包，可以到 [Google Code](http://code.google.com/p/msysgit) 的页面上下载 `exe` 安装文件并运行：

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

完成安装之后，就可以使用命令行的 `git` 工具（已经自带了 `ssh` 客户端）了，另外还有一个图形界面的 `Git` 项目管理工具。

1.5 初次运行 Git 前的配置

一般新的系统上，我们都需要先配置下自己的 `Git` 工作环境。配置工作只需一次，以后升级时还会沿用现在的配置。当然，如果需要，你随时可以用相同的命令修改已有的配置。

`Git` 提供了一个叫做 `git config` 的工具（译注：实际是 `git-config` 命令，只不过可以通过 `git` 加一个名字来呼叫此命令。），专门用来配置或读取相应的工作环境变量。而正是由这些环境变量，决定了 `Git` 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

- `/etc/gitconfig` 文件：系统中对所有用户都普遍适用的配置。若使用 `git config` 时用 `--system` 选项，读写的就是这个文件。
- `~/.gitconfig` 文件：用户目录下的配置文件只适用于该用户。若使用 `git config` 时用 `--global` 选项，读写的就是这个文件。
- 当前项目的 `git` 目录中的配置文件（也就是工作目录中的 `.git/config` 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 `.git/config` 里的配置会覆盖 `/etc/gitconfig` 中的同名变量。

在 `Windows` 系统上，`Git` 会找寻用户主目录下的 `.gitconfig` 文件。主目录即 `$HOME` 变量指定的目录，一般都是 `C:\Documents and Settings\%USER%`。此外，`Git` 还会尝试找寻 `/etc/gitconfig` 文件，只不过看当初 `Git` 装在什么目录，就以此作为根目录来定位。

用户信息

第一个要配置的是你个人的用户名称和电子邮件地址。这两条配置很重要，每次 `Git` 提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

如果用了 `--global` 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所

有的项目都会默认使用这里配置的用户信息。如果要在某个特定的项目中使用其他名字或者电邮，只要去掉`--global` 选项重新配置即可，新的设定保存在当前项目的`.git/config` 文件里。

文本编辑器

接下来要设置的是默认使用的文本编辑器。`Git` 需要你输入一些额外消息的时候，会自动调用一个外部文本编辑器给你用。默认会使用操作系统指定的默认编辑器，一般可能会是 `Vi` 或者 `Vim`。如果你有其他偏好，比如 `Emacs` 的话，可以重新设置：

```
$ git config --global core.editor emacs
```

差异分析工具

还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。比如要改用 `vimdiff` 的话：

```
$ git config --global merge.tool vimdiff
```

`Git` 可以理解 `kdiff3`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff`, `gvimdiff`, `ecmerge`, 和 `opendiff` 等合并工具的输出信息。当然，你也可以指定使用自己开发的工具，具体怎么做可以参阅第七章。

查看配置信息

要检查已有的配置信息，可以使用 `git config --list` 命令：

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 `/etc/gitconfig` 和 `~/.gitconfig`），不过最终 `Git` 实际采用的是最后一个。

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

```
$ git config user.name
Scott Chacon
```

1.6 获取帮助

想了解 `Git` 的各式工具该怎么用，可以阅读它们的使用帮助，方法有三：

```
$ git help
```

```
$ git --help $ man git-
```

比如，要学习 **config** 命令可以怎么用，运行：

```
$ git help config
```

我们随时都可以浏览这些帮助信息而无需连网。不过，要是你觉得还不够，可以到 **Freenode** IRC 服务器（irc.freenode.net）上的 **#git** 或 **#github** 频道寻求他人帮助。这两个频道上总有着上百号人，大多都有着丰富的 **git** 知识，并且乐于助人。

1.7 小结

至此，你该对 **Git** 有了点基本认识，包括它和以前你使用的 **CVCS** 之间的差别。现在，在你的系统上应该已经装好了 **Git**，设置了自己的名字和电邮。接下来让我们继续学习 **Git** 的基础知识。

Git 基础

读完本章你就能上手使用 Git 了。本章将介绍几个最基本的，也是最常用的 Git 命令，以后绝大多数时间里用到的也就是这几个命令。读完本章，你就能初始化一个新的代码仓库，做一些适当配置；开始或停止跟踪某些文件；暂存或提交某些更新。我们还会展示如何让 Git 忽略某些文件，或是名称符合特定模式的文件；如何既快且容易地撤消犯下的小错误；如何浏览项目的更新历史，查看某两次更新之间的差异；以及如何从远程仓库拉数据下来或者推数据上去。

2.1 取得项目的 Git 仓库

有两种取得 Git 项目仓库的方法。第一种是在现存的目录下，通过导入所有文件来创建新的 Git 仓库。第二种是从已有的 Git 仓库克隆出一个新的镜像仓库来。

在工作目录中初始化新仓库

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：

```
$ git init
```

初始化后，在当前目录下会出现一个名为 `.git` 的目录，所有 Git 需要的数据和资源都存放在这个目录中。不过目前，仅仅是按照既有的结构框架初始化好了里边所有的文件和目录，但我们还没有开始跟踪管理项目中的任何一个文件。（在第九章我们会详细说明刚才创建的 `.git` 目录中究竟有哪些文件，以及都起些什么作用。）

如果当前目录下有几个文件想要纳入版本控制，需要先用 `git add` 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

稍后我们再逐一解释每条命令的意思。不过现在，你已经得到了一个实际维护着若干文件的 Git 仓库。

从现有仓库克隆

如果想对某个开源项目出一份力，可以先把该项目的 Git 仓库复制一份出来，这就需要用到 `git clone` 命令。如果你熟悉其他的 VCS 比如 Subversion，你可能已经注意到这里使用的是 `clone` 而不是 `checkout`。这是个非常重要的差别，Git 收取的是项目历史的所有数据（每一个文件的每一个版本），服务器上有的数据克隆之后本地也都有了。实际上，即便

服务器的磁盘发生故障，用任何一个克隆出来的客户端都可以重建服务器上的仓库，回到当初克隆时的状态(虽然可能会丢失某些服务器端的挂钩设置，但所有版本的数据仍旧还在，有关细节请参考第四章)。

克隆仓库的命令格式为 `git clone [url]`。比如，要克隆 Ruby 语言的 Git 代码仓库 Grit，可以用下面的命令：

```
$ git clone git://github.com/schacon/grit.git
```

这会在当前目录下创建一个名为“grit”的目录，其中包含一个 `.git` 的目录，用于保存下载下来的所有版本记录，然后从中取出最新版本的文件拷贝。如果进入这个新建的 `grit` 目录，你会看到项目中的所有文件已经在里边了，准备好后续的开发和使用。如果希望在克隆的时候，自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字：

```
$ git clone git://github.com/schacon/grit.git mygrit
```

唯一的差别就是，现在新建的目录成了 `mygrit`，其他的都和上边的一样。

Git 支持许多数据传输协议。之前的例子使用的是 `git://` 协议，不过你也可以用 `http(s)://` 或者 `user@server:/path.git` 表示的 SSH 传输协议。我们会在第四章详细介绍所有这些协议在服务器端该如何配置使用，以及各种方式之间的利弊。

2.2 记录每次更新到仓库

现在我们手上已经有了一个真实项目的 Git 仓库，并从这个仓库中取出了所有文件的工作拷贝。接下来，对这些文件作些修改，在完成了一个阶段的目标之后，提交本次更新到仓库。

请记住，工作目录下面的所有文件都不外乎这两种状态：已跟踪或未跟踪。已跟踪的文件是指本来就被纳入版本控制管理的文件，在上次快照中有它们的记录，工作一段时间后，它们的状态可能是未更新，已修改或者已放入暂存区。而所有其他文件都属于未跟踪文件。它们既没有上次更新时的快照，也不在当前的暂存区域。初次克隆某个仓库时，工作目录中的所有文件都属于已跟踪文件，且状态为未修改。

在编辑过某些文件之后，Git 将这些文件标为已修改。我们逐步把这些修改过的文件放到暂存区域，直到最后一次性提交所有这些暂存起来的文件，如此重复。所以使用 Git 时的文件状态变化周期如图 2-1 所示。

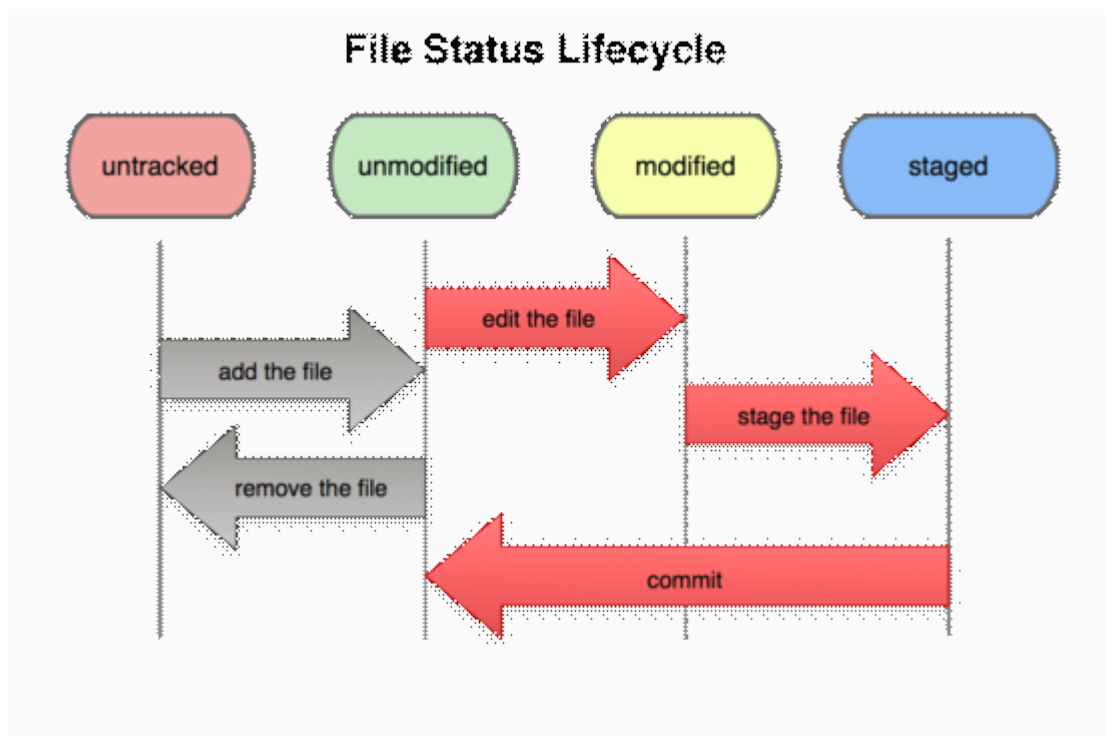


图 2-1. 文件的状态变化周期

要确定哪些文件当前处于什么状态，可以用 `git status` 命令。如果在克隆仓库之后立即执行此命令，会看到类似这样的输出：

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

这说明你当前的工作目录相当干净。换句话说，当前没有任何跟踪着的文件，也没有任何文件在上次提交后更改过。此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪的新文件，否则 `Git` 会在这里列出来。最后，该命令还显示了当前所在的分支是 `master`，这是默认的分支名称，实际是可以修改的，现在先不用考虑。下一章我们会详细讨论分支和引用。

现在让我们用 `vim` 编辑一个新文件 `README`，保存退出后运行 `git status` 会看到该文件出现在未跟踪文件列表中：

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add
```

```
..." to include in what will be committed) # # README nothing added to commit  
but untracked files present (use "git add" to track)
```

就是在“Untracked files”这行下面。Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，因而不用担心把临时文件什么的也归入版本管理。不过现在的例子中，我们确实想要跟踪管理 README 这个文件。

跟踪新文件

使用命令 `git add` 开始跟踪一个新文件。所以，要跟踪 README 文件，运行：

```
$ git add README
```

此时再运行 `git status` 命令，会看到 README 文件已被跟踪，并处于暂存状态：

```
$ git status  
  
# On branch master  
  
# Changes to be committed:  
  
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README #
```

只要在“Changes to be committed”这行下面的，就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。你可能会想起之前我们使用 `git init` 后就运行了 `git add` 命令，开始跟踪当前目录下的文件。在 `git add` 后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下的所有文件。（译注：其实 `git add` 的潜台词就是把目标文件快照放入暂存区域，也就是 `add file into staged area`，同时未曾跟踪过的文件标记为需要跟踪。这样就好理解后续 `add` 操作的实际意义了。）

暂存已修改文件

现在我们修改下之前已跟踪过的文件 `benchmarks.rb`，然后再次运行 `status` 命令，会看到这样的状态报告：

```
$ git status  
  
# On branch master  
  
# Changes to be committed:  
  
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README # # Changed but not updated: # (use "git
add ..." to update what will be committed) # # modified: benchmarks.rb #
```

文件 **benchmarks.rb** 出现在“**Changed but not updated**”这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行 **git add** 命令（这是个多功能命令，根据目标文件的状态不同，此命令的效果也不同：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等）。现在让我们运行 **git add** 将 **benchmarks.rb** 放到暂存区，然后再看看 **git status** 的输出：

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README # modified: benchmarks.rb #
```

现在两个文件都已暂存，下次提交时就会一并记录到仓库。假设此时，你想要在 **benchmarks.rb** 里再加条注释，重新编辑存盘后，准备好提交。不过且慢，再运行 **git status** 看看：

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README # modified: benchmarks.rb # # Changed
but not updated: # (use "git add ..." to update what will be committed) # # modified:
benchmarks.rb #
```

怎么回事？**benchmarks.rb** 文件出现了两次！一次算未暂存，一次算已暂存，这怎么可能呢？好吧，实际上 **Git** 只不过暂存了你运行 **git add** 命令时的版本，如果现在提交，那么提交的是添加注释前的版本，而非当前工作目录中的版本。所以，运行了 **git add** 之后又作了修订的文件，需要重新运行 **git add** 把最新版本重新暂存起来：

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README # modified: benchmarks.rb #
```

忽略某些文件

一般我们总会有些文件无需纳入 **Git** 的管理，也不希望它们总出现在未跟踪文件列表。通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。我们可以创建一个名为 `.gitignore` 的文件，列出要忽略的文件模式。来看一个实际的例子：

```
$ cat .gitignore
*.o
*.a
*~
```

第一行告诉 **Git** 忽略所有以 `.o` 或 `.a` 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的，我们用不着跟踪它们的版本。第二行告诉 **Git** 忽略所有以波浪符（`~`）结尾的文件，许多文本编辑软件（比如 **Emacs**）都用这样的文件名保存副本。此外，你可能还需要忽略 `log`，`tmp` 或者 `pid` 目录，以及自动生成的文档等等。要养成一开始就设置好 `.gitignore` 文件的习惯，以免将来误提交这类无用的文件。

文件 `.gitignore` 的格式规范如下：

- 所有空行或者以注释符号 `#` 开头的行都会被 **Git** 忽略。
- 可以使用标准的 **glob** 模式匹配。* 匹配模式最后跟反斜杠（`/`）说明要忽略的是目录。* 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（`!`）取反。

所谓的 **glob** 模式是指 **shell** 所使用的简化了的正则表达式。星号（`*`）匹配零个或多个任意字符；`[abc]` 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 `a`，要么匹配一个 `b`，要么匹配一个 `c`）；问号（`?`）只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如`[0-9]` 表示匹配所有 `0` 到 `9` 的数字）。

我们再看一个 `.gitignore` 文件的例子：

```
# 此为注释 - 将被 Git 忽略
*.a          # 忽略所有 .a 结尾的文件
```

```
!lib.a    # 但 lib.a 除外
/TODO     # 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO
build/    # 忽略 build/ 目录下的所有文件
doc/*.txt # 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
```

查看已暂存和未暂存的更新

实际上 `git status` 的显示比较简单，仅仅是列出了修改过的文件，如果要查看具体修改了什么地方，可以用 `git diff` 命令。稍后我们会详细介绍 `git diff`，不过现在，它已经能回答我们的两个问题了：当前做的哪些更新还没有暂存？有哪些更新已经暂存起来准备好了下次提交？`git diff` 会使用文件补丁的格式显示具体添加和删除的行。

假如再次修改 `README` 文件后暂存，然后编辑 `benchmarks.rb` 文件后先别暂存，运行 `status` 命令，会看到：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) ## new file: README ## Changed but not updated: # (use "git
add ..." to update what will be committed) ## modified: benchmarks.rb #
```

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 `git diff`：

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
```

```
log = git.commits('master', 15)
log.size
```

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

若要看已经暂存起来的文件和上次提交时的快照之间的差异，可以用 `git diff --cached` 命令。（Git 1.6.1 及更高版本还允许使用 `git diff --staged`，效果是相同的，但更好记些。）来看看实际的效果：

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

请注意，单单 `git diff` 不过是显示还没有暂存起来的改动，而不是这次工作和上次提交之间的差异。所以有时候你一下子暂存了所有更新过的文件后，运行 `git diff` 后却什么也没有，就是这个原因。

像之前说的，暂存 `benchmarks.rb` 后再编辑，运行 `git status` 会看到暂存前后的两个版本：

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changed but not updated:
#
```

```
# modified:  benchmarks.rb
#
```

现在运行 **git diff** 看暂存前后的变化:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
main()
```

```
##pp Grit::GitRuby.cache_client.stats
+# test line
```

然后用 **git diff --cached** 查看已经暂存起来的变化:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

提交更新

现在的暂存区域已经准备妥当可以提交了。在此之前,请一定要确认还有什么修改过的或新建的文件还没有 **git add** 过,否则提交的时候不会记录这些还没暂存起来的变化。所以,每次准备提交前,先用 **git status** 看下,是不是都已暂存起来了,然后再运行提交命令 **git commit**:


```
$ git commit
```

这种方式会启动文本编辑器以便输入本次提交的说明。（默认会启用 `shell` 的环境变量 `$EDITOR` 所指定的软件，一般都是 `vim` 或 `emacs`。当然也可以按照第一章介绍的方式，使用 `git config --global core.editor` 命令设定你喜欢的编辑软件。）

编辑器会显示类似下面的文本信息（本例选用 `Vim` 的屏显方式展示）：

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Changes to be committed:
#
#   (use "git reset HEAD
```

```
..." to unstage) # # new file: README # modified: benchmarks.rb ~ ~ ~
".git/COMMIT_EDITMSG" 10L, 283C
```

可以看到，默认的提交消息包含最后一次运行 `git status` 的输出，放在注释行里，另外开头还有一空行，供你输入提交说明。你完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。（如果觉得这还不够，可以用 `-v` 选项将修改差异的每一行都包含到注释中来。）退出编辑器时，`Git` 会丢掉注释行，将说明内容和本次更新提交到仓库。

另外也可以用 `-m` 参数后跟提交说明的方式，在一行命令中提交更新：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README
```

好，现在你已经创建了第一个提交！可以看到，提交后它会告诉你，当前是在哪个分支（`master`）提交的，本次提交的完整 `SHA-1` 校验和是什么（`463dc4f`），以及在本次提交中，有多少文件修订过，多少行添改和删改过。

记住，提交时记录的是放在暂存区域的快照，任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。`Git` 提供

了一个跳过使用暂存区域的方式，只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤：

```
$ git status

# On branch master

#

# Changed but not updated:

#

#   modified:   benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)
```

看到了吗？提交之前不再需要 `git add` 文件 `benchmarks.rb` 了。

移除文件

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。可以用 `git rm` 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，运行 `git status` 时就会在 “Changed but not updated” 部分（也就是_未暂存_清单）看到：

```
$ rm grit.gemspec

$ git status

# On branch master

#

# Changed but not updated:

#   (use "git add/rm ..." to update what will be committed) # # deleted:   grit.gemspec #
```

然后再运行 `git rm` 记录此次移除文件的操作：

```
$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master
```

```
#
# Changes to be committed:
#   (use "git reset HEAD
... " to unstage) # # deleted: grit.gemspec #
```

最后提交的时候，该文件就不再纳入版本管理了。如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 **-f**（译注：即 **force** 的首字母），以防误删除文件后丢失修改的内容。

另外一种情况是，我们想把文件从 **Git** 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，仅是从跟踪清单中删除。比如一些大型日志文件或者一堆[.a](#) 编译文件，不小心纳入仓库后，要移除跟踪但不删除文件，以便稍后在 **.gitignore** 文件中补上，用 **--cached** 选项即可：

```
$ git rm --cached readme.txt
```

后面可以列出文件或者目录的名字，也可以使用 **glob** 模式。比方说：

```
$ git rm log/\*.log
```

注意到星号 ***** 之前的反斜杠 ****，因为 **Git** 有它自己的文件模式扩展匹配方式，所以我们不用 **shell** 来帮忙展开（译注：实际上不加反斜杠也可以运行，只不过按照 **shell** 扩展的话，仅仅删除指定目录下的文件而不会递归匹配。上面的例子本来就指定了目录，所以效果等同，但下面的例子就会用递归方式匹配，所以必须加反斜杠。）。此命令删除所有 **log/** 目录下扩展名为 **.log** 的文件。类似的比如：

```
$ git rm \*~
```

会递归删除当前目录及其子目录中所有 **~** 结尾的文件。

移动文件

不像其他的 **VCS** 系统，**Git** 并不跟踪文件移动操作。如果在 **Git** 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。不过 **Git** 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 **Git** 的 **mv** 命令时一定会困惑不已。要在 **Git** 中对文件改名，可以这么做：

```
$ git mv file_from file_to
```

它会恰如预期般正常工作。实际上，即便此时查看状态信息，也会明白无误地看到关于重命

名操作的说明：

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # renamed: README.txt -> README #
```

其实，运行 **git mv** 就相当于运行了下面三条命令：

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

如此分开操作，**Git** 也会意识到这是一次改名，所以不管何种方式都一样。当然，直接用 **git mv** 轻便得多，不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

2.3 查看提交历史

在提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，可以使用 **git log** 命令查看。

接下来的例子会用我专门用于演示的 **simplegit** 项目，运行下面的命令获取该项目源代码：

```
git clone git://github.com/schacon/simplegit-progit.git
```

然后在此项目中运行 **git log**，应该会看到下面的输出：

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon
```

```
Date: Mon Mar 17 21:52:11 2008 -0700 changed the version number commit
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Author: Scott Chacon Date: Sat Mar 15
16:40:33 2008 -0700 removed unnecessary test code commit
a11bef06a3f659402fe7563abf99ad00de2209e6 Author: Scott Chacon Date: Sat Mar 15
10:31:28 2008 -0700 first commit
```

默认不用任何参数的话，**git log** 会按提交时间列出所有的更新，最近的更新排在最上面。看到了吗，每次更新都有一个 **SHA-1** 校验和、作者的名字和电子邮件地址、提交时间，最后缩进一个段落显示提交说明。

git log 有许多选项可以帮助你搜寻感兴趣的提交，接下来我们介绍些最常用的。

我们常用 **-p** 选项展开显示每次提交的内容差异，用 **-2** 则仅显示最近的两次更新：

```
$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon
```

```
Date: Mon Mar 17 21:52:11 2008 -0700 changed the version number diff --git
a/Rakefile b/Rakefile index a874b73..8f94139 100644 --- a/Rakefile +++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask' spec = Gem::Specification.new do
|s| - s.version = "0.1.0" + s.version = "0.1.1" s.author = "Scott Chacon" commit
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Author: Scott Chacon Date: Sat Mar 15
16:40:33 2008 -0700 removed unnecessary test code diff --git a/lib/simplegit.rb
b/lib/simplegit.rb index a0a60ae..47c6340 100644 --- a/lib/simplegit.rb +++
b/lib/simplegit.rb @@ -18,8 +18,3 @@ class SimpleGit end end - -if $0 == __FILE__
- git = SimpleGit.new - puts git.show -end \ No newline at end of file
```

在做代码审查，或者要快速浏览其他协作者提交的更新都作了哪些改动时，就可以用这个选项。此外，还有许多摘要选项可以用，比如 **--stat**，仅显示简要的增改行数统计：

```
$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon
```

```
Date: Mon Mar 17 21:52:11 2008 -0700 changed the version number Rakefile |
```

```

2 +- 1 files changed, 1 insertions(+), 1 deletions(-) commit
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Author: Scott Chacon Date: Sat Mar 15
16:40:33 2008 -0700 removed unnecessary test code lib/simplegit.rb | 5 ----- 1
files changed, 0 insertions(+), 5 deletions(-) commit
a11bef06a3f659402fe7563abf99ad00de2209e6 Author: Scott Chacon Date: Sat Mar 15
10:31:28 2008 -0700 first commit README | 6 ++++++ Rakefile | 23
+++++ lib/simplegit.rb | 25 ++++++ 3 files
changed, 54 insertions(+), 0 deletions(-)

```

每个提交都列出了修改过的文件，以及其中添加和移除的行数，并在最后列出所有增减行数小计。还有个常用的 `--pretty` 选项，可以指定使用完全不同于默认格式的方式展示提交历史。比如用 `oneline` 将每个提交放在一行显示，这在提交数很大时非常有用。另外还有 `short`, `full` 和 `fuller` 可以用，展示的信息或多或少有些不同，请自己动手实践一下看看效果如何。

```

$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

```

但最有意思的是 `format`，可以定制要显示的记录格式，这样的输出便于后期编程提取分析，像这样：

```

$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit

```

表 2-1 列出了常用的格式占位符写法及其代表的意义。

选项	说明
%H	提交对象（commit）的完整哈希字符串
%h	提交对象的简短哈希字符串
%T	树对象（tree）的完整哈希字符串
%t	树对象的简短哈希字符串
%P	父对象（parent）的完整哈希字符串
%p	父对象的简短哈希字符串
%an	作者（author）的名字
%ae	作者的电子邮件地址
%ad	作者修订日期（可以用 <code>-date=</code> 选项定制格式）

%ar 作者修订日期，按多久以前的方式显示
%cn 提交者 (committer) 的名字
%ce 提交者的电子邮件地址
%cd 提交日期
%cr 提交日期，按多久以前的方式显示
%s 提交说明

你一定奇怪_作者 (author)_和_提交者 (committer)_之间究竟有何差别，其实作者指的是实际作出修改的人，提交者指的是最后将此 工作成果提交到仓库的人。所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者。我们会在第五章 再详细介绍两者之间的细微差别。

用 **oneline** 或 **format** 时结合 **--graph** 选项，可以看到开头多出一些 **ASCII** 字符串表示的简单图形，形象地展示了每个提交所在的分支及其分化衍合情况。在我们之前提到的 **Grit** 项目仓库中可以看到：

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

以上只是简单介绍了一些 **git log** 命令支持的选项。表 2-2 还列出了一些其他常用的选项及其释义。

选项 说明

-p 按补丁格式显示每个更新之间的差异。
--stat 显示每次更新的文件修改统计信息。
--shortstat 只显示 --stat 中最后的行数修改添加移除统计。
--name-only 仅在提交信息后显示已修改的文件清单。
--name-status 显示新增、修改、删除的文件清单。
--abbrev-commit 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
--relative-date 使用较短的相对时间显示（比如，“2 weeks ago”）。
--graph 显示 ASCII 图形表示的分支合并历史。

`--pretty` 使用其他格式显示历史提交信息。可用的选项包括 `oneline`, `short`, `full`, `fuller` 和 `format` (后跟指定格式)。

限制输出长度

除了定制输出格式的选项之外, `git log` 还有许多非常实用的限制输出长度的选项, 也就是只输出部分提交信息。之前我们已经看到过 `-2` 了, 它只显示最近的两条提交, 实际上, 这是 `-n` 选项的写法, 其中的 `n` 可以是任何自然数, 表示仅显示最近的若干条提交。不过实践中我们是不太用这个选项的, `Git` 在输出所有提交时会自动调用分页程序 (`less`), 要看更早的更新只需翻到下页即可。

另外还有按照时间作限制的选项, 比如 `--since` 和 `--until`。下面的命令列出所有最近两周内的提交:

```
$ git log --since=2.weeks
```

你可以给出各种时间格式, 比如说具体的某一天 (“`2008-01-15`”) 或者是多久以前 (“`2 years 1 day 3 minutes ago`”)。

还可以给出若干搜索条件, 列出符合的提交。用 `--author` 选项显示指定作者的提交, 用 `--grep` 选项搜索提交说明中的关键字。(请注意, 如果要得到同时满足这两个选项搜索条件的提交, 就必须用 `--all-match` 选项。)

如果只关心某些文件或者目录的历史提交, 可以在 `git log` 选项的最后指定它们的路径。因为是放在最后位置上的选项, 所以用两个短划线 (`--`) 隔开之前的选项和后面限定的路径名。

表 2-3 还列出了其他常用的类似选项。

选项 说明

`-(n)` 仅显示最近的 `n` 条提交

`--since`, `--after` 仅显示指定时间之后的提交。

`--until`, `--before` 仅显示指定时间之前的提交。

`--author` 仅显示指定作者相关的提交。

`--committer` 仅显示指定提交者相关的提交。

来看一个实际的例子, 如果要查看 `Git` 仓库中, 2008 年 10 月期间, Junio Hamano 提交的但未合并的测试脚本 (位于项目的 `t/` 目录下的文件), 可以用下面的查询命令:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
    --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
```

```
51a94af - Fix "checkout --track -b newbranch" on detach  
b0ad11e - pull: allow "git pull origin $something:$cur
```

Git 项目有 20,000 多条提交，但我们给出搜索选项后，仅列出了其中满足条件的 6 条。

使用图形化工具查阅提交历史

有时候图形化工具更容易展示历史提交的变化，随 Git 一同发布的 `gitk` 就是这样一种工具。它是用 Tcl/Tk 写成的，基本上相当于 `git log` 命令的可视化版本，凡是 `git log` 可以用的选项也都能用在 `gitk` 上。在项目工作目录中输入 `gitk` 命令后，就会启动图 2-2 所示的界面。

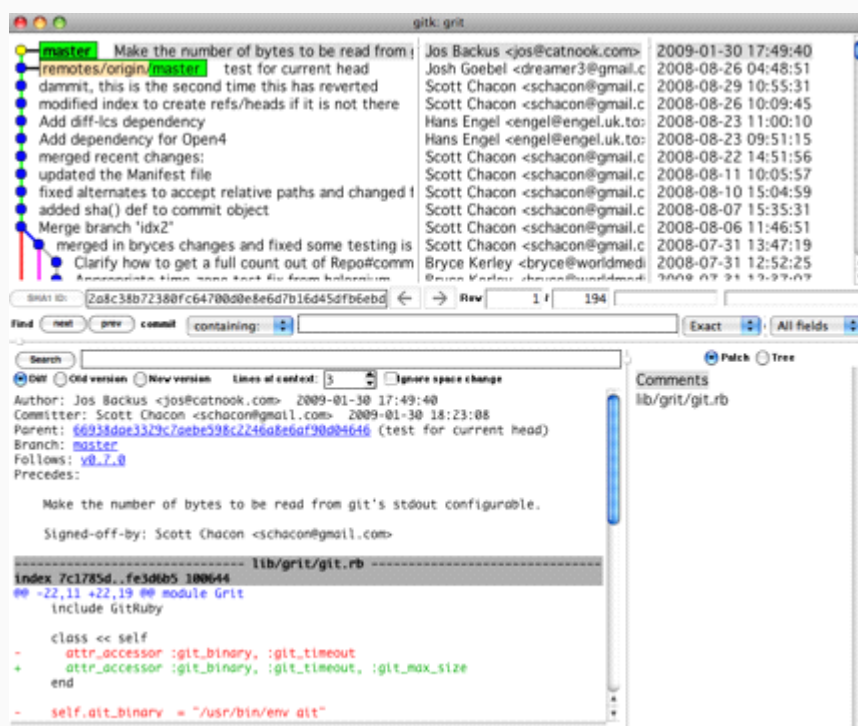


图 2-2. `gitk` 的图形界面上半个窗口显示的是历次提交的分支祖先图谱，下半个窗口显示当前点选的提交对应的具体差异。

2.4 撤消操作

任何时候，你都有可能需要撤消刚才所做的某些操作。接下来，我们会介绍一些基本的撤消操作相关的命令。请注意，有些操作并不总是可以撤消的，所以请务必谨慎小心，一旦失误，就有可能丢失部分工作成果。

修改最后一次提交

有时候我们提交完了才发现漏掉了几个文件没有加，或者提交信息写错了。想要撤消刚才的

提交操作，可以使用 `--amend` 选项重新提交：

```
$ git commit --amend
```

此命令将使用当前的暂存区域快照提交。如果刚才提交完没有作任何改动，直接运行此命令的话，相当于有机会重新编辑提交说明，但将要提交的文件快照和之前的一样。

启动文本编辑器后，会看到上次提交时的说明，编辑它确认没问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交。

如果刚才提交时忘了暂存某些修改，可以先补上暂存操作，然后再运行 `--amend` 提交：

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

上面的三条命令最终只是产生一个提交，第二个提交命令修正了第一个的提交内容。

取消已经暂存的文件

接下来的两个小节将演示如何取消暂存区域中的文件，以及如何取消工作目录中已修改的文件。不用担心，查看文件状态的时候就提示了该如何撤消，所以不需要死记硬背。来看下面的例子，有两个修改过的文件，我们想要分开提交，但不小心用 `git add .` 全加到了暂存区域。该如何撤消暂存其中的一个文件呢？其实，`git status` 的命令输出已经告诉了我们该怎么做：

```
$ git add .
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD ..." to unstage) # # modified: README.txt # modified: benchmarks.rb #
```

就在“Changes to be committed”下面，括号中有提示，可以使用 `git reset HEAD ...` 的方式取消暂存。好吧，我们来试试取消暂存 `benchmarks.rb` 文件：

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # modified: README.txt # # Changed but not updated: # (use
"git add ..." to update what will be committed) # (use "git checkout -- ..." to
discard changes in working directory) # # modified: benchmarks.rb #
```

这条命令看起来有些古怪，先别管，能用就行。现在 **benchmarks.rb** 文件又回到了之前已修改未暂存的状态。

取消对文件的修改

如果觉得刚才对 **benchmarks.rb** 的修改完全没有必要，该如何取消修改，回到之前的状态（也就是修改之前的版本）呢？**git status** 同样提示了具体的撤消方法，接着上面的例子，现在未暂存区域看起来像这样：

```
# Changed but not updated:
#   (use "git add
```

```
..." to update what will be committed) # (use "git checkout -- ..." to discard
changes in working directory) # # modified: benchmarks.rb #
```

在第二个括号中，我们看到了抛弃文件修改的命令（至少在 **Git 1.6.1** 以及更高版本中会这样提示，如果你还在用老版本，我们强烈建议你升级，以获取最佳的用户体验），让我们试试看：

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # modified: README.txt #
```

可以看到，该文件已经恢复到修改前的版本。你可能已经意识到了，这条命令有些危险，所

有对文件的修改都没有了，因为我们刚刚把之前版本的文件复制过来重写了此文件。所以在用这条命令前，请务必确定真的不再需要保留刚才的修改。如果只是想回退版本，同时保留刚才的修改以便将来继续工作，可以用下章介绍的 **stashing** 和分支来处理，应该会更好些。

记住，任何已经提交到 **Git** 的都可以被恢复。即便在已经删除的分支中的提交，或者用 **--amend** 重新改写的提交，都可以被恢复（关于数据恢复的内容见第九章）。所以，你可能失去的数据，仅限于没有提交过的，对 **Git** 来说它们就像从未存在过一样。

2.5 远程仓库的使用

要参与任何一个 **Git** 项目的协作，必须要了解该如何管理远程仓库。远程仓库是指托管在网络上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写。同他人协作开发某个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。管理远程仓库的工作，包括添加远程库，移除废弃的远程库，管理各式远程库分支，定义是否跟踪这些分支，等等。本节我们将详细讨论远程库的管理和使用。

查看当前的远程库

要查看当前配置有哪些远程仓库，可以用 **git remote** 命令，它会列出每个远程库的简短名字。在克隆完某个项目后，至少可以看到一个名为 **origin** 的远程库，**Git** 默认使用这个名字来标识你所克隆的原始仓库：

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

也可以加上 **-v** 选项（译注：此为 **--verbose** 的简写，取首字母），显示对应的克隆地址：

```
$ git remote -v
origin git://github.com/schacon/ticgit.git
```

如果有多个远程仓库，此命令将全部列出。比如在我的 **Grit** 项目中，可以看到：

```
$ cd grit
$ git remote -v
bakkdoor git://github.com/bakkdoor/grit.git
cho45 git://github.com/cho45/grit.git
defunkt git://github.com/defunkt/grit.git
koke git://github.com/koke/grit.git
origin git@github.com:mojombo/grit.git
```

这样一来，我就可以非常轻松地从这个用户的仓库中，拉取他们的提交到本地。请注意，上面列出的地址只有 **origin** 用的是 **SSH URL** 链接，所以也只有这个仓库我能推送数据上去（我们会在第四章解释原因）。

添加远程仓库

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用，运行 **git remote add [shortname] [url]**:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

现在可以用字符串 **pb** 指代对应的仓库地址了。比如说，要抓取所有 **Paul** 有的，但本地仓库没有的信息，可以运行 **git fetch pb**:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master    -> pb/master
* [new branch]      ticgit    -> pb/ticgit
```

现在，**Paul** 的主干分支（**master**）已经完全可以在本地访问了，对应的名字是 **pb/master**，你可以将它合并到自己的某个分支，或者切换到这个分支，看看有什么有趣的更新。

从远程仓库抓取数据

正如之前所看到的，可以用下面的命令从远程仓库抓取数据到本地：

```
$ git fetch [remote-name]
```

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。（我们会在第三章详细讨论关于分支的概念和操作。）

如果是克隆了一个仓库，此命令会自动将远程仓库归于 **origin** 名下。所以，**git fetch origin** 会抓取从你上次克隆以来别人上传到此远程仓库中的所有更新（或是上次 **fetch** 以来别人提交的更新）。有一点很重要，需要记住，**fetch** 命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支，只有当你确实准备好了，才能手工合并。

如果设置了某个分支用于跟踪某个远端仓库的分支（参见下节及第三章的内容），可以使用 **git pull** 命令自动抓取数据下来，然后将远端分支自动合并到本地仓库中当前分支。在日常工作中我们经常这么用，既快且好。实际上，默认情况下 **git clone** 命令本质上就是自动创建了本地的 **master** 分支用于跟踪远程仓库中的 **master** 分支（假设远程仓库确实有 **master** 分支）。所以一般我们运行 **git pull**，目的都是要从原始克隆的远端仓库中抓取数据后，合并到工作目录中的当前分支。

推送数据到远程仓库

项目进行到一个阶段，要同别人分享目前的成果，可以将本地仓库中的数据推送到远程仓库。实现这个任务的命令很简单：**git push [remote-name] [branch-name]**。如果要把本地的 **master** 分支推送到 **origin** 服务器上（再次说明下，克隆操作会自动使用默认的 **master** 和 **origin** 名字），可以运行下面的命令：

```
$ git push origin master
```

只有在所克隆的服务器上有写权限，或者同一时刻没有其他人在推数据，这条命令才会如期完成任务。如果你推数据前，已经有其他人推送了若干更新，那你的推送操作就会被驳回。你必须先把他们的更新抓取到本地，合并到自己的项目中，然后才可以再次推送。有关推送数据到远程仓库的详细内容见第三章。

查看远程仓库信息

我们可以通过命令 **git remote show [remote-name]** 查看某个远程仓库的详细信息，比如要看所克隆的 **origin** 仓库，可以运行：

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
```



```
master
ticgit
```

除了对应的克隆地址外，它还给出了许多额外的信息。它友善地告诉你如果是在 **master** 分支，就可以用 **git pull** 命令抓取数据合并到本地。另外还列出了所有处于跟踪状态中的远端分支。

上面的例子非常简单，而随着使用 **Git** 的深入，**git remote show** 给出的信息可能会像这样：

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

它告诉我们，运行 **git push** 时缺省推送的分支是什么（译注：最后两行）。它还显示了有哪些远端分支还没有同步到本地（译注：第六行的 **caching** 分支），哪些已同步到本地的远端分支在远端服务器上已被删除（译注：**Stale tracking branches** 下面的两个分支），以及运行 **git pull** 时将自动合并哪些分支（译注：前四行中列出的 **issues** 和 **master** 分支）。

远程仓库的删除和重命名

在新版 **Git** 中可以用 **git remote rename** 命令修改某个远程仓库在本地的简短名称，比如想把 **pb** 改成 **paul**，可以这么运行：

```
$ git remote rename pb paul
$ git remote
origin
paul
```

注意，对远程仓库的重命名，也会使对应的分支名称发生变化，原来的 `pb/master` 分支现在成了 `paul/master`。

碰到远端仓库服务器迁移，或者原来的克隆镜像不再使用，又或者某个参与者不再贡献代码，那么需要移除对应的远端仓库，可以运行 `git remote rm` 命令：

```
$ git remote rm paul
$ git remote
origin
```

2.6 打标签

同大多数 VCS 一样，Git 也可以对某一时间点上的版本打上标签。人们在发布某个软件版本（比如 `v1.0` 等等）的时候，经常这么做。本节我们一起来学习如何列出所有可用的标签，如何新建标签，以及各种不同类型标签之间的差别。

列显已有的标签

列出现有标签的命令非常简单，直接运行 `git tag` 即可：

```
$ git tag
v0.1
v1.3
```

显示的标签按字母顺序排列，所以标签的先后并不表示重要程度的轻重。

我们可以用特定的搜索模式列出符合条件的标签。在 Git 自身项目仓库中，有着超过 240 个标签，如果你只对 1.4.2 系列的版本感兴趣，可以运行下面的命令：

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

新建标签

Git 使用的标签有两种类型：轻量级的（**lightweight**）和含附注的（**annotated**）。轻量级标签就像是不会变化的分支，实际上它就是个指向特定提交对象的引用。而含附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 **GNU Privacy Guard (GPG)** 来签署或验证。一般我们都建议使用含附注型的标签，以便保留相关信息；当然，如果只是临时性加注标签，或者不需要旁注额外信息，用轻量级标签也没问题。

含附注的标签

创建一个含附注类型的标签非常简单，用 **-a**（译注：取 **annotated** 的首字母）指定标签名字即可：

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

而 **-m** 选项则指定了对应的标签说明，Git 会将此说明一同保存在标签对象中。如果没有给出该选项，Git 会启动文本编辑软件供你输入标签说明。

可以使用 **git show** 命令查看相应标签的版本信息，并连同显示打标签时的提交对象。

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon
```

```

Date: Mon Feb 9 14:45:11 2009 -0800 my version 1.4 commit
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge: 4a447f7... a6b4c97... Author:
Scott Chacon Date: Sun Feb 8 19:02:46 2009 -0800 Merge branch 'experiment'
```

我们可以看到在提交对象信息上面，列出了此标签的提交者和提交时间，以及相应的标签说明。

签署标签

如果你有自己的私钥，还可以用 **GPG** 来签署标签，只需要把之前的 **-a** 改为 **-s**（译注：取 **signed** 的首字母）即可：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Scott Chacon
```

```
" 1024-bit DSA key, ID F721C45A, created 2009-02-09
```

现在再运行 **git show** 会看到对应的 **GPG** 签名也附在其内：

```
$ git show v1.5

tag v1.5

Tagger: Scott Chacon
```

```

Date: Mon Feb 9 15:22:20 2009 -0800 my signed 1.5 tag -----BEGIN PGP
SIGNATURE----- Version: GnuPG v1.4.8 (Darwin)
iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAj82/ =WryJ -----END PGP SIGNATURE----- commit
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge: 4a447f7... a6b4c97... Author:
Scott Chacon Date: Sun Feb 8 19:02:46 2009 -0800 Merge branch 'experiment'
```

稍后我们再学习如何验证已经签署的标签。

轻量级标签

轻量级标签实际上就是一个保存着对应提交对象的校验和信息的文件。要创建这样的标签，一个 **-a**、**-s** 或 **-m** 选项都不用，直接给出标签名字即可：

```
$ git tag v1.4-lw

$ git tag

v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

现在运行 **git show** 查看此标签信息，就只有相应的提交对象摘要：

```
$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
```

```
Merge: 4a447f7... a6b4c97...
```

```
Author: Scott Chacon
```

```
Date: Sun Feb 8 19:02:46 2009 -0800 Merge branch 'experiment'
```

验证标签

可以使用 `git tag -v [tag-name]`（译注：取 **verify** 的首字母）的方式验证已经签署的标签。此命令会调用 **GPG** 来验证签名，所以你需要有签署者的公钥，存放在 **keyring** 中，才能验证：

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano
```

```
1158138501 -0700 GIT 1.4.2.1 Minor fixes since 1.4.2, including git-mv and
git-http with alternates. gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using
DSA key ID F3119B9A gpg: Good signature from "Junio C Hamano " gpg: aka "[jpeg
image of size 1513]" Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6
D9A4 F311 9B9A
```

若是没有签署者的公钥，会报告类似下面这样的错误：

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

后期加注标签

你甚至可以在后期对早先的某次提交加注标签。比如在下面展示的提交历史中：

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
```

```
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

我们忘了在提交“**updated rakefile**”后为此项目打上版本号 **v1.2**，没关系，现在也能做。只要在打标签的时候跟上对应提交对象的校验和（或前几位字符）即可：

```
$ git tag -a v1.2 9fceb02
```

可以看到我们已经补上了标签：

```
$ git tag
```

```
v0.1
```

```
v1.2
```

```
v1.3
```

```
v1.4
```

```
v1.4-lw
```

```
v1.5
```

```
$ git show v1.2
```

```
tag v1.2
```

```
Tagger: Scott Chacon
```

```
Date: Mon Feb 9 15:32:16 2009 -0800 version 1.2 commit
9fceb02d0ae598e95dc970b74767f19372d61af8 Author: Magnus Chacon Date: Sun Apr 27
20:43:35 2008 -0700 updated rakefile ...
```

分享标签

默认情况下，**git push** 并不会把标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。其命令格式如同推送分支，运行 **git push origin [tagname]** 即可：

```
$ git push origin v1.5
```

```
Counting objects: 50, done.
```

```
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
```

```
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v1.5 -> v1.5
```

如果要一次推送所有本地新增的标签上去，可以使用 `--tags` 选项：

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v0.1 -> v0.1
* [new tag]          v1.2 -> v1.2
* [new tag]          v1.4 -> v1.4
* [new tag]          v1.4-lw -> v1.4-lw
* [new tag]          v1.5 -> v1.5
```

现在，其他人克隆共享仓库或拉取数据同步后，也会看到这些标签。

2.7 技巧和窍门

在结束本章之前，我还想和大家分享一些 **Git** 使用的技巧和窍门。很多使用 **Git** 的开发者可能根本就没用过这些技巧，我们也不是说在读过本书后非得用这些技巧不可，但至少应该有所了解吧。说实话，有了这些小窍门，我们的工作可以变得更简单，更轻松，更高效。

自动完成

如果你用的是 **Bash shell**，可以试试看 **Git** 提供的自动完成脚本。下载 **Git** 的源代码，进入 `contrib/completion` 目录，会看到一个 `git-completion.bash` 文件。将此文件复制到你自己的用户主目录中（译注：按照下面的示例，还应改名加上点：`cp git-completion.bash ~/.git-completion.bash`），并把下面一行内容添加到你的 `.bashrc` 文件中：

```
source ~/.git-completion.bash
```

也可以为系统上所有用户都设置默认使用此脚本。**Mac** 上将此脚本复制到 `/opt/local/etc/bash_completion.d` 目录中，**Linux** 上则复制到 `/etc/bash_completion.d/` 目录中。这两处目录中的脚本，都会在 **Bash** 启动时自动加载。

如果在 **Windows** 上安装了 **msysGit**，默认使用的 **Git Bash** 就已经配好了这个自动完成脚本，可以直接使用。

在输入 **Git** 命令的时候可以敲两次跳格键 (**Tab**)，就会看到列出所有匹配的可用命令建议：

```
$ git co
```

```
commit config
```

此例中，键入 **git co** 然后连按两次 **Tab** 键，会看到两个相关的建议（命令） **commit** 和 **config**。继而输入 **m** 会自动完成 **git commit** 命令的输入。

命令的选项也可以用这种方式自动完成，其实这种情况更实用些。比如运行 **git log** 的时候忘了相关选项的名字，可以输入开头的几个字母，然后敲 **Tab** 键看看有哪些匹配的：

```
$ git log --s
```

```
--shortstat --since= --src-prefix= --stat --summary
```

这个技巧不错吧，可以节省很多输入和查阅文档的时间。

Git 命令别名

Git 并不会推断你输入的几个字符将会是哪条命令，不过如果想偷懒，少敲几个命令的字符，可以用 **git config** 为命令设置别名。来看看下面的例子：

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

现在，如果要输入 **git commit** 只需键入 **git ci** 即可。而随着 **Git** 使用的深入，会有很多经常要用到的命令，遇到这种情况，不妨建个别名提高效率。

使用这种技术还可以创造出新的命令，比方说取消暂存文件时的输入比较繁琐，可以自己设置一下：

```
$ git config --global alias.unstage 'reset HEAD --'
```

这样一来，下面的两条命令完全等同：

```
$ git unstage fileA
$ git reset HEAD fileA
```


显然，使用别名的方式看起来更清楚。另外，我们还经常设置 `last` 命令：

```
$ git config --global alias.last 'log -1 HEAD'
```

然后要看最后一次的提交信息，就变得简单多了：

```
$ git last  
commit 66938dae3329c7aebe598c2246a8e6af90d04646  
Author: Josh Goebel
```

```
Date: Tue Aug 26 19:48:51 2008 +0800 test for current head Signed-off-by: Scott  
Chacon
```

可以看出，实际上 **Git** 只是简单地在命令中替换了你设置的别名。不过有时候我们希望运行某个外部命令，而非 **Git** 的附属工具，这个好办，只需要在命令前加上 `!` 就行。如果你自己写了些处理 **Git** 仓库信息的脚本的话，就可以用这种技术包装起来。作为演示，我们可以设置用 `git visual` 启动 `gitk`：

```
$ git config --global alias.visual '!gitk'
```

2.8 小结

到目前为止，你已经学会了最基本的 **Git** 操作：创建和克隆仓库，做出更新，暂存并提交这些更新，以及查看所有历史更新记录。接下来，我们将学习 **Git** 的必杀技特性：分支模型。

Git 详解之三 Git 分支

您的评价:

很差

[收藏该经验](#)

Git 分支

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线工作的同时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

有人把 Git 的分支模型称为“必杀技特性”，而正是因为它，将 Git 从版本控制系统家族里区分出来。Git 有何特别之处呢？Git 的分支可谓是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。

3.1 何谓分支

为了理解 Git 分支的实现方式，我们需要回顾一下 Git 是如何储存数据的。或许你还记得第一章的内容，Git 保存的不是文件差异或者变化量，而只是一系列文件快照。

在 Git 中提交时，会保存一个提交（commit）对象，该对象包含一个指向暂存内容快照的指针，包含本次提交的作者等相关附属信息，包含零个或多个指向该提交对象的父对象指针：首次提交是没有直接祖先的，普通提交有一个祖先，由两个或多个分支合并产生的提交则有多个祖先。

为直观起见，我们假设在工作目录中有三个文件，准备将它们暂存后提交。暂存操作会对每一个文件计算校验和（即第一章中提到的 SHA-1 哈希字符串），然后把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 类型的对象存储这些快照），并将校验和加入暂存区域：

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

当使用 `git commit` 新建一个提交对象前，Git 会先计算每一个子目录（本例中就是项目根目录）的校验和，然后在 Git 仓库中将这目录保存为树（tree）对象。之后 Git 创建的提交对象，除了包含相关提交信息以外，还包含着指向这个树对象（项目根目录）的指针，如此它就可以在将来需要的时候，重现此次快照的内容了。

现在，Git 仓库中有五个对象：三个表示文件快照内容的 blob 对象；一个记录着目录树内容及其中各个文件对应 blob 对象索引的 tree 对象；以及一个包含指向 tree 对象（根目录）的索引和其他提交信息元数据的 commit 对象。概念上来说，仓库中的各个对象保存

的数据和相互关系看起来如图 3-1 所示：

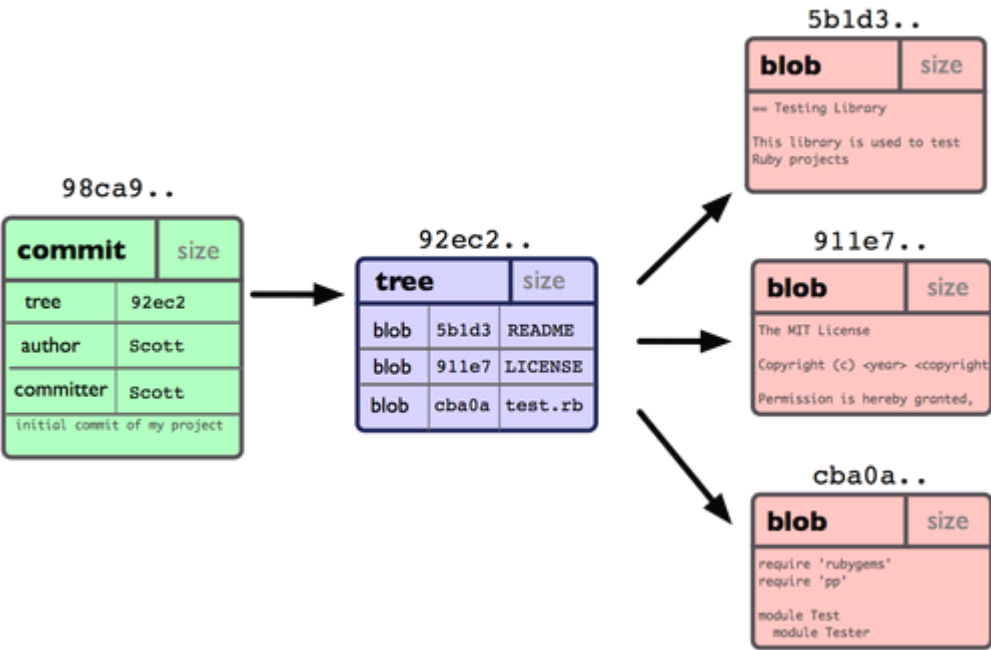


图 3-1. 单个提交对象在仓库中的数据结构

作些修改后再次提交，那么这次的提交对象会包含一个指向上次提交对象的指针（译注：即下图中的 **parent** 对象）。两次提交后，仓库历史会变成图 3-2 的样子：

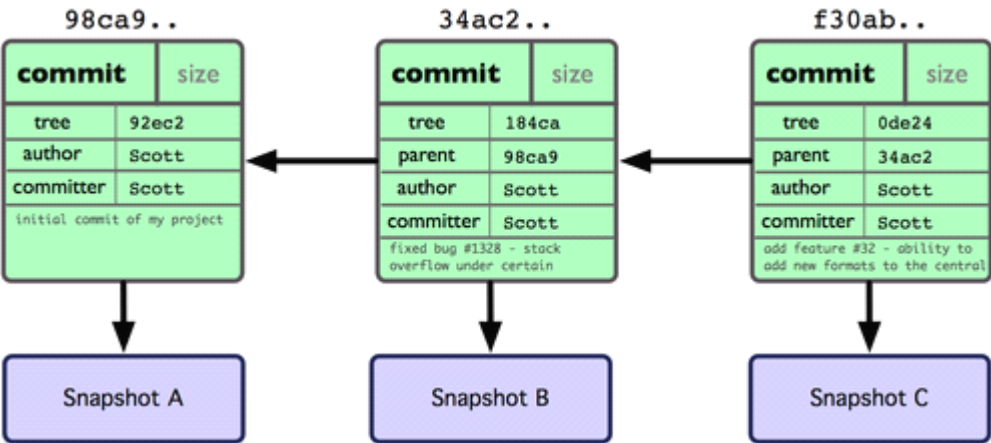


图 3-2. 多个提交对象之间的链接关系

现在来谈分支。Git 中的分支，其实本质上仅仅是个指向 **commit** 对象的可变指针。Git 会使用 **master** 作为分支的默认名字。在若干次提交后，你其实已经有了一个指向最后一次提交对象的 **master** 分支，它在每次提交的时候都会自动向前移动。

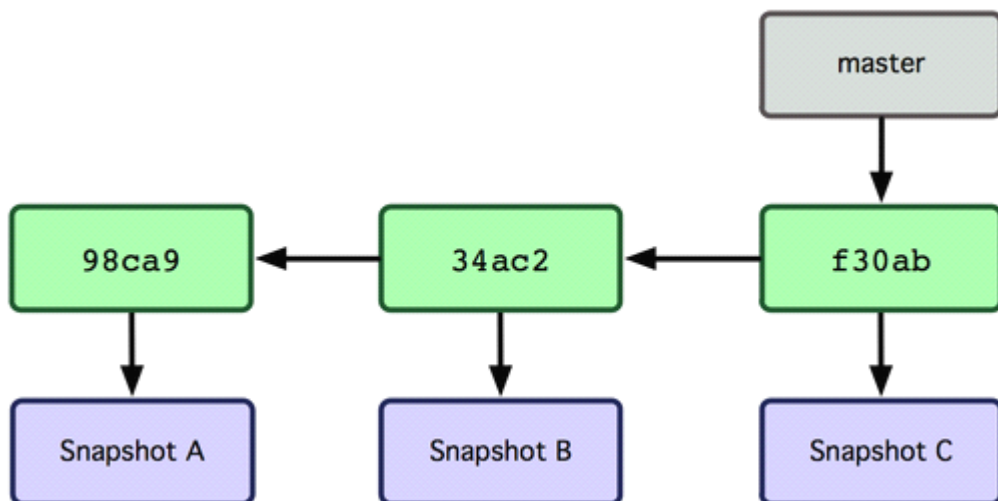


图 3-3. 分支其实就是从某个提交对象往回看的历史

那么，Git 又是如何创建一个新的分支的呢？答案很简单，创建一个新的分支指针。比如新建一个 `testing` 分支，可以使用 `git branch` 命令：

```
$ git branch testing
```

这会在当前 `commit` 对象上新建一个分支指针（见图 3-4）。

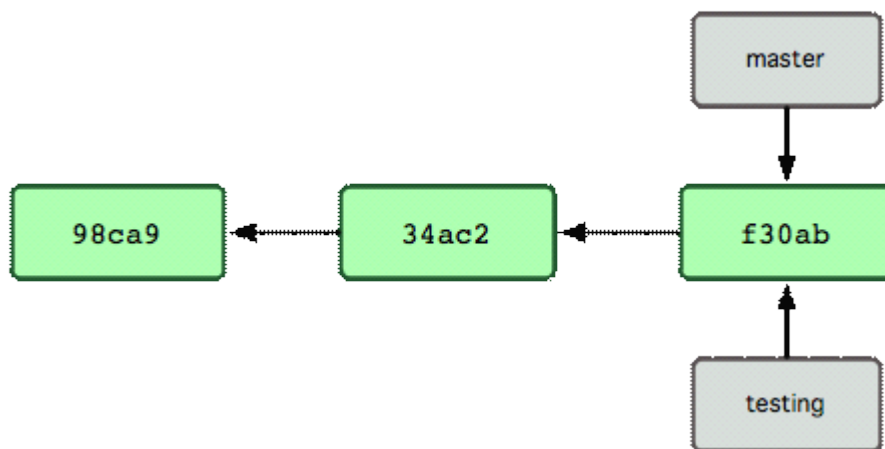


图 3-4. 多个分支指向提交数据的历史

那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 `HEAD` 的特别指针。请注意它和你熟知的许多其他版本控制系统（比如 `Subversion` 或 `CVS`）里的 `HEAD` 概念大不相同。在 Git 中，它是一个指向你正在工作中的本地分支的指针（译注：将 `HEAD` 想象为当前分支的别名。）。运行 `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 `master` 分支里工作（参考图 3-5）。

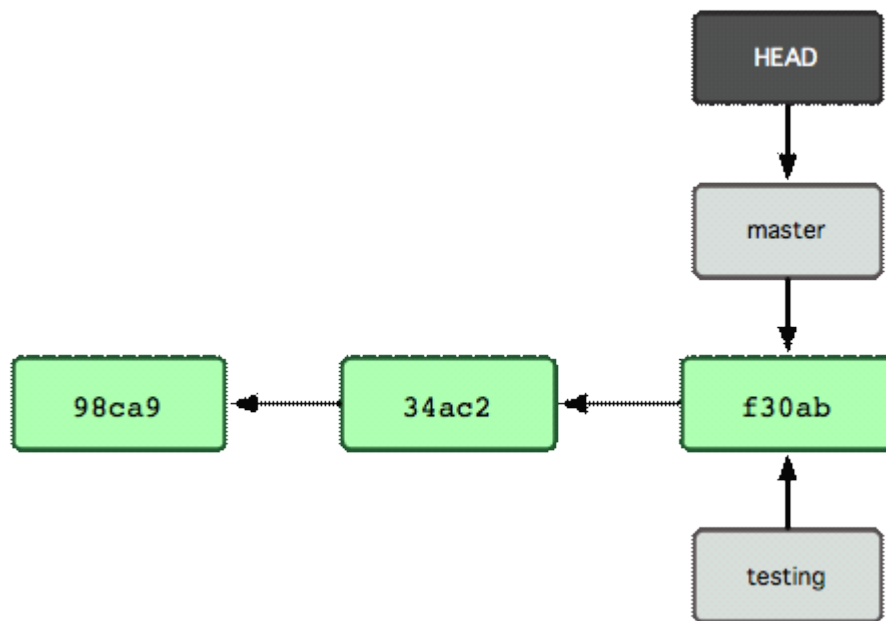


图 3-5. HEAD 指向当前所在的分支

要切换到其他分支，可以执行 `git checkout` 命令。我们现在转换到新建的 `testing` 分支：

```
$ git checkout testing
```

这样 `HEAD` 就指向了 `testing` 分支（见图3-6）。

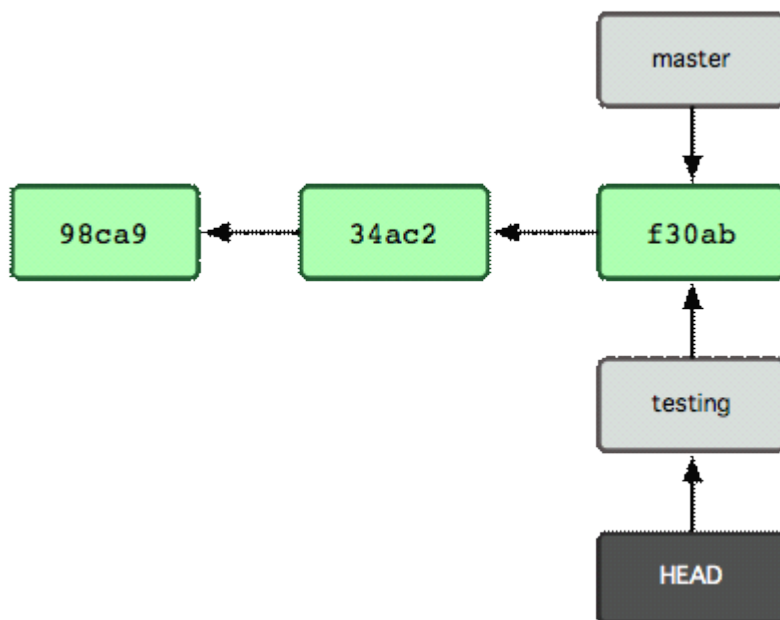


图 3-6. HEAD 在你转换分支时指向新的分支

这样的实现方式会给我们带来什么好处呢？好吧，现在不妨再提交一次：

```
$ vim test.rb
```

```
$ git commit -a -m 'made a change'
```

图 3-7 展示了提交后的结果。

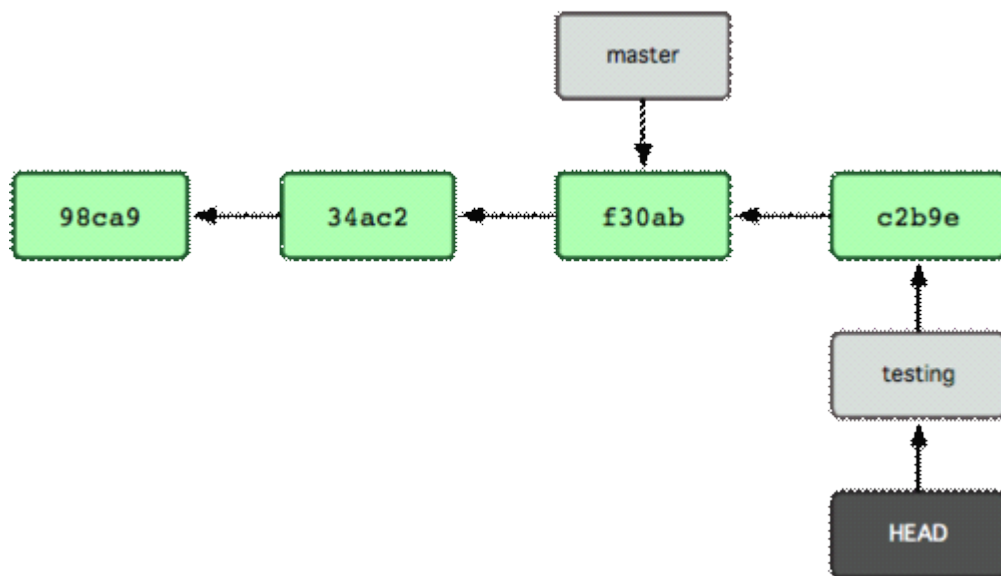


图 3-7. 每次提交后 HEAD 随着分支一起向前移动

非常有趣，现在 **testing** 分支向前移动了一格，而 **master** 分支仍然指向原先 `git checkout` 时所在的 **commit** 对象。现在我们回到 **master** 分支看看：

```
$ git checkout master
```

图 3-8 显示了结果。

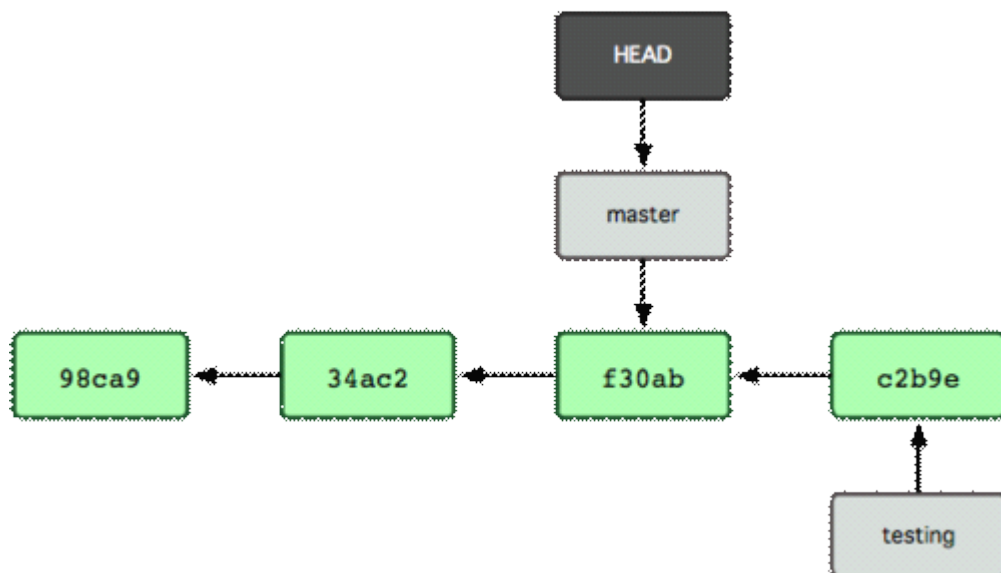


图 3-8. HEAD 在一次 checkout 之后移动到了另一个分支

这条命令做了两件事。它把 **HEAD** 指针移回到 **master** 分支，并把工作目录中的文件换成

了 **master** 分支所指向的快照内容。也就是说，现在开始所做的改动，将始于本项目中一个较老的版本。它的主要作用是将 **testing** 分支里作出的修改暂时取消，这样你就可以向另一个方向进行开发。

我们作些修改后再次提交：

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

现在我们的项目提交历史产生了分叉（如图 3-9 所示），因为刚才我们创建了一个分支，转换到其中进行了一些工作，然后又回到原来的主分支进行了另外一些工作。这些改变分别孤立在不同的分支里：我们可以在不同分支里反复切换，并在时机成熟时把它们合并到一起。而所有这些工作，仅仅需要 **branch** 和 **checkout** 这两条命令就可以完成。

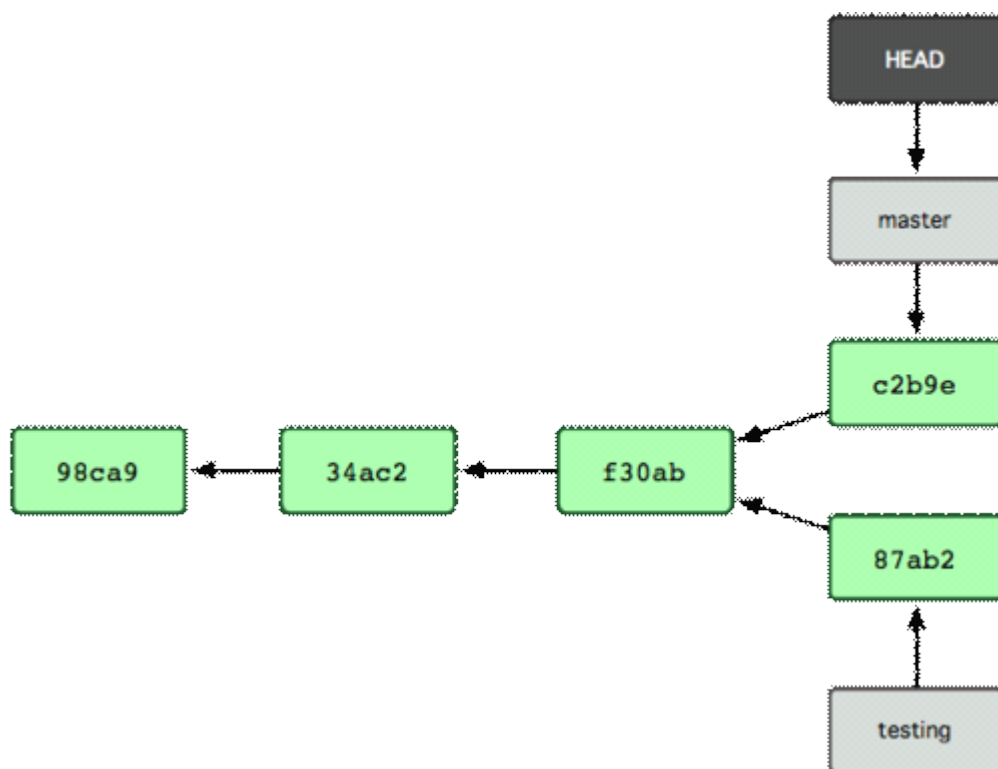


图 3-9. 不同流向的分支历史

由于 **Git** 中的分支实际上仅是一个包含所指对象校验和（40 个字符长度 **SHA-1** 字符串）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。

这和大多数版本控制系统形成了鲜明对比，它们管理分支大多采取备份所有项目文件到特定目录的方式，所以根据项目文件数量和大小不同，可能花费的时间也会有相当大的差别，快则几秒，慢则数分钟。而 **Git** 的实现与项目复杂度无关，它永远可以在几毫秒的时间内

完成分支的创建和切换。同时，因为每次提交时都记录了祖先信息（译注：即 **parent** 对象），将来要合并分支时，寻找恰当的合并基础（译注：即共同祖先）的工作其实已经自然而然地摆在那里了，所以实现起来非常容易。**Git** 鼓励开发者频繁使用分支，正是因为有着这些特性作保障。

接下来看看，我们为什么应该频繁使用分支。

3.2 分支的新建与合并

现在让我们来看一个简单的分支与合并的例子，实际工作中大体也会用到这样的工作流程：

1. 开发某个网站。 2. 为实现某个新的需求，创建一个分支。 3. 在这个分支上开展工作。

假设此时，你突然接到一个电话说有个很严重的问题需要紧急修补，那么可以按照下面的方式处理：

1. 返回到原先已经发布到生产服务器上的分支。 2. 为这次紧急修补建立一个新分支，并在其中修复问题。 3. 通过测试后，回到生产服务器所在的分支，将修补分支合并进来，然后再推送到生产服务器上。 4. 切换到之前实现新需求的分支，继续工作。

分支的新建与切换

首先，我们假设你正在项目中愉快地工作，并且已经提交了几次更新（见图 3-10）。

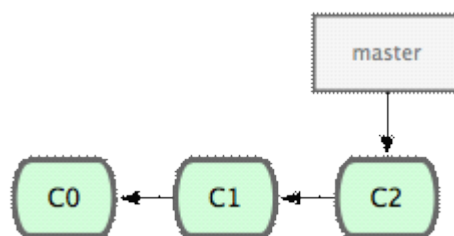


图 3-10. 一个简短的提交历史

现在，你决定要修补问题追踪系统上的 **#53** 问题。顺带说明下，**Git** 并不同任何特定的问题追踪系统打交道。这里为了说明要解决的问题，才把新建的分支取名为 **iss53**。要新建并切换到该分支，运行 **git checkout** 并加上 **-b** 参数：

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

这相当于执行下面这两条命令：

```
$ git branch iss53
$ git checkout iss53
```


图 3-11 示意该命令的执行结果。

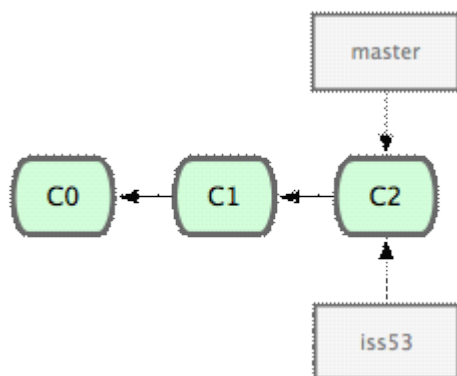


图 3-11. 创建了一个新分支的指针

接着你开始尝试修复问题，在提交了若干次更新后，`iss53` 分支的指针也会随着向前推进，因为它就是当前分支（换句话说，当前的 `HEAD` 指针正指向 `iss53`，见图 3-12）：

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

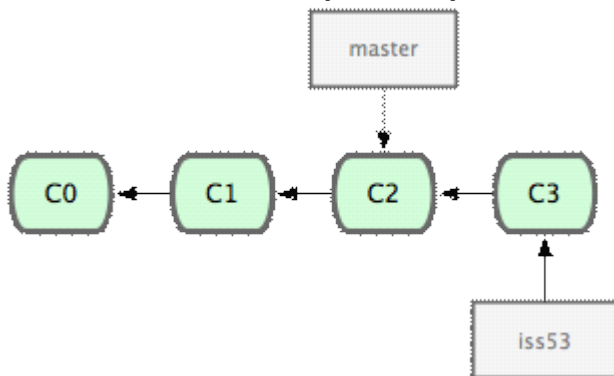


图 3-12. `iss53` 分支随工作进展向前推进

现在你就接到了那个网站问题的紧急电话，需要马上修补。有了 `Git`，我们就不需要同时发布这个补丁和 `iss53` 里作出的修改，也不需要创建和发布该补丁到服务器之前花费大力气来复原这些修改。唯一需要的仅仅是切换回 `master` 分支。

不过在此之前，留心你的暂存区或者工作目录里，那些还没有提交的修改，它会和你即将检出的分支产生冲突从而阻止 `Git` 为你切换分支。切换分支的时候最好保持一个清洁的工作区域。稍后会介绍几个绕过这种问题的办法（分别叫做 `stashing` 和 `commit amending`）。目前已经提交了所有的修改，所以接下来可以正常转换到 `master` 分支：

```
$ git checkout master
Switched to branch "master"
```

此时工作目录中的内容和你在解决问题 `#53` 之前一模一样，你可以集中精力进行紧急修

补。这一点值得牢记：**Git** 会把工作目录的内容恢复为检出某分支时它所指向的那个提交对象的快照。它会自动添加、删除和修改文件以确保目录的内容和你当时提交时完全一样。

接下来，你得进行紧急修补。我们创建一个紧急修补分支 **hotfix** 来开展工作，直到搞定（见图 3-13）：

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```

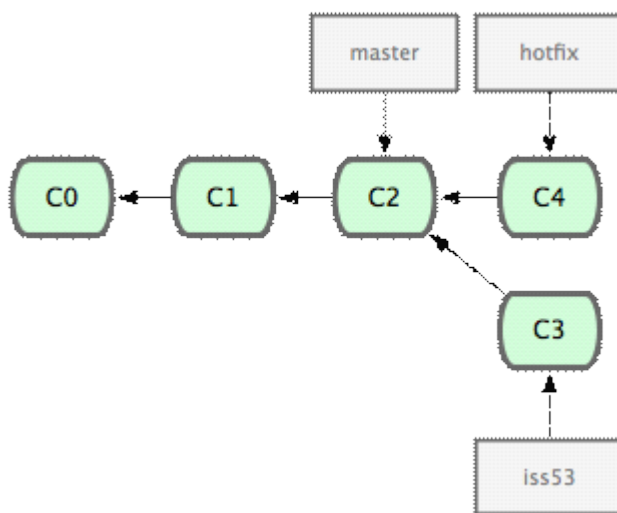


图 3-13. hotfix 分支是从 master 分支所在点分化出来的

有必要作些测试，确保修补是成功的，然后回到 **master** 分支并把它合并进来，然后发布到生产服务器。用 **git merge** 命令来进行合并：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README | 1 -
1 files changed, 0 insertions(+), 1 deletions(-)
```

请注意，合并时出现了“**Fast forward**”的提示。由于当前 **master** 分支所在的提交对象是要并入的 **hotfix** 分支的直接上游，**Git** 只需把 **master** 分支指针直接右移。换句话说，如果顺着分支走下去可以到达另一个分支的话，那么 **Git** 在合并两者时，只会简单地把指针右移，因为这种单线的历史分支不存在任何需要解决的分歧，所以这种合并过程可以称为快

进（Fast forward）。

现在最新的修改已经在当前 **master** 分支所指向的提交对象中了，可以部署到生产服务器上去了（见图 3-14）。

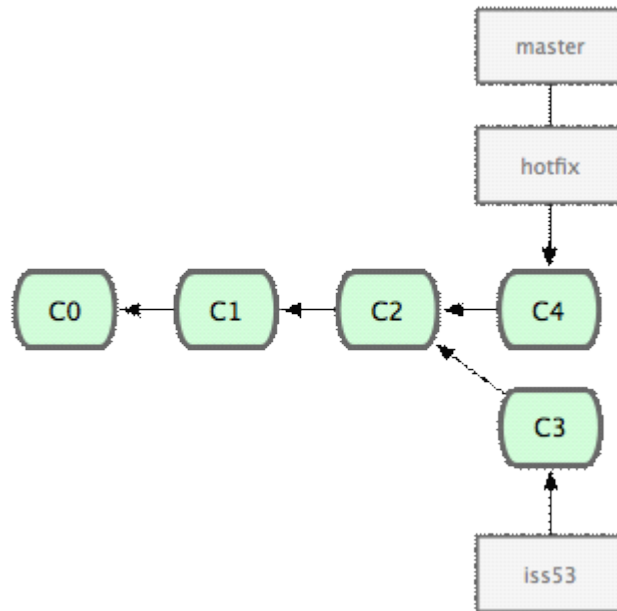


图 3-14. 合并之后，**master** 分支和 **hotfix** 分支指向同一位置。

在那个超级重要的修补发布以后，你想要回到被打扰之前的工作。由于当前 **hotfix** 分支和 **master** 都指向相同的提交对象，所以 **hotfix** 已经完成了历史使命，可以删掉了。使用 **git branch** 的 **-d** 选项执行删除操作：

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

现在回到之前未完成的 **#53** 问题修复分支上继续工作（图 3-15）：

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
1 files changed, 1 insertions(+), 0 deletions(-)
```

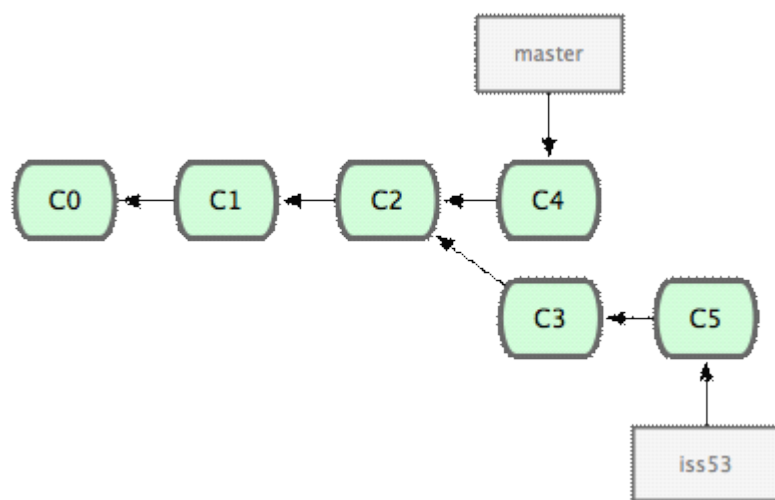


图 3-15. iss53 分支可以不受影响继续推进。

不用担心之前 hotfix 分支的修改内容尚未包含到 iss53 中来。如果确实需要纳入此次修补，可以用 `git merge master` 把 master 分支合并到 iss53；或者等 iss53 完成之后，再将 iss53 分支中的更新并入 master。

分支的合并

在问题 #53 相关的工作完成之后，可以合并回 master 分支。实际操作同前面合并 hotfix 分支差不多，只需回到 master 分支，运行 `git merge` 命令指定要合并进来的分支：

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

请注意，这次合并操作的底层实现，并不同于之前 hotfix 的并入方式。因为这次你的开发历史是从更早的地方开始分叉的。由于当前 master 分支所指向的提交对象（C4）并不是 iss53 分支的直接祖先，Git 不得不进行一些额外处理。就此例而言，Git 会用两个分支的末端（C4 和 C5）以及它们的共同祖先（C2）进行一次简单的三方合并计算。图 3-16 用红框标出了 Git 用于合并的三个提交对象：

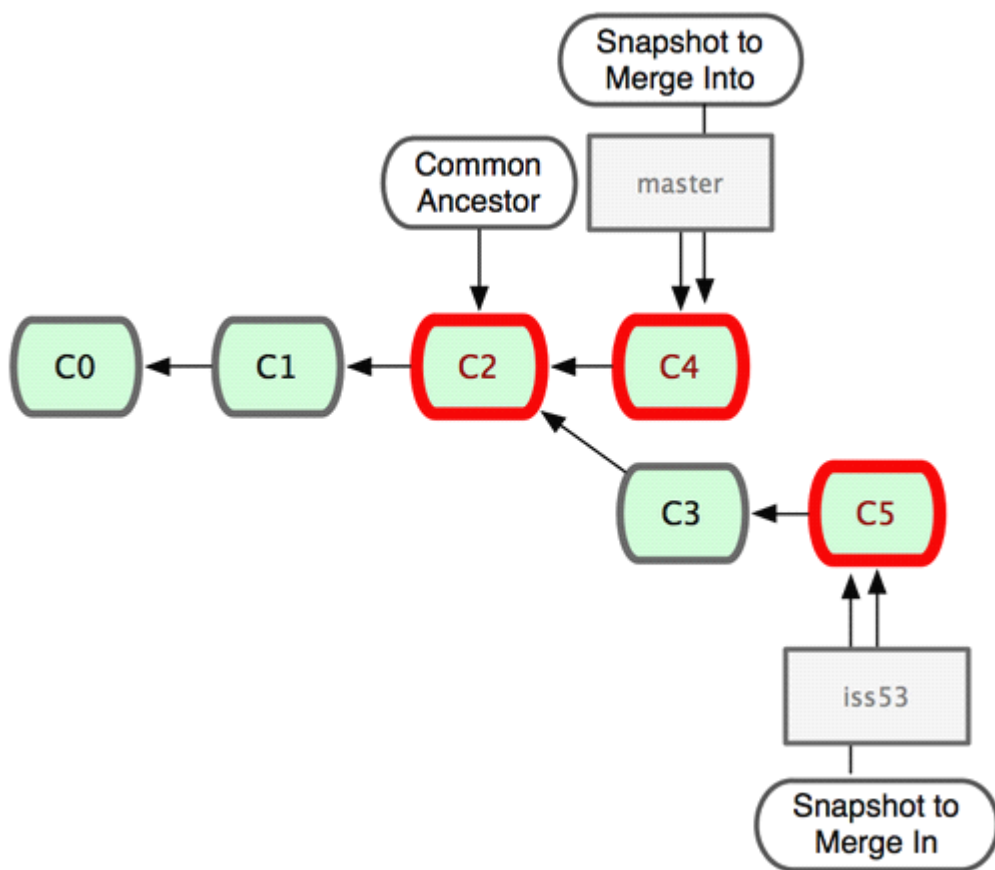


图 3-16. Git 为分支合并自动识别出最佳的同源合并点。

这次，Git 没有简单地把分支指针右移，而是对三方合并后的结果重新做一个新的快照，并自动创建一个指向它的提交对象（C6）（见图 3-17）。这个提交对象比较特殊，它有两个祖先（C4 和 C5）。

值得一提的是 Git 可以自己裁决哪个共同祖先才是最佳合并基础；这和 CVS 或 Subversion（1.5 以后的版本）不同，它们需要开发者手工指定合并基础。所以此特性让 Git 的合并操作比其他系统都要简单不少。

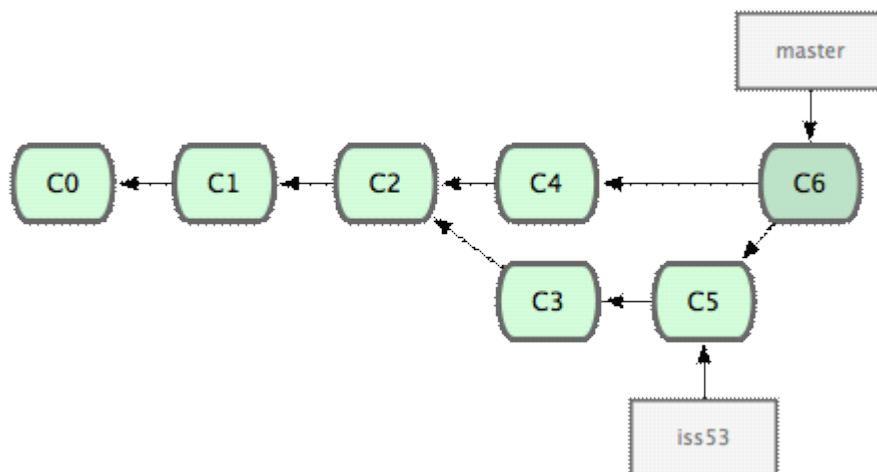


图 3-17. Git 自动创建了一个包含了合并结果的提交对象。

既然之前的工作成果已经合并到 **master** 了，那么 **iss53** 也就没用了。你可以就此删除它，并在问题追踪系统里关闭该问题。

```
$ git branch -d iss53
```

遇到冲突时的分支合并

有时候合并操作并不会如此顺利。如果在不同的分支中都修改了同一个文件的同一部分，**Git** 就无法干净地把两者合到一起（译注：逻辑上说，这种问题只能由人来裁决。）。如果你在解决问题 **#53** 的过程中修改了 **hotfix** 中修改的部分，将得到类似下面的结果：

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git 作了合并，但没有提交，它会停下来等你解决冲突。要看看哪些文件在合并时发生冲突，可以用 **git status** 查阅：

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add
```

```
..." to update what will be committed) # (use "git checkout -- ..." to discard
changes in working directory) # # unmerged: index.html #
```

任何包含未解决冲突的文件都会以未合并（**unmerged**）的状态列出。**Git** 会在有冲突的文件里加入标准的冲突解决标记，可以通过它们来手工定位并解决这些冲突。可以看到此文件包含类似下面这样的部分：

```
<<<<<< HEAD:index.html
```

```
contact : email.support@github.com
```

```
=====
```

```
please contact us at support@github.com
```

```
>>>>>>> iss53:index.html
```

可以看到 `=====` 隔开的上半部分，是 `HEAD`（即 `master` 分支，在运行 `merge` 命令时所切换到分支）中的内容，下半部分是在 `iss53` 分支中的内容。解决冲突的办法无非是二者选其一或者由你亲自整合到一起。比如你可以通过把这段内容替换为下面这样来解决：

```
please contact us at email.support@github.com
```

这个解决方案各采纳了两个分支中的一部分内容，而且我还删除了 `<<<<<<<`，`=====` 和 `>>>>>>>` 这些行。在解决了所有文件里的所有冲突后，运行 `git add` 将把它们标记为已解决状态（译注：实际上就是来一次快照保存到暂存区域。）。因为一旦暂存，就表示冲突已经解决。如果你想用一个有图形界面的工具来解决这些问题，不妨运行 `git mergetool`，它会调用一个可视化的合并工具并引导你解决所有冲突：

```
$ git mergetool
```

```
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
```

```
Merging the files: index.html
```

```
Normal merge conflict for 'index.html':
```

```
{local}: modified
```

```
{remote}: modified
```

```
Hit return to start merge resolution tool (opendiff):
```

如果不想用默认的合并工具（Git 为我默认选择了 **opendiff**，因为我在 **Mac** 上运行了该命令），你可以在上方“**merge tool candidates**”里找到可用的合并工具列表，输入你想用的工具名。我们将在第七章讨论怎样改变环境中的默认值。

退出合并工具以后，**Git** 会询问你合并是否成功。如果回答是，它会为你把相关文件暂存起来，以表明状态为已解决。

再运行一次 **git status** 来确认所有冲突都已解决：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
..." to unstage) # # modified: index.html #
```

如果觉得满意了，并且确认所有冲突都已解决，也就是进入了暂存区，就可以用 **git commit** 来完成这次合并提交。提交的记录差不多是这样：

```
Merge branch 'iss53'
```

```
Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

如果想给将来看这次合并的人一些方便，可以修改该信息，提供更多合并细节。比如你都作了哪些改动，以及这么做的原因。有时候裁决冲突的理由并不直接或明显，有必要略加注解。

3.3 分支的管理

到目前为止，你已经学会了如何创建、合并和删除分支。除此之外，我们还需要学习如何管

理分支，在日后的常规工作中会经常用到下面介绍的管理命令。

git branch 命令不仅仅能创建和删除分支，如果不加任何参数，它会给出当前所有分支的清单：

```
$ git branch
  iss53
* master
  testing
```

注意看 **master** 分支前的 ***** 字符：它表示当前所在的分支。也就是说，如果现在提交更新，**master** 分支将随着开发进度前移。若要查看各个分支最后一个提交对象的信息，运行 **git branch -v**：

```
$ git branch -v
  iss53  93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

要从该清单中筛选出你已经（或尚未）与当前分支合并的分支，可以用 **--merge** 和 **--no-merged** 选项（Git 1.5.6 以上版本）。比如用 **git branch --merge** 查看哪些分支已被并入当前分支（译注：也就是说哪些分支是当前分支的直接上游。）：

```
$ git branch --merged
  iss53
* master
```

之前我们已经合并了 **iss53**，所以在这里会看到它。一般来说，列表中没有 ***** 的分支通常都可以用 **git branch -d** 来删掉。原因很简单，既然已经把它们所包含的工作整合到了其他分支，删掉也不会损失什么。

另外可以用 **git branch --no-merged** 查看尚未合并的工作：

```
$ git branch --no-merged
  testing
```

它会显示还未合并进来的分支。由于这些分支中还包含着尚未合并进来的工作成果，所以简单地用 **git branch -d** 删除该分支会提示错误，因为那样做会丢失数据：

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

不过，如果你确实想要删除该分支上的改动，可以用大写的删除选项 **-D** 强制执行，就像

上面提示信息中给出的那样。

3.4 利用分支进行开发的工作流程

现在我们已经学会了新建分支和合并分支，可以（或应该）用它来做点什么呢？在本节，我们会介绍一些利用分支进行开发的工作流程。而正是由于分支管理的便捷，才衍生出了这类典型的工作模式，你可以根据项目的实际情况选择一种用用看。

长期分支

由于 Git 使用简单的三方合并，所以就算在较长一段时间内，反复多次把某个分支合并到另一分支，也不是什么难事。也就是说，你可以同时拥有多个开放的分支，每个分支用于完成特定的任务，随着开发的推进，你可以随时把某个特性分支的成果并到其他分支中。

许多使用 Git 的开发者都喜欢用这种方式来开展工作，比如仅在 **master** 分支中保留完全稳定的代码，即已经发布或即将发布的代码。与此同时，他们还有一个名为 **develop** 或 **next** 的平行分支，专门用于后续的开发，或仅用于稳定性测试 — 当然并不是说一定要绝对稳定，不过一旦进入某种稳定状态，便可以把它合并到 **master** 里。这样，在确保这些已完成的特性分支（短期分支，比如之前的 **iss53** 分支）能够通过所有测试，并且不会引入更多错误之后，就可以并到主干分支中，等待下一次的发布。

本质上我们刚才谈论的，是随着提交对象不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支总是比较靠前（见图 3-18）。

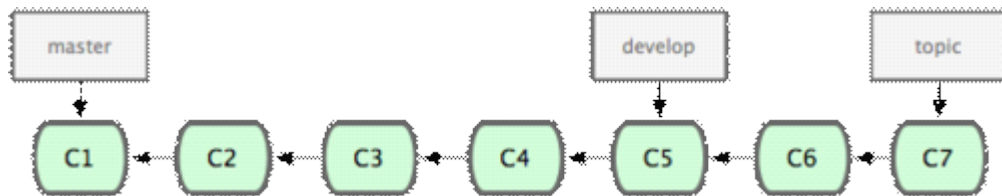


图 3-18. 稳定分支总是比较老旧。

或者把它们想象成工作流水线，或许更好理解一些，经过测试的提交对象集合被遴选到更稳定的流水线（见图 3-19）。

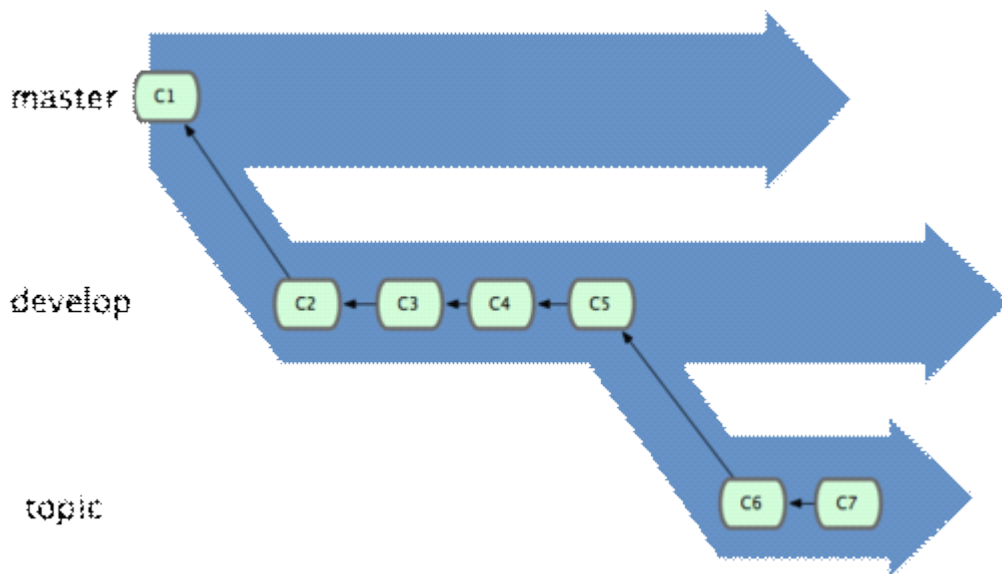


图 3-19. 想象成流水线可能会容易点。

你可以用这招维护不同层次的稳定性。某些大项目还会有个 `proposed`（建议）或 `pu`（`proposed updates`, 建议更新）分支，它包含着那些可能还没有成熟到进入 `next` 或 `master` 的内容。这么做的目的是拥有不同层次的稳定性：当这些分支进入到更稳定的水平时，再把它们合并到更高层分支中去。再次说明下，使用多个长期分支的做法并非必需，不过一般来说，对于特大型项目或特复杂的项目，这么做确实更容易管理。

特性分支

在任何规模的项目中都可以使用特性（`Topic`）分支。一个特性分支是指一个短期的，用来实现单一特性或与其相关工作的分支。可能你在以前的版本控制系统里从未做过类似这样的事情，因为通常创建与合并分支消耗太大。然而在 `Git` 中，一天之内建立、使用、合并再删除多个分支是常见的事。

我们在上节的例子里已经见过这种用法了。我们创建了 `iss53` 和 `hotfix` 这两个特性分支，在提交了若干更新后，把它们合并到主干分支，然后删除。该技术允许你迅速且完全的进行语境切换 — 因为你的工作分散在不同的流水线里，每个分支里的改变都和它的目标特性相关，浏览代码之类的事情因而变得更简单了。你可以把作出的改变保持在特性分支中几分钟，几天甚至几个月，等它们成熟以后再合并，而不用在乎它们建立的顺序或者进度。

现在来看一个实际的例子。请看图 3-20，由下往上，起先我们在 `master` 工作到 `C1`，然后开始一个新分支 `iss91` 尝试修复 91 号缺陷，提交到 `C6` 的时候，又冒出一个解决该问题的新办法，于是从之前 `C4` 的地方又分出一个分支 `iss91v2`，干到 `C8` 的时候，又回到主干 `master` 中提交了 `C9` 和 `C10`，再回到 `iss91v2` 继续工作，提交 `C11`，接着，又冒出个不太确定的想法，从 `master` 的最新提交 `C10` 处开了个新的分支 `dumbidea` 做些试验。

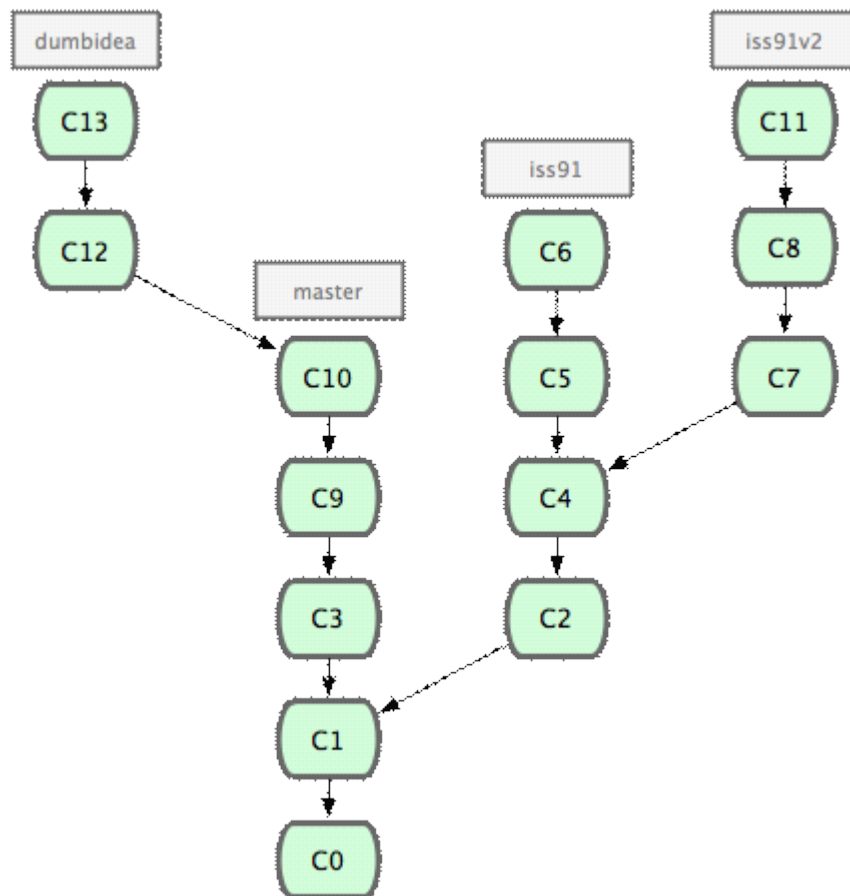


图 3-20. 拥有多个特性分支的提交历史。

现在，假定两件事情：我们最终决定使用第二个解决方案，即 `iss91v2` 中的办法；另外，我们把 `dumbidea` 分支拿给同事们看了以后，发现它竟然是个天才之作。所以接下来，我们准备抛弃原来的 `iss91` 分支（实际上会丢弃 `C5` 和 `C6`），直接在主干中并入另外两个分支。最终的提交历史将变成图 3-21 这样：

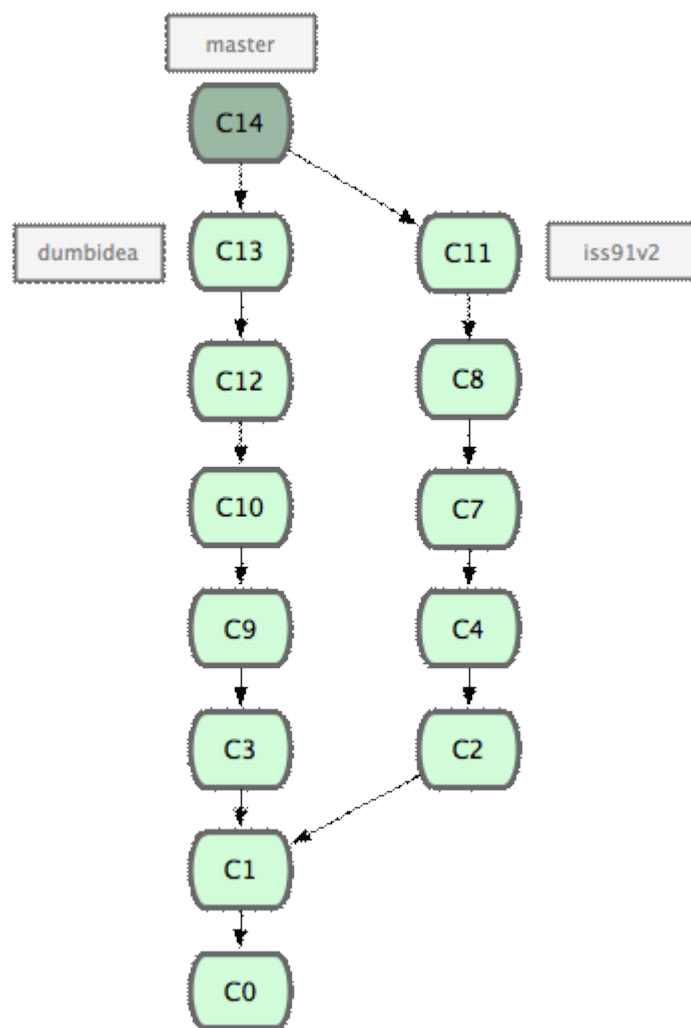


图 3-21. 合并了 `dumbidea` 和 `iss91v2` 后的分支历史。

请务必牢记这些分支全部都是本地分支，这一点很重要。当你在使用分支及合并的时候，一切都是在你自己的 `Git` 仓库中进行的 — 完全不涉及与服务器的交互。

3.5 远程分支

远程分支（`remote branch`）是对远程仓库中的分支的索引。它们是一些无法移动的本地分支；只有在 `Git` 进行网络交互时才会更新。远程分支就像是书签，提醒着你上次连接远程仓库时上面各分支的位置。

我们用（远程仓库名）/（分支名）这样的形式表示远程分支。比如我们想看看上次同 `origin` 仓库通讯时 `master` 的样子，就应该查看 `origin/master` 分支。如果你和同伴一起修复某个问题，但他们先推送了一个 `iss53` 分支到远程仓库，虽然你可能也有一个本地的 `iss53` 分支，但指向服务器上最新更新的他应该是 `origin/iss53` 分支。

可能有点乱，我们不妨举例说明。假设你们团队有个地址为 `git.ourcompany.com` 的 Git 服务器。如果你从这里克隆，Git 会自动为你将此远程仓库命名为 `origin`，并下载其中所有的数据，建立一个指向它的 `master` 分支的指针，在本地命名为 `origin/master`，但你无法在本地更改其数据。接着，Git 建立一个属于你自己的本地 `master` 分支，始于 `origin` 上 `master` 分支相同的位置，你可以就此开始工作（见图 3-22）：

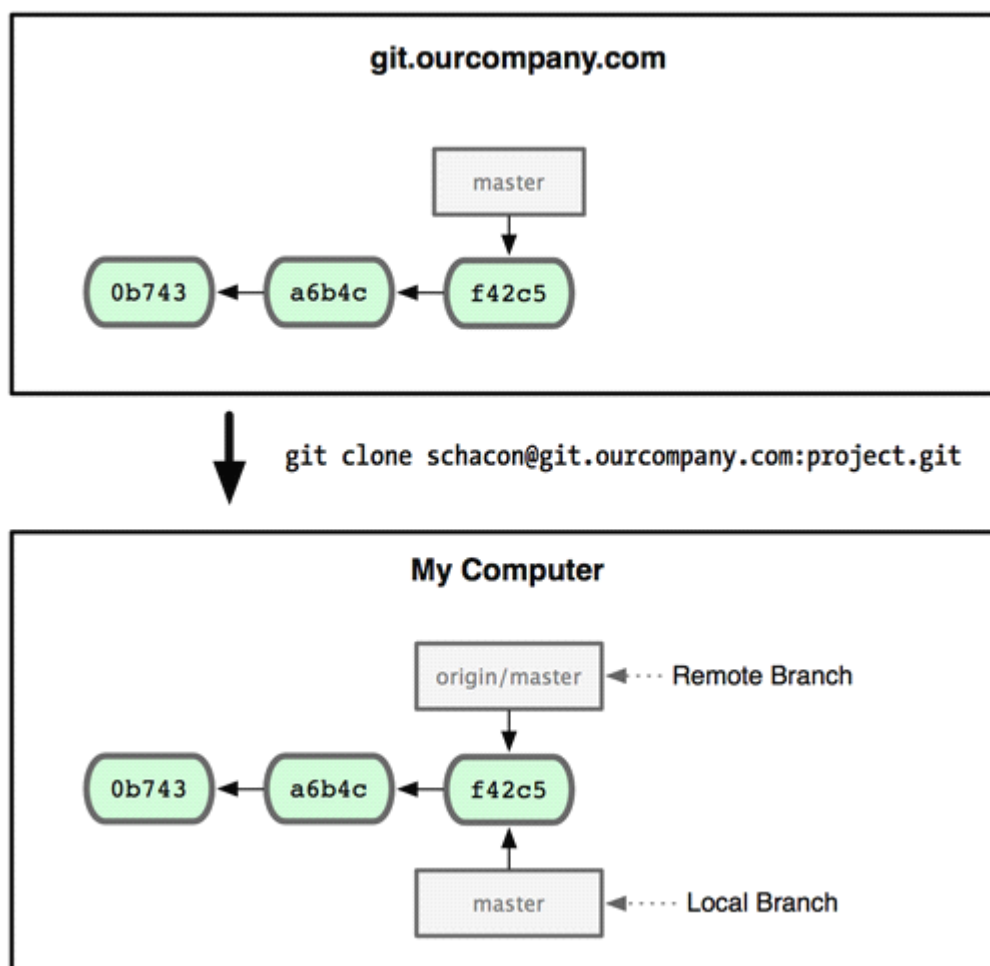


图 3-22. 一次 Git 克隆会建立你自己的本地分支 `master` 和远程分支 `origin/master`，它们都指向 `origin/master` 分支的最后一次提交。

如果你在本地的 `master` 分支做了些改动，与此同时，其他人向 `git.ourcompany.com` 推送了他们的更新，那么服务器上的 `master` 分支就会向前推进，而于此同时，你在本地的提交历史正朝向不同方向发展。不过只要你不和服务器通讯，你的 `origin/master` 指针仍然保持原位不会移动（见图 3-23）。

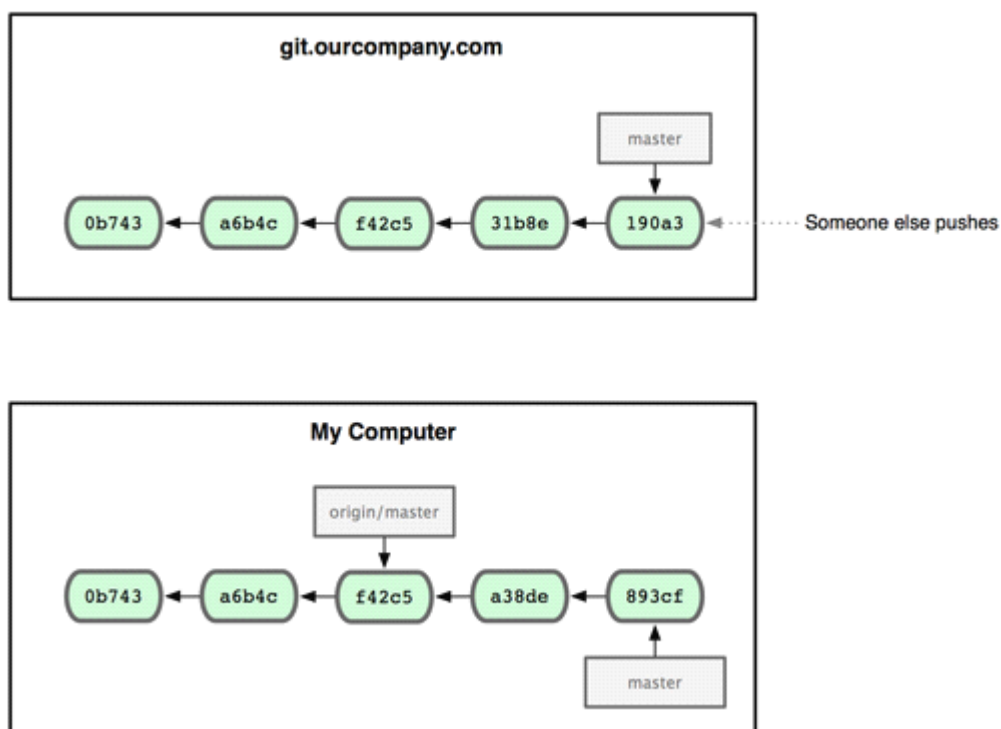


图 3-23. 在本地工作的同时有人向远程仓库推送内容会让提交历史开始分流。

可以运行 `git fetch origin` 来同步远程服务器上的数据到本地。该命令首先找到 `origin` 是哪个服务器（本例为 `git.ourcompany.com`），从上面获取你尚未拥有的数据，更新你本地的数据库，然后把 `origin/master` 的指针移到它最新的位置上（见图 3-24）。

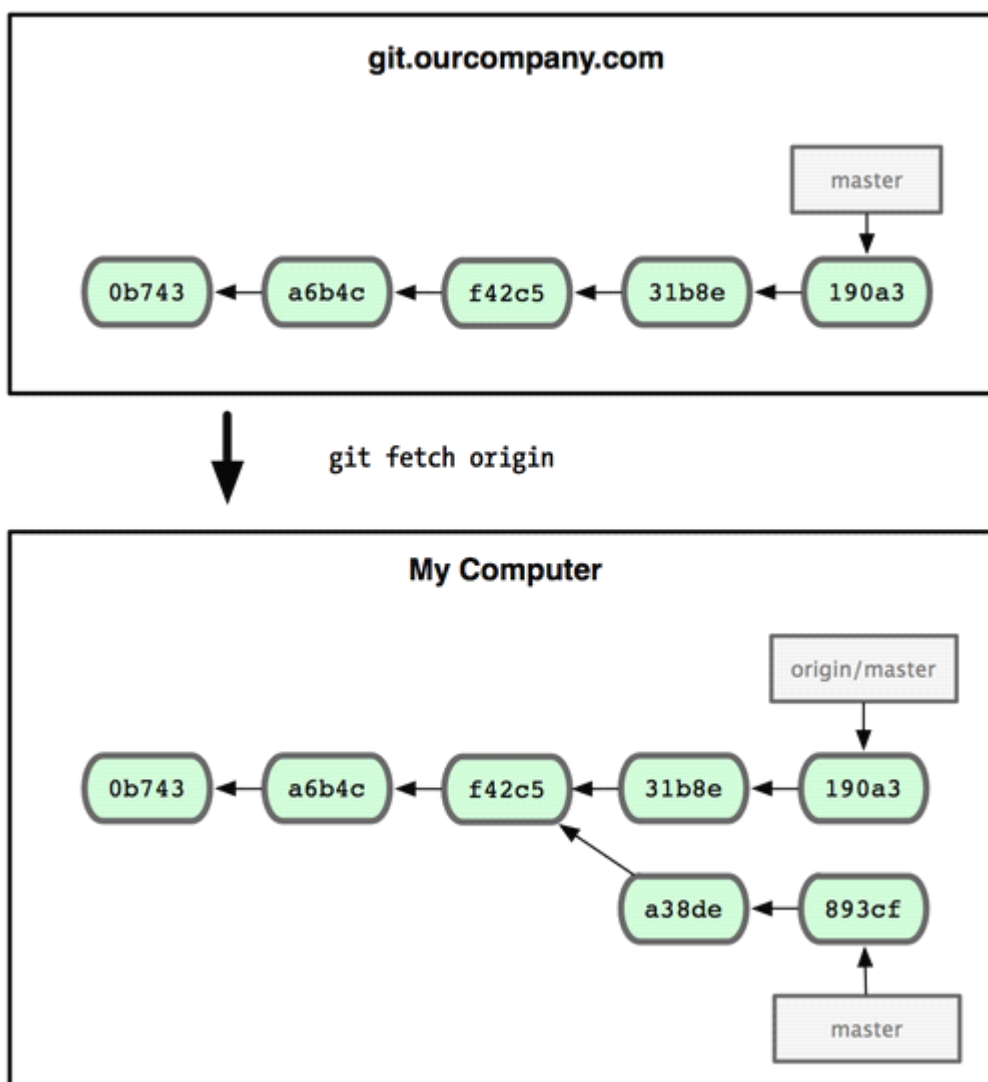


图 3-24. `git fetch` 命令会更新 `remote` 索引。

为了演示拥有多个远程分支（在不同的远程服务器上）的项目是如何工作的，我们假设你还有另一个仅供你的敏捷开发小组使用的内部服务器 `git.team1.ourcompany.com`。可以用第二章中提到的 `git remote add` 命令把它加为当前项目的远程分支之一。我们把它命名为 `teamone`，以便代替原始的 Git 地址（见图 3-25）。

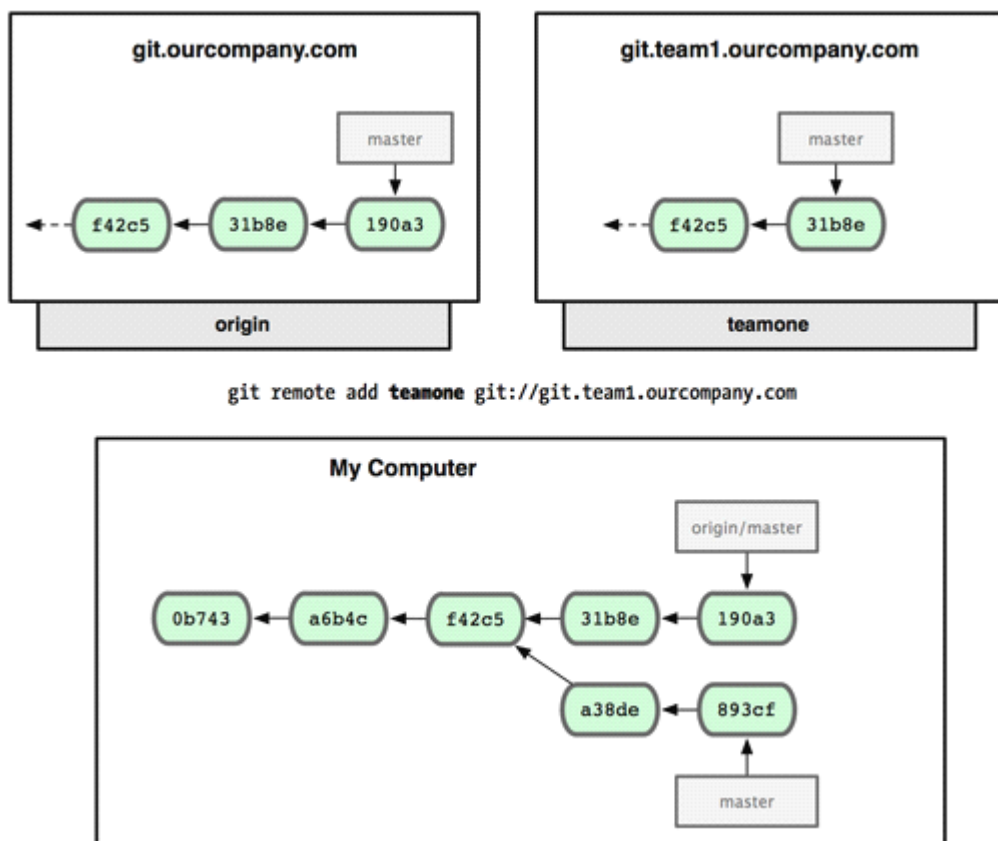


图 3-25. 把另一个服务器加为远程仓库

现在你可以用 `git fetch teamone` 来获取小组服务器上你还没有的数据了。由于当前该服务器上的内容是你 `origin` 服务器上的子集，Git 不会下载任何数据，而只是简单地创建一个名为 `teamone/master` 的分支，指向 `teamone` 服务器上 `master` 分支所在的提交对象 31b8e（见图 3-26）。

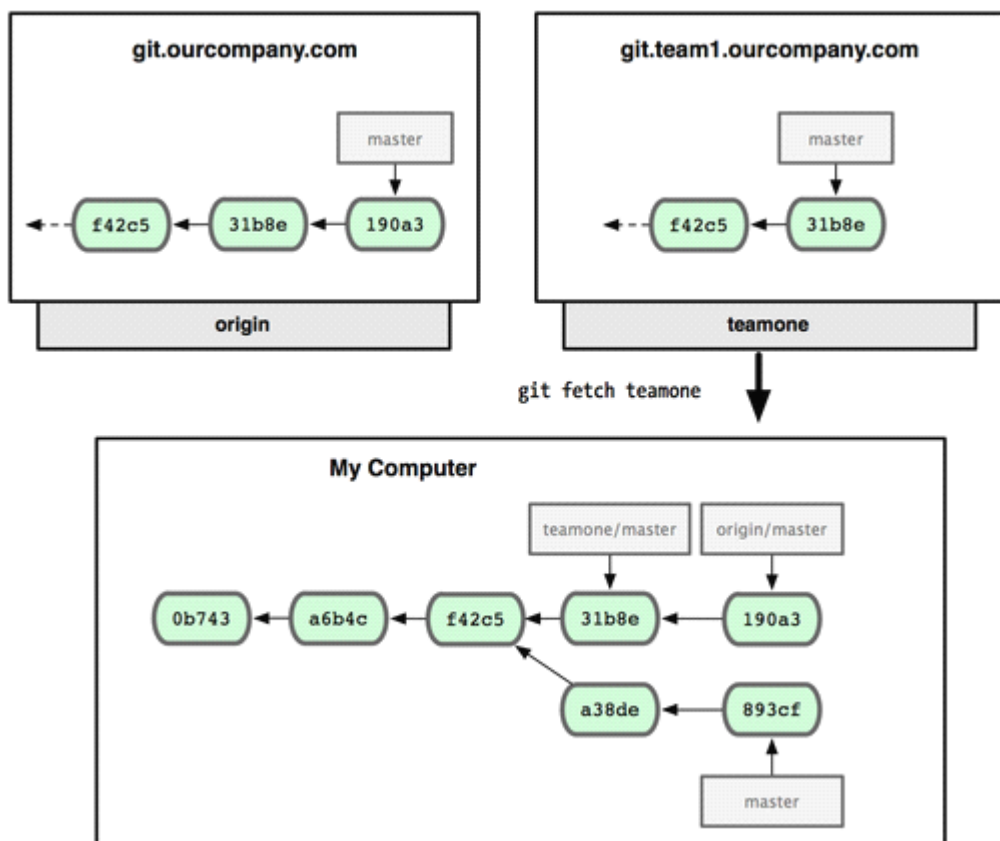


图 3-26. 你在本地有了一个指向 teamone 服务器上 master 分支的索引。

推送本地分支

要想和其他人分享某个本地分支，你需要把它推送到一个你拥有写权限的远程仓库。你的本地分支不会被自动同步到你引入的远程服务器上，除非你明确执行推送操作。换句话说，对于无意分享的分支，你尽管保留为私人分支好了，而只推送那些协同工作要用到的特性分支。

如果你有个叫 `serverfix` 的分支需要和他人一起开发，可以运行 `git push` (远程仓库名) (分支名)：

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

这其实有点像条捷径。Git 自动把 `serverfix` 分支名扩展为 `refs/heads/serverfix:refs/heads/serverfix`，意为“取出我在本地的 `serverfix` 分支，推送到远

程仓库的 `serverfix` 分支中去”。我们将在第九章进一步介绍 `refs/heads/` 部分的细节，不过一般使用的时候都可以省略它。也可以运行 `git push origin serverfix:serverfix` 来实现相同的效果，它的意思是“上传我本地的 `serverfix` 分支到远程仓库中去，仍旧称它为 `serverfix` 分支”。通过此语法，你可以把本地分支推送到某个命名不同的远程分支：若想把远程分支叫作 `awesomebranch`，可以用 `git push origin serverfix:awesomebranch` 来推送数据。

接下来，当你的协作者再次从服务器上获取数据时，他们将得到一个新的远程分支

`origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

值得注意的是，在 `fetch` 操作下载好新的远程分支之后，你仍然无法在本地编辑该远程仓库中的分支。换句话说，在本例中，你不会有一个新的 `serverfix` 分支，有的只是一个你无法移动的 `origin/serverfix` 指针。

如果要把该内容合并到当前分支，可以运行 `git merge origin/serverfix`。如果想要一份自己的 `serverfix` 来开发，可以在远程分支的基础上分化出一个新的分支来：

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

这会切换到新建的 `serverfix` 本地分支，其内容同远程分支 `origin/serverfix` 一致，这样你就可以在里面继续开发了。

跟踪远程分支

从远程分支 `checkout` 出来的本地分支，称为跟踪分支(tracking branch)。跟踪分支是一种和远程分支有直接联系的本地分支。在跟踪分支里输入 `git push`，Git 会自行推断应该向哪个服务器的哪个分支推送数据。反过来，在这些分支里运行 `git pull` 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

在克隆仓库时，Git 通常会创建一个名为 `master` 的分支来跟踪 `origin/master`。这正是 `git push` 和 `git pull` 一开始就能正常工作的原因。当然，你可以随心所欲地设定为其它跟踪分支，比如 `origin` 上除了 `master` 之外的其它分支。刚才我们已经看到了这样的例子：`git checkout -b [分支名] [远程名]/[分支名]`。如果你有 1.6.2 以上版本的 Git，还可以

用`--track` 选项简化：

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

要为本地分支设定不同于远程分支的名字，只需在前个版本的命令里换个名字：

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

现在你的本地分支 `sf` 会自动向 `origin/serverfix` 推送和抓取数据了。

删除远程分支

如果不再需要某个远程分支了，比如搞定了某个特性并把它合并进了远程的 `master` 分支（或任何其他存放稳定代码的地方），可以用这个非常无厘头的语法来删除它：`git push [远程名]:[分支名]`。如果想在服务器上删除 `serverfix` 分支，运行下面的命令：

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
- [deleted]          serverfix
```

咚！服务器上的分支没了。你最好特别留心这一页，因为你一定会用到那个命令，而且你可能会忘掉它的语法。有种方便记忆这条命令的方法：记住我们不久前见过的 `git push [远程名][本地分支]:[远程分支]` 语法，如果省略 `[本地分支]`，那就等于是在说“在这里提取空白然后把它变成`[远程分支]`”。

3.6 分支的衍合

把一个分支整合到另一个分支的办法有两种：`merge` 和 `rebase`（译注：`rebase` 的翻译暂定为“衍合”，大家知道就可以了。）。在本章我们会学习什么是衍合，如何使用衍合，为什么衍合操作如此富有魅力，以及我们应该在什么情况下使用衍合。

基本的衍合操作

请回顾之前有关合并的一节（见图 3-27），你会看到开发进程分叉到两个不同分支，又各自提交了更新。

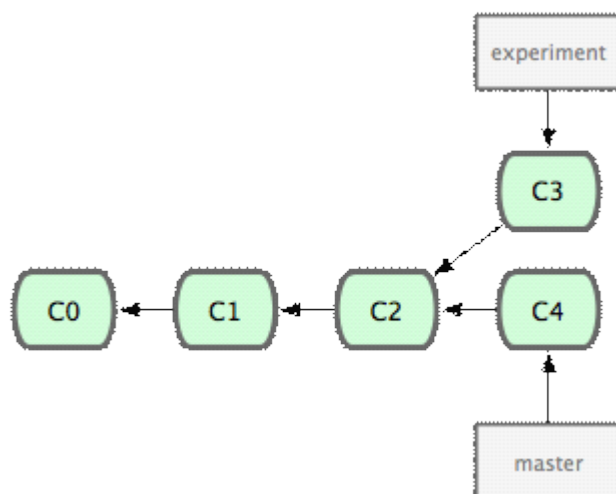


图 3-27. 最初分叉的提交历史。

之前介绍过，最容易的整合分支的方法是 `merge` 命令，它会把两个分支最新的快照（C3 和 C4）以及二者最新的共同祖先（C2）进行三方合并，合并的结果是产生一个新的提交对象（C5）。如图 3-28 所示：

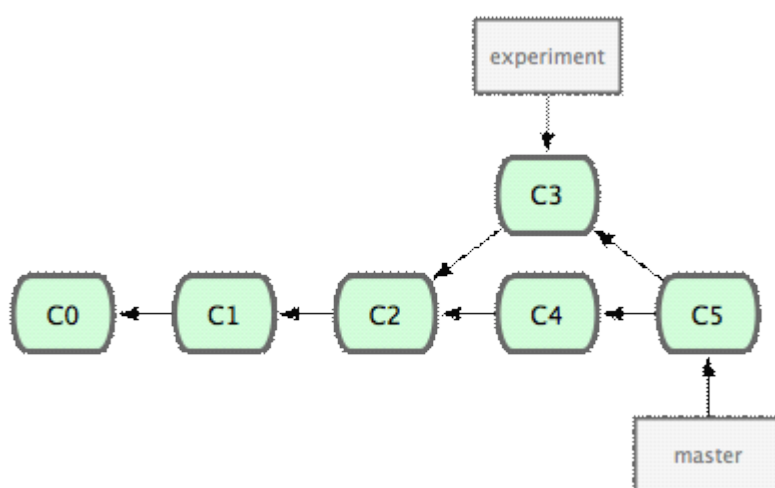


图 3-28. 通过合并一个分支来整合分叉了的历史。

其实，还有另外一个选择：你可以把在 C3 里产生的变化补丁在 C4 的基础上重新打一遍。在 Git 里，这种操作叫做衍合（`rebase`）。有了 `rebase` 命令，就可以把在一个分支里提交的改变移到另一个分支里重放一遍。

在上面这个例子中，运行：

```
$ git checkout experiment
```

```
$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: added staged command
```

它的原理是回到两个分支最近的共同祖先，根据当前分支（也就是要进行衍合的分支 **experiment**）后续的历次提交对象（这里只有一个 **C3**），生成一系列文件补丁，然后以基底分支（也就是主干分支 **master**）最后一个提交对象（**C4**）为新的出发点，逐个应用之前准备好的补丁文件，最后会生成一个新的合并提交对象（**C3'**），从而改写 **experiment** 的提交历史，使它成为 **master** 分支的直接下游，如图 3-29 所示：

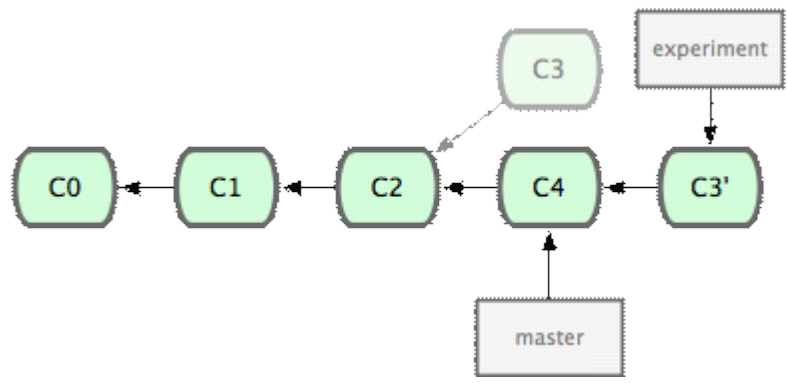


图 3-29. 把 **C3** 里产生的改变到 **C4** 上重演一遍。

现在回到 **master** 分支，进行一次快速合并（见图 3-30）：

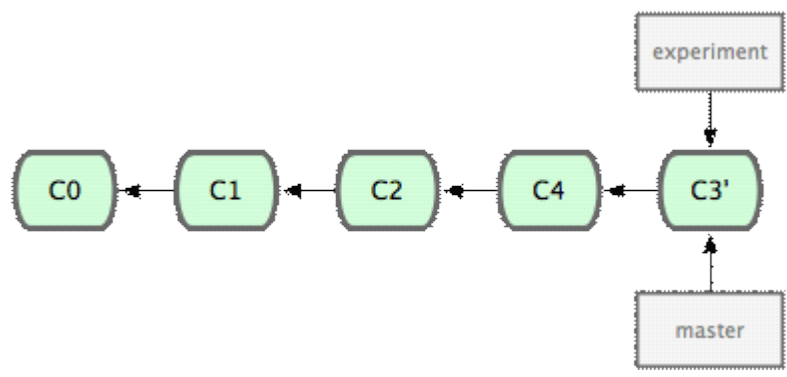


图 3-30. **master** 分支的快速。

现在的 **C3'** 对应的快照，其实和普通的三方合并，即上个例子中的 **C5** 对应的快照内容一模一样了。虽然最后整合得到的结果没有任何区别，但衍合能产生一个更为整洁的提交历史。如果视察一个衍合过的分支的历史记录，看起来会更 清楚：仿佛所有修改都是在一根线上先后进行的，尽管实际上它们原本是同时并行发生的。

一般我们使用衍合的目的，是想要得到一个能在远程分支上干净应用的补丁 —— 比如某些项目你不是维护者，但想帮点忙的话，最好用衍合：先在自己的一个分支里进行开发，当准备向主项目提交补丁的时候，根据最新的 **origin/master** 进行一次衍合操作然后再提交，这样维护者就不需要做任何整合工作（译注：实际上是把解决分支补丁同最新主干代码之间冲突的责任，化转为由提交补丁的人来解决。），只需根据你提供的仓库地址作一次快速合并，或者直接采纳你提交的补丁。

请注意，合并结果中最后一次提交所指向的快照，无论是通过衍合，还是三方合并，都会得到相同的快照内容，只不过提交历史不同罢了。衍合是按照每行的修改次序重演一遍修改，而合并是把最终结果合在一起。

有趣的衍合

衍合也可以放到其他分支进行，并不一定非得根据分化之前的分支。以图 3-31 的历史为例，我们为了给服务器端代码添加一些功能而创建了特性分支 **server**，然后提交 **C3** 和 **C4**。然后又从 **C3** 的地方再增加一个 **client** 分支来对客户端代码进行一些相应修改，所以提交了 **C8** 和 **C9**。最后，又回到 **server** 分支提交了 **C10**。

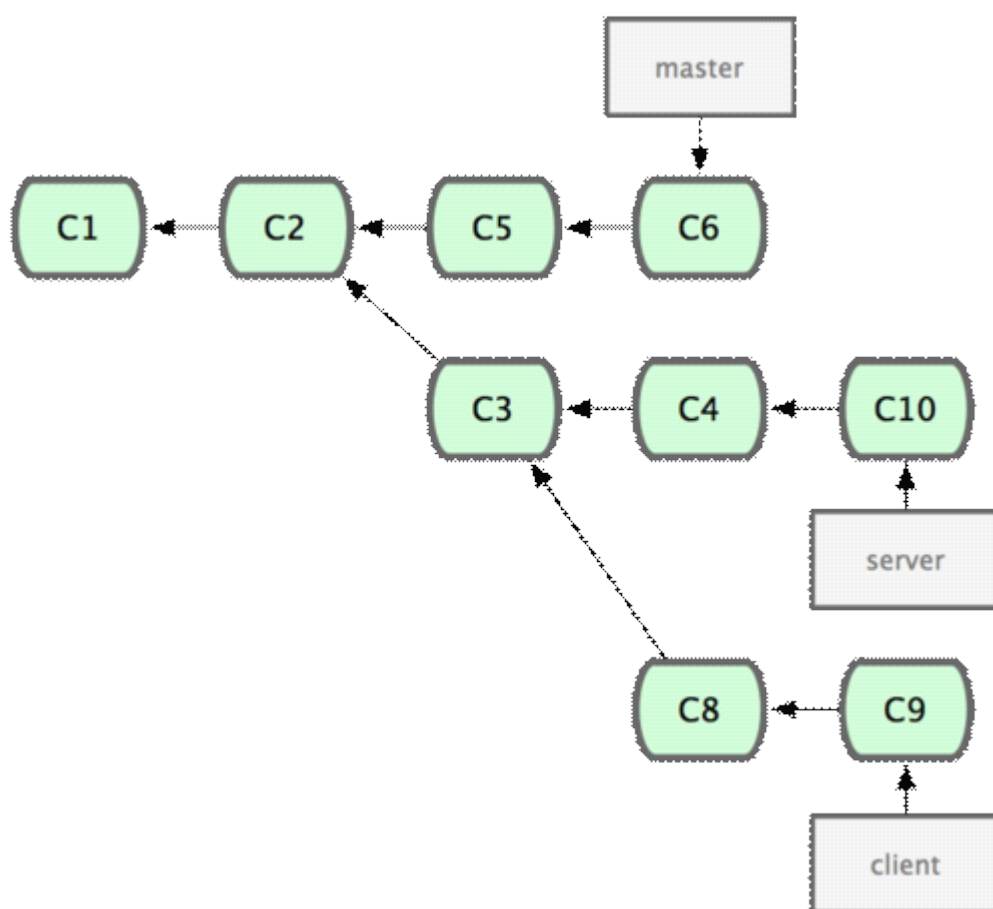


图 3-31. 从一个特性分支里再分出一个特性分支的历史。

假设在接下来的一次软件发布中，我们决定先把客户端的修改并到主线中，而暂缓并入服务端软件的修改（因为还需要进一步测试）。这个时候，我们就可以把基于 **server** 分支而非 **master** 分支的改变（即 **C8** 和 **C9**），跳过 **server** 直接放到 **master** 分支中重演一遍，但这需要用 **git rebase** 的 **--onto** 选项指定新的基底分支 **master**：

```
$ git rebase --onto master server client
```

这好比在说：“取出 **client** 分支，找出 **client** 分支和 **server** 分支的共同祖先之后的变化，

然后把它们在 **master** 上重演一遍”。是不是有点复杂？不过它的结果如图 3-32 所示，非常酷（译注：虽然 **client** 里的 C8, C9 在 C3 之后，但这仅表明时间上的先后，而非在 C3 修改的基础上进一步改动，因为 **server** 和 **client** 这两个分支对应的代码应该是两套文件，虽然这么说不是很严格，但应理解为在 C3 时间点之后，对另外的文件所做的 C8, C9 修改，放到主干重演。）：

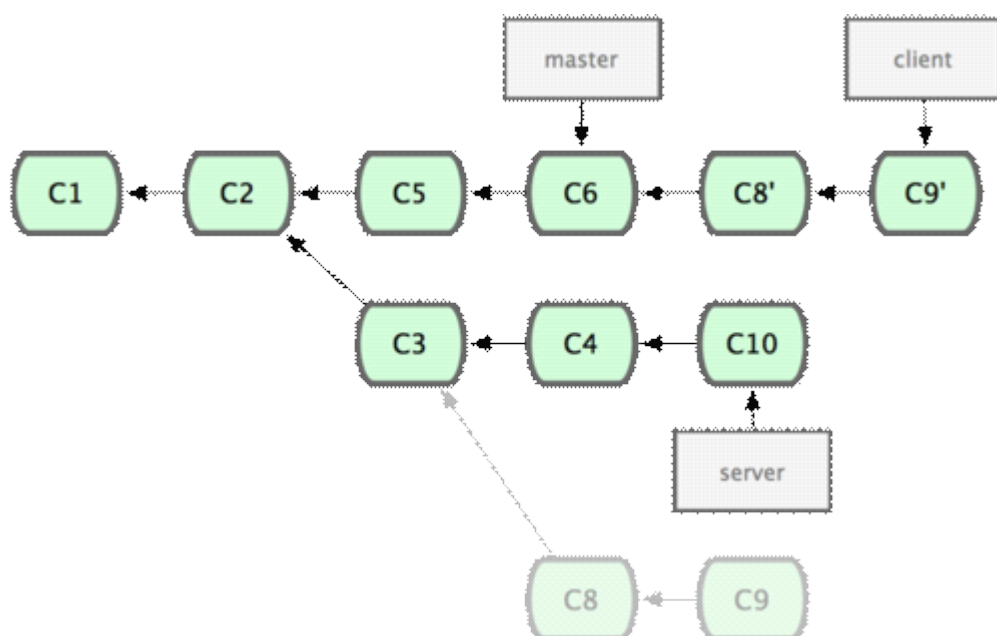


图 3-32. 将特性分支上的另一个特性分支衍合到其他分支。

现在可以快进 **master** 分支了（见图 3-33）：

```
$ git checkout master  
$ git merge client
```

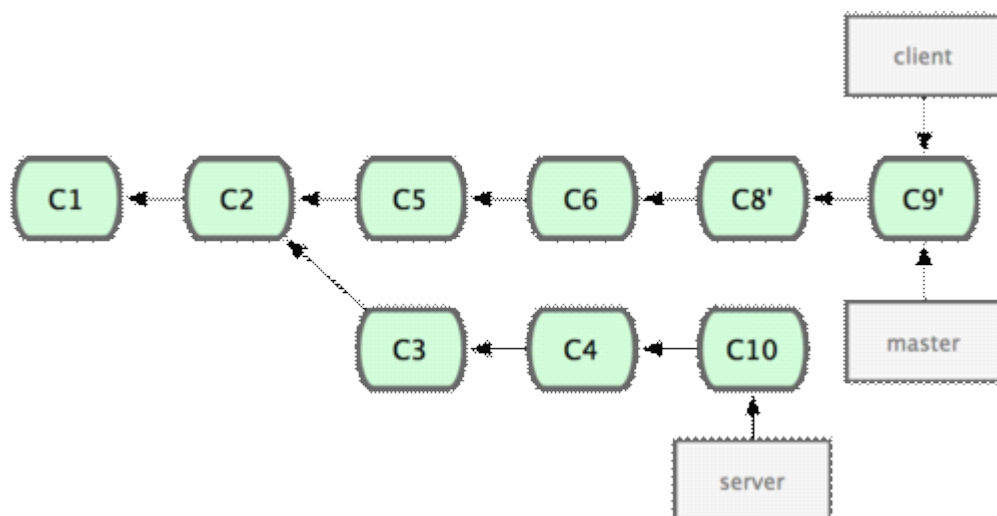


图 3-33. 快进 `master` 分支，使之包含 `client` 分支的变化。

现在我们决定把 `server` 分支的变化也包含进来。我们可以直接把 `server` 分支衍合到 `master`，而不用手工切换到 `server` 分支后再执行衍合操作 — `git rebase [主分支] [特性分支]` 命令会先取出特性分支 `server`，然后在主分支 `master` 上重演：

```
$ git rebase master server
```

于是，`server` 的进度应用到 `master` 的基础上，如图 3-34 所示：

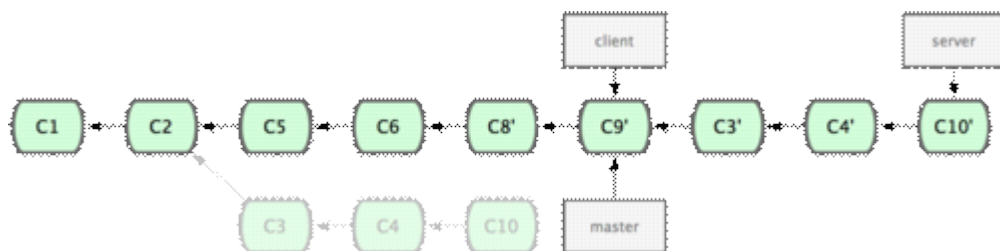


图 3-34. 在 `master` 分支上衍合 `server` 分支。

然后就可以快进主干分支 `master` 了：

```
$ git checkout master
```

```
$ git merge server
```

现在 `client` 和 `server` 分支的变化都已经集成到主干分支来了，可以删掉它们了。最终我们的提交历史会变成图 3-35 的样子：

```
$ git branch -d client
```

```
$ git branch -d server
```

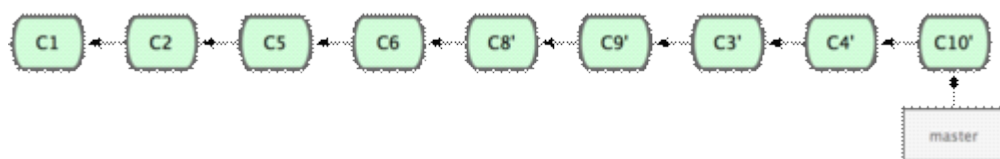


图 3-35. 最终的提交历史

衍合的风险

呃，奇妙的衍合也并非完美无缺，要用它得遵守一条准则：

一旦分支中的提交对象发布到公共仓库，就千万不要对该分支进行衍合操作。

如果你遵循这条金科玉律，就不会出差错。否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

在进行衍合的时候，实际上抛弃了一些现存的提交对象而创造了一些类似但不同的新的提交

对象。如果你把原来分支中的提交对象发布出去，并且其他人更新下载后在其基础上开展工作，而稍后你又用 `git rebase` 抛弃这些提交对象，把新的重演后的提交对象发布出去的话，你的合作者就不得不重新合并他们的工作，这样当你再次从他们那里获取内容时，提交历史就会变得一团糟。

下面我们用一个实际例子来说明为什么公开的衍合会带来问题。假设你从一个中央服务器克隆然后在它的基础上搞了一些开发，提交历史类似图 3-36 所示：

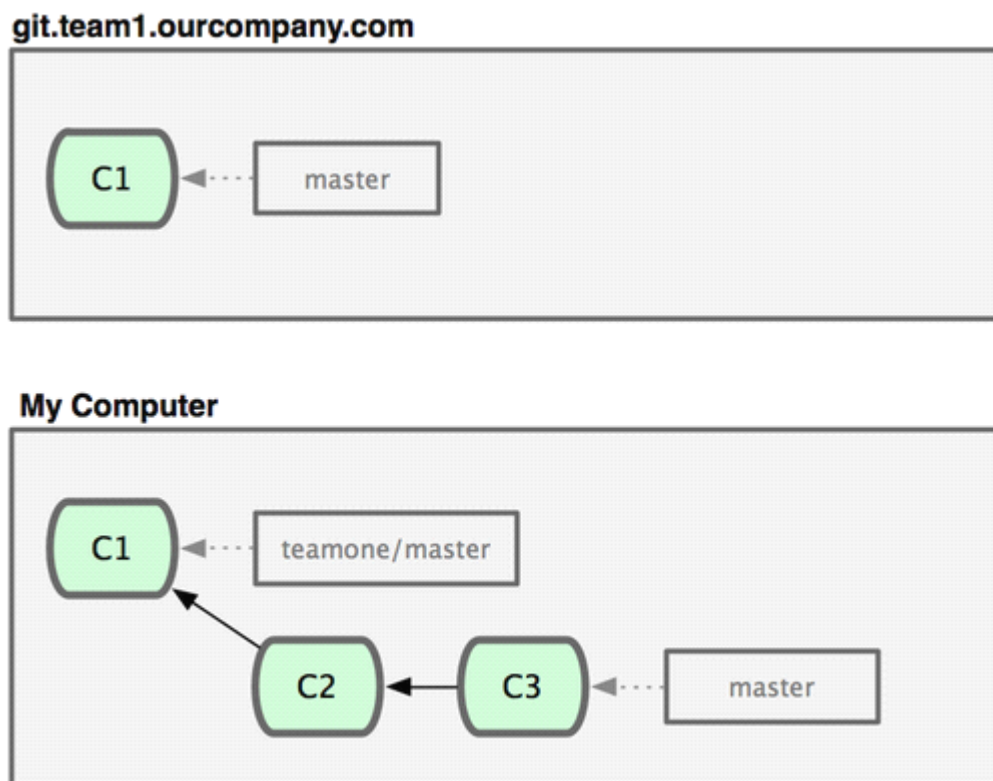


图 3-36. 克隆一个仓库，在其基础上工作一番。

现在，某人在 C1 的基础上做了些改变，并合并他自己的分支得到结果 C6，推送到中央服务器。当你抓取并合并这些数据到你本地的开发分支中后，会得到合并结果 C7，历史提交会变成图 3-37 这样：

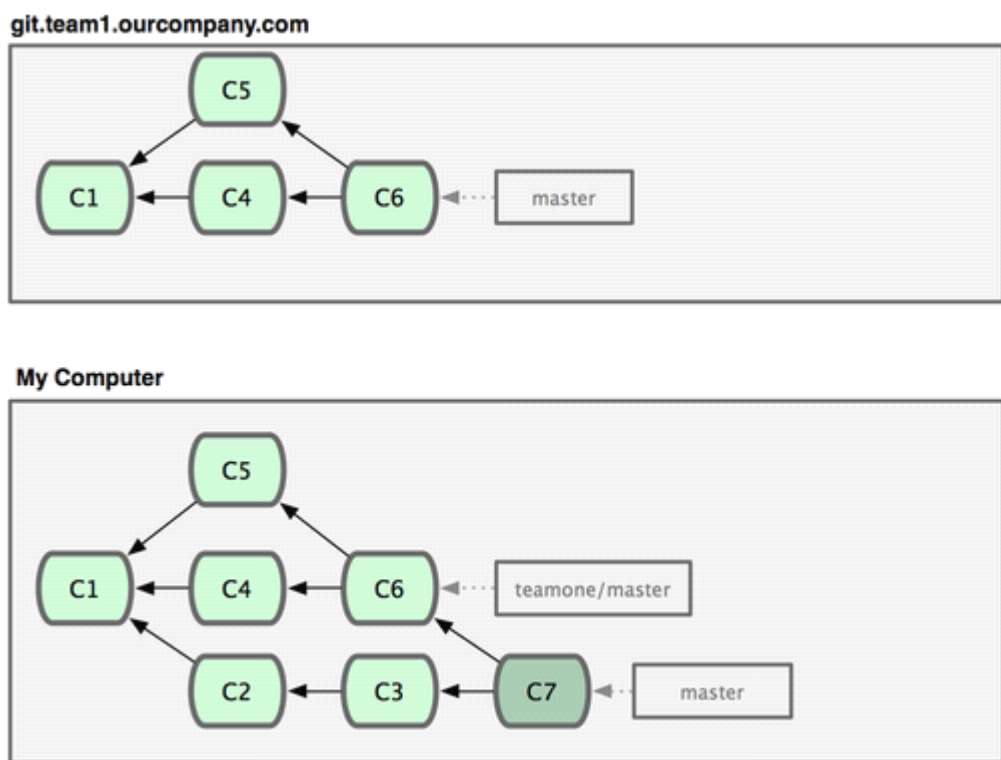


图 3-37. 抓取他人提交，并入自己主干。

接下来，那个推送 C6 上来的人决定用衍合取代之前的合并操作；继而又用 `git push --force` 覆盖了服务器上的历史，得到 C4'。而之后当你再从服务器上下载最新提交后，会得到：

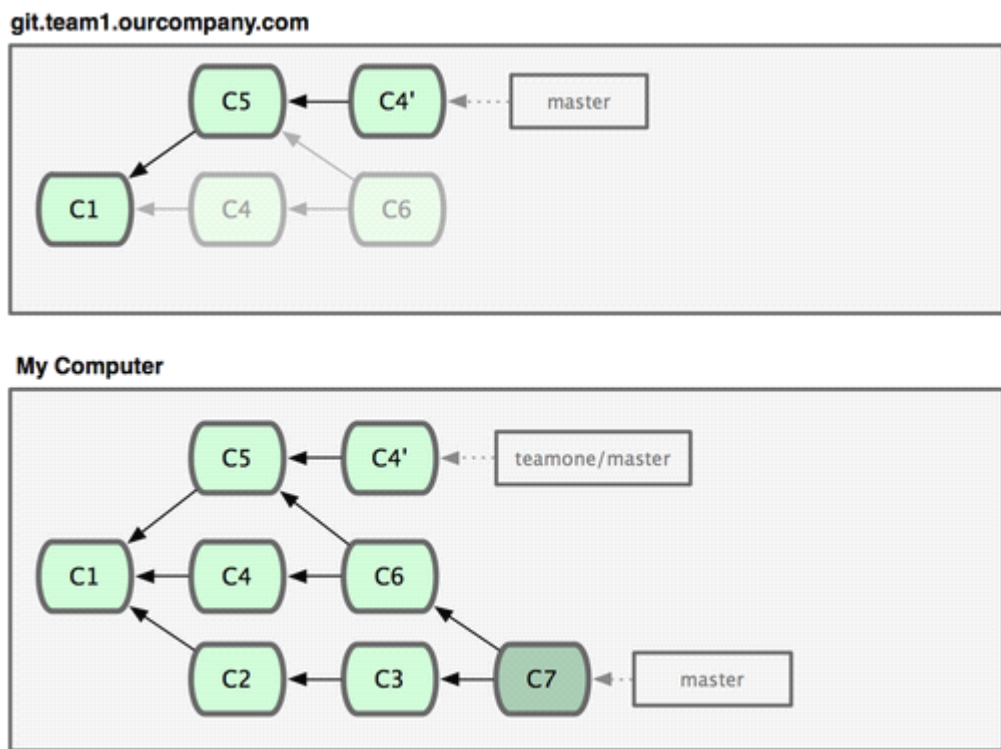


图 3-38. 有人推送了衍合后得到的 C4'，丢弃了你作为开发基础的 C4 和 C6。

下载更新后需要合并，但此时衍合产生的提交对象 C4' 的 SHA-1 校验值和之前 C4 完全不同，所以 Git 会把它当作新的提交对象处理，而实际上此刻你的提交历史 C7 中早已包含了 C4 的修改内容，于是合并操作会把 C7 和 C4' 合并为 C8（见图 3-39）：

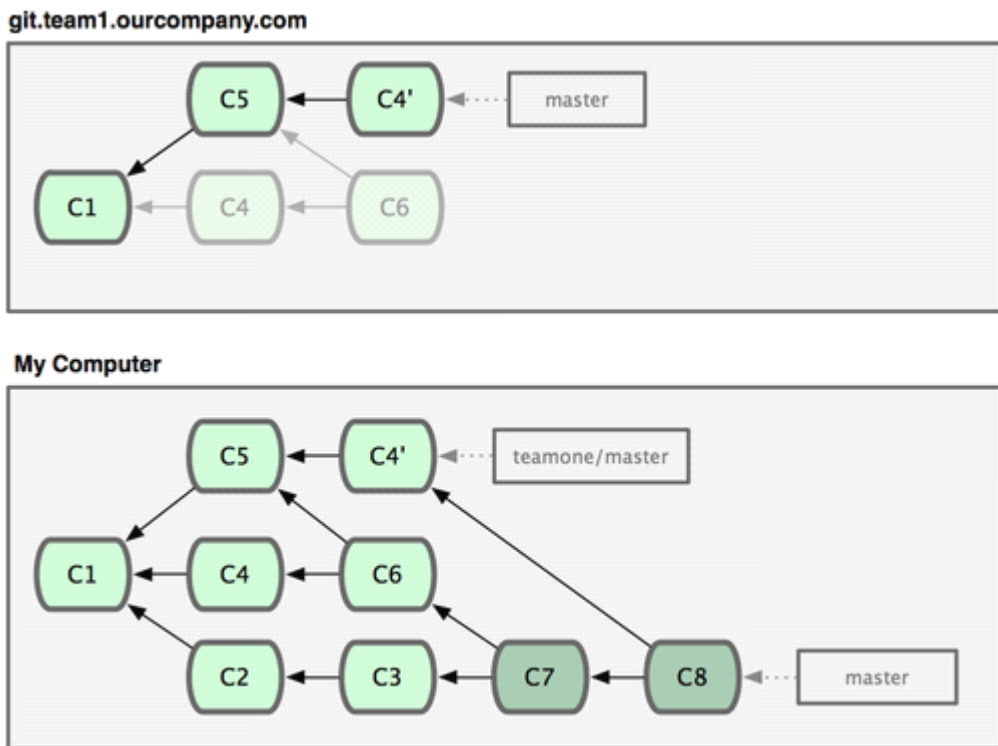


图 3-39. 你把相同的内容又合并了一遍，生成一个新的提交 C8。

C8 这一步的合并是迟早会发生的，因为只有这样才能和其他协作者提交的内容保持同步。而在 C8 之后，你的提交历史里就会同时包含 C4 和 C4'，两者有着不同的 SHA-1 校验值，如果用 `git log` 查看历史，会看到两个提交拥有相同的作者日期与说明，令人费解。而更糟的是，当你把这样的历史推送到服务器后，会再次把这些衍合后的提交引入到中央服务器，进一步困扰其他人（译注：这个例子中，出问题的责任方是那个发布了 C6 后又用衍合发布 C4' 的人，其他人会因此反馈双重历史到共享主干，从而混淆大家的视听。）。

如果把衍合当成一种在推送之前清理提交历史的手段，而且仅仅衍合那些尚未公开的提交对象，就没问题。如果衍合那些已经公开的提交对象，并且已经有人基于这些提交对象开展了后续开发工作的话，就会出现叫人沮丧的麻烦。

3.7 小结

读到这里，你应该已经学会了如何创建分支并切换到新分支，在不同分支间转换，合并本地分支，把分支推送到共享服务器上，使用共享分支与他人协作，以及在分享之前进行衍合。

服务器上的 Git

到目前为止，你应该已经学会了使用 **Git** 来完成日常工作。然而，如果想与他人合作，还需要一个远程的 **Git** 仓库。尽管技术上可以从个人的仓库里推送和拉取修改内容，但我们不鼓励这样做，因为一不留心就很容易弄混其他人的进度。另外，你也一定希望合作者们即使在自己不开机的时候也能从仓库获取数据 — 拥有一个更稳定的公共仓库十分有用。因此，更好的合作方式是建立一个大家都可以访问的共享仓库，从那里推送和拉取数据。我们将把这个仓库称为“**Git 服务器**”；代理一个 **Git** 仓库只需要花费很少的资源，几乎从不需要整个服务器来支持它的运行。

架设一台 **Git** 服务器并不难。第一步是选择与服务器通讯的协议。本章第一节将介绍可用的协议以及各自优缺点。下面一节将介绍一些针对各个协议典型的设置以及如何在服务器上实施。最后，如果你不介意在他人服务器上保存你的代码，又想免去自己架设和维护服务器的麻烦，倒可以试试我们介绍的几个仓库托管服务。

如果你对架设自己的服务器没兴趣，可以跳到本章最后一节去看看如何申请一个代码托管服务的账户然后继续下一章，我们会在那里讨论分布式源码控制环境的林林总总。

远程仓库通常只是一个_裸仓库（bare repository）_ — 即一个没有当前工作目录的仓库。因为该仓库只是一个合作媒介，所以不需要从硬盘上取出最新版本的快照；仓库里存放的仅仅是 **Git** 的数据。简单地说，裸仓库就是你工作目录中 `.git` 子目录内的内容。

4.1 协议

Git 可以使用四种主要的协议来传输数据：本地传输，**SSH** 协议，**Git** 协议和 **HTTP** 协议。下面分别介绍一下哪些情形应该使用（或避免使用）这些协议。

值得注意的是，除了 **HTTP** 协议外，其他所有协议都要求在服务器端安装并运行 **Git**。

本地协议

最基本的就是_本地协议（Local protocol）_，所谓的远程仓库在该协议中的表示，就是硬盘上的另一个目录。这常见于团队每一个成员都对一个共享的文件系统（例如 **NFS**）拥有访问权，或者比较少见的多人共用同一台电脑的情况。后面一种情况并不安全，因为所有代码仓库实例都储存在同一台电脑里，增加了灾难性数据损失的可能性。

如果你使用一个共享的文件系统，就可以在一个本地文件系统中克隆仓库，推送和获取。克隆的时候只需要将远程仓库的路径作为 **URL** 使用，比如下面这样：

```
$ git clone /opt/git/project.git
```

或者这样：

```
$ git clone file:///opt/git/project.git
```

如果在 URL 开头明确使用 `file://`，那么 Git 会以一种略微不同的方式运行。如果你只给出路径，Git 会尝试使用硬链接或直接复制它所需要的文件。如果使用了 `file://`，Git 会调用它平时通过网络来传输数据的工序，而这种方式效率相对较低。使用 `file://` 前缀的主要原因是当你需要一个不包含无关引用或对象的干净仓库副本的时候——一般指从其他版本控制系统导入的，或类似情形（参见第 9 章的维护任务）。我们这里仅仅使用普通路径，这样更快。

要添加一个本地仓库作为现有 Git 项目的远程仓库，可以这样做：

```
$ git remote add local_proj /opt/git/project.git
```

然后就可以像在网络上一向向这个远程仓库推送和获取数据了。

优点

基于文件仓库的优点在于它的简单，同时保留了现存文件的权限和网络访问权限。如果你的团队已经有一个全体共享的文件系统，建立仓库就十分容易了。你只需把一份裸仓库的副本放在大家都能访问的地方，然后像对其他共享目录一样设置读写权限就可以了。我们将在下一节“在服务器上部署 Git”中讨论如何导出一个裸仓库的副本。

这也是从别人工作目录中获取工作成果的快捷方法。假如你和你的同事在一个项目中合作，他们想让你检出一些东西的时候，运行类似 `git pull /home/john/project` 通常会比他们推送到服务器，而你再从服务器获取简单得多。

缺点

这种方法的缺点是，与基本的网络连接访问相比，难以控制从不同位置来的访问权限。如果你想从家里的笔记本电脑上推送，就要先挂载远程硬盘，这和基于网络连接的访问相比更加困难和缓慢。

另一个很重要的问题是该方法不一定就是最快的，尤其是对于共享挂载的文件系统。本地仓库只有在你对数据访问速度快的时候才快。在同一个服务器上，如果二者同时允许 Git 访问本地硬盘，通过 NFS 访问仓库通常会比 SSH 慢。

SSH 协议

Git 使用的传输协议中最常见的可能就是 SSH 了。这是因为大多数环境已经支持通过 SSH 对服务器的访问——即便还没有，架设起来也很容易。SSH 也是唯一一个同时支持读写操作的网络协议。另外两个网络协议（HTTP 和 Git）通常都是只读的，所以虽然二者对大多数人都可用，但执行写操作时还是需要 SSH。SSH 同时也是一个验证授权的网络协议；而因为其普遍性，一般架设和使用都很容易。

通过 SSH 克隆一个 Git 仓库，你可以像下面这样给出 ssh:// 的 URL：

```
$ git clone ssh://user@server:project.git
```

或者不指明某个协议 — 这时 Git 会默认使用 SSH：

```
$ git clone user@server:project.git
```

如果不指明用户，Git 会默认使用当前登录的用户名连接服务器。

优点

使用 SSH 的好处有很多。首先，如果你想拥有对网络仓库的写权限，基本上不可能不使用 SSH。其次，SSH 架设相对比较简单 — SSH 守护进程很常见，很多网络管理员都有一些使用经验，而且很多操作系统都自带了它或者相关的管理工具。再次，通过 SSH 进行访问是安全的 — 所有数据传输都是加密和授权的。最后，和 Git 及本地协议一样，SSH 也很高效，会在传输之前尽可能压缩数据。

缺点

SSH 的限制在于你 cannot 通过它实现仓库的匿名访问。即使仅为读取数据，人们也必须在能通过 SSH 访问主机的前提下才能访问仓库，这使得 SSH 不利于开源的项目。如果你仅仅在公司网络里使用，SSH 可能是你唯一需要使用的协议。如果想允许对项目的匿名只读访问，那么除了为自己推送而架设 SSH 协议之外，还需要支持其他协议以便他人访问读取。

Git 协议

接下来是 Git 协议。这是一个包含在 Git 软件包中的特殊守护进程；它会监听一个提供类似于 SSH 服务的特定端口（9418），而无需任何授权。打算支持 Git 协议的仓库，需要先创建 git-export-daemon-ok 文件 — 它是协议进程提供仓库服务的必要条件 — 但除此之外该服务没有什么安全措施。要么所有人都能克隆 Git 仓库，要么谁也不能。这也意味着该协议通常不能用来进行推送。你可以允许推送操作；然而由于没有授权机制，一旦允许该操作，网络上任何一个知道项目 URL 的人将都有推送权限。不用说，这是十分罕见的情况。

优点

Git 协议是现存最快的传输协议。如果你在提供一个有很大访问量的公共项目，或者一个不需要对读操作进行授权的庞大项目，架设一个 Git 守护进程来供应仓库是个不错的选择。它使用与 SSH 协议相同的数据传输机制，但省去了加密和授权的开销。

缺点

Git 协议消极的一面是缺少授权机制。用 Git 协议作为访问项目的唯一方法通常是不可取

的。一般的做法是，同时提供 **SSH** 接口，让几个开发者拥有推送（写）权限，其他人通过 **git://** 拥有只读权限。**Git** 协议可能也是最难架设的协议。它要求有单独的守护进程，需要定制 — 我们将在本章的“**Gitosis**”一节详细介绍它的架设 — 需要设定 **xinetd** 或类似的程序，而这些工作就没那么轻松了。该协议还要求防火墙开放 **9418** 端口，而企业级防火墙一般不允许对这个非标准端口的访问。大型企业级防火墙通常会封锁这个少见的端口。

HTTP/S 协议

最后还有 **HTTP** 协议。**HTTP** 或 **HTTPS** 协议的优美之处在于架设的简便性。基本上，只需要把 **Git** 的裸仓库文件放在 **HTTP** 的根目录下，配置一个特定的 **post-update** 挂钩（hook）就可以搞定（**Git** 挂钩的细节见第 7 章）。此后，每个能访问 **Git** 仓库所在服务器上 **web** 服务的人都可以进行克隆操作。下面的操作可以允许通过 **HTTP** 对仓库进行读取：

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

这样就可以了。**Git** 附带的 **post-update** 挂钩会默认运行合适的命令（**git update-server-info**）来确保通过 **HTTP** 的获取和克隆正常工作。这条命令在你用 **SSH** 向仓库推送内容时运行；之后，其他人就可以用下面的命令来克隆仓库：

```
$ git clone http://example.com/gitproject.git
```

在本例中，我们使用了 **Apache** 设定中常用的 **/var/www/htdocs** 路径，不过你可以使用任何静态 **web** 服务 — 把裸仓库放在它的目录里就行。**Git** 的数据是以最基本的静态文件的形式提供的（关于如何提供文件的详情见第 9 章）。

通过 **HTTP** 进行推送操作也是可能的，不过这种做法不太常见，并且牵扯到复杂的 **WebDAV** 设定。由于很少用到，本书将略过对该内容的讨论。如果对 **HTTP** 推送协议感兴趣，不妨打开这个地址看一下操作方法：

<http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>。通过 **HTTP** 推送的好处之一是你可以使用任何 **WebDAV** 服务器，不需要为 **Git** 设定特殊环境；所以如果主机提供商支持通过 **WebDAV** 更新网站内容，你也可以使用这项功能。

优点

使用 **HTTP** 协议的好处是易于架设。几条必要的命令就可以让全世界读取到仓库的内容。花费不过几分钟。**HTTP** 协议不会占用过多服务器资源。因为它一般只用到静态的 **HTTP** 服务提供所有数据，普通的 **Apache** 服务器平均每秒能支撑数千个文件的并发访问 — 哪

怕让一个小型服务器超载都很难。

你也可以通过 **HTTPS** 提供只读的仓库，这意味着你可以加密传输内容；你甚至可以要求客户端使用特定签名的 **SSL** 证书。一般情况下，如果到了这一步，使用 **SSH** 公共密钥可能是更简单的方案；不过也存在一些特殊情况，这时通过 **HTTPS** 使用带签名的 **SSL** 证书或者其他基于 **HTTP** 的只读连接授权方式是更好的解决方案。

HTTP 还有个额外的好处：**HTTP** 是一个如此常见的协议，以至于企业级防火墙通常都允许其端口的通信。

缺点

HTTP 协议的消极面在于，相对来说客户端效率更低。克隆或者下载仓库内容可能会花费更多时间，而且 **HTTP** 传输的体积和网络开销比其他任何一个协议都大。因为它没有按需供应的能力 — 传输过程中没有服务端的动态计算 — 因而 **HTTP** 协议经常会被称为_傻瓜 (dumb)_协议。更多 **HTTP** 协议和其他协议效率上的差异见第 9 。

4.2 在服务器上部署 Git

开始架设 **Git** 服务器前，需要先把现有仓库导出为裸仓库 — 即一个不包含当前工作目录的仓库。做法直截了当，克隆时用 **--bare** 选项即可。裸仓库的目录名一般以 **.git** 结尾，像这样：

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

该命令的输出或许会让人有些不解。其实 **clone** 操作基本上相当于 **git init** 加 **git fetch**，所以这里出现的其实是 **git init** 的输出，先由它建立一个空目录，而之后传输数据对象的操作并无任何输出，只是悄悄在幕后执行。现在 **my_project.git** 目录中已经有了一份 **Git** 目录数据的副本。

整体上的效果大致相当于：

```
$ cp -Rf my_project/.git my_project.git
```

但在配置文件中有若干小改动，不过对用户来讲，使用方式都一样，不会有什么影响。它仅取出 **Git** 仓库的必要原始数据，存放在该目录中，而不会另外创建工作目录。

把裸仓库移到服务器上

有了裸仓库的副本后，剩下的就是把它放到服务器上并设定相关协议。假设一个域名为 **git.example.com** 的服务器已经架设好，并可以通过 **SSH** 访问，我们打算把所有 **Git** 仓库储存在 **/opt/git** 目录下。只要把裸仓库复制过去：

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

现在，所有对该服务器有 **SSH** 访问权限，并可读取 `/opt/git` 目录的用户都可以用下面的命令克隆该项目：

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

如果某个 **SSH** 用户对 `/opt/git/my_project.git` 目录有写权限，那他就有推送权限。如果到该项目目录中运行 `git init` 命令，并加上 `--shared` 选项，那么 **Git** 会自动修改该仓库目录的组权限为可写（译注：实际上 `--shared` 可以指定其他行为，只是默认为将组权限改为可写并执行 `g+sx`，所以最后会得到 `rws`。）。

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

由此可见，根据现有的 **Git** 仓库创建一个裸仓库，然后把它放上你和同事都有 **SSH** 访问权的服务器是多么容易。现在已经可以开始在同一项目上密切合作了。

值得注意的是，这的确确是架设一个少数人具有连接权的 **Git** 服务的全部 — 只要在服务器上加入可以用 **SSH** 登录的帐号，然后把裸仓库放在大家都有读写权限的地方。一切都准备妥当，无需更多。

下面的几节中，你会了解如何扩展到更复杂的设定。这些内容包含如何避免为每一个用户建立一个账户，给仓库添加公共读取权限，架设网页界面，使用 **Gitosis** 工具等等。然而，只是和几个人在一个不公开的项目上合作的话，仅仅是一个 **SSH** 服务器和裸仓库就足够了，记住这点就可以了。

小型安装

如果设备较少或者你只想在小型开发团队里尝试 **Git**，那么一切都很简单。架设 **Git** 服务最复杂的地方在于账户管理。如果需要仓库对特定的用户可读，而给另一部分用户读写权限，那么访问和许可的安排就比较困难。

SSH 连接

如果已经有了一个所有开发成员都可以用 **SSH** 访问的服务器，架设第一个服务器将变得异常简单，几乎什么都不用做（正如上节中介绍的那样）。如果需要对仓库进行更复杂的访问控制，只要使用服务器操作系统的本地文件访问许可机制就行了。

如果需要团队里的每个人都对仓库有写权限，又不能给每个人在服务器上建立账户，那么提供 **SSH** 连接就是唯一的选择了。我们假设用来共享仓库的服务器已经安装了 **SSH** 服务，而且你通过它访问服务器。

有好几个办法可以让团队的每个人都有访问权。第一个办法是给每个人建立一个账户，直截了当但略过繁琐。反复运行 `adduser` 并给所有人设定临时密码可不是好玩的。

第二个办法是在主机上建立一个 `git` 账户，让每个需要写权限的人发送一个 `SSH` 公钥，然后将其加入 `git` 账户的 `~/.ssh/authorized_keys` 文件。这样一来，所有人都将通过 `git` 账户访问主机。这丝毫不会影响提交的数据 — 访问主机用的身份不会影响提交对象的提交者信息。

另一个办法是让 `SSH` 服务器通过某个 `LDAP` 服务，或者其他已经设定好的集中授权机制，来进行授权。只要每个人都能获得主机的 `shell` 访问权，任何可用的 `SSH` 授权机制都能达到相同效果。

4.3 生成 SSH 公钥

大多数 `Git` 服务器都会选择使用 `SSH` 公钥来进行授权。系统中的每个用户都必须提供一个公钥用于授权，没有的话就要生成一个。生成公钥的过程在所有操作系统上都差不多。首先先确认一下是否已经有一个公钥了。`SSH` 公钥默认储存在账户的主目录下的 `~/.ssh` 目录。进去看看：

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config           id_dsa.pub
```

关键是看有没有用 `something` 和 `something.pub` 来命名的一对文件，这个 `something` 通常就是 `id_dsa` 或 `id_rsa`。有 `.pub` 后缀的文件就是公钥，另一个文件则是密钥。假如没有这些文件，或者干脆连 `.ssh` 目录都没有，可以用 `ssh-keygen` 来创建。该程序在 `Linux/Mac` 系统上由 `SSH` 包提供，而在 `Windows` 上则包含在 `MSysGit` 包里：

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

它先要求你确认保存公钥的位置（`.ssh/id_rsa`），然后它会让你重复一个密码两次，如果不想在使用公钥的时候输入密码，可以留空。

现在，所有做过这一步的用户都得把它们的公钥给你或者 `Git` 服务器的管理员（假设 `SSH`

服务被设定为使用公钥机制)。他们只需要复制 `.pub` 文件的内容然后发邮件给管理员。公钥的样子大致如下:

```
$ cat ~/.ssh/id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpKDhrfHYl7SbrmTIpNLTGK9Tjom/BWDSU
GP1+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2sold0lQraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

关于在多个操作系统上设立相同 SSH 公钥的教程,可以查阅 [GitHub](http://github.com/guides/providing-your-ssh-key) 上有关 SSH 公钥的向导: <http://github.com/guides/providing-your-ssh-key>。

4.4 架设服务器

现在我们过一边服务器端架设 SSH 访问的流程。本例将使用 `authorized_keys` 方法来给用户授权。我们还将假定使用类似 `Ubuntu` 这样的标准 `Linux` 发行版。首先,创建一个名为 `'git'` 的用户,并为其创建一个 `.ssh` 目录。

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

接下来,把开发者的 SSH 公钥添加到这个用户的 `authorized_keys` 文件中。假设你通过电邮收到了几个公钥并存到了临时文件里。重复一下,公钥大致看起来是这个样子:

```
$ cat /tmp/id_rsa.john.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x41hJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NysnEAZuXz0jTTyAUfirtU3Z5E003C4oxOj6H0rfIFlkKI9MAQLMdpGWlGYEIgS9Ez
Sdfd8AcCiicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHuKEOmffjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

只要把它们逐个追加到 `authorized_keys` 文件尾部即可:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

现在可以用 **--bare** 选项运行 **git init** 来建立一个裸仓库，这会初始化一个不包含工作目录的仓库。

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

这时，**Join**，**Josie** 或者 **Jessica** 就可以把它加为远程仓库，推送一个分支，从而把第一个版本的项目文件上传到仓库里了。值得注意的是，每次添加一个新项目都需要通过 **shell** 登入主机并创建一个裸仓库目录。我们不妨以 **gitserver** 作为 **git** 用户及项目仓库所在的主机名。如果在网络内部运行该主机，并在 **DNS** 中设定 **gitserver** 指向该主机，那么以下这些命令都是可用的：

```
# 在 John 的电脑上
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

这样，其他人的克隆和推送也一样变得很简单：

```
$ git clone git@gitserver:/opt/git/project.git
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

用这个方法可以很快捷地为少数几个开发者架设一个可读写的 **Git** 服务。

作为一个额外的防范措施，你可以用 **Git** 自带的 **git-shell** 工具限制 **git** 用户的活动范围。只要把它设为 **git** 用户登入的 **shell**，那么该用户就无法使用普通的 **bash** 或者 **csch** 什么的 **shell** 程序。编辑 **/etc/passwd** 文件：

```
$ sudo vim /etc/passwd
```

在文件末尾，你应该能找到类似这样的行：

```
git:x:1000:1000::/home/git:/bin/sh
```

把 **bin/sh** 改为 **/usr/bin/git-shell**（或者用 **which git-shell** 查看它的实际安装路径）。该行修改后的样子如下：

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

现在 **git** 用户只能用 **SSH** 连接来推送和获取 **Git** 仓库，而不能直接使用主机 **shell**。尝试普通 **SSH** 登录的话，会看到下面这样的拒绝信息：

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

4.5 公共访问

匿名的读取权限该怎么实现呢？也许除了内部私有的项目之外，你还需要托管一些开源项目。或者因为要用一些自动化的服务器来进行编译，或者有一些经常变化的服务器群组，而又不想整天生成新的 **SSH** 密钥 — 总之，你需要简单的匿名读取权限。

或许对小型的配置来说最简单的办法就是运行一个静态 **web** 服务，把它的根目录设定为 **Git** 仓库所在的位置，然后开启本章第一节提到的 **post-update** 挂钩。这里继续使用之前的例子。假设仓库处于 **/opt/git** 目录，主机上运行着 **Apache** 服务。重申一下，任何 **web** 服务程序都可以达到相同效果；作为范例，我们将用一些基本的 **Apache** 设定来展示大体需要的步骤。

首先，开启挂钩：

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

如果用的是 **Git 1.6** 之前的版本，则可以省略 **mv** 命令 — **Git** 是从较晚的版本才开始在挂钩实例的结尾添加 **.sample** 后缀名的。

post-update 挂钩是做什么的呢？其内容大致如下：

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

意思是当通过 **SSH** 向服务器推送时，**Git** 将运行这个 **git-update-server-info** 命令来更新匿名 **HTTP** 访问获取数据时所需要的文件。

接下来，在 **Apache** 配置文件中添加一个 **VirtualHost** 条目，把文档根目录设为 **Git** 项目所在的根目录。这里我们假定 **DNS** 服务已经配置好，会把对 **gitserver** 的请求发送到这台主机：

```
ServerName git.gitserver DocumentRoot /opt/git Order allow, deny allow from  
all
```

另外，需要把 `/opt/git` 目录的 Unix 用户组设定为 `www-data`，这样 web 服务才可以读取仓库内容，因为运行 CGI 脚本的 Apache 实例进程默认就是以该用户的身份起来的：

```
$ chgrp -R www-data /opt/git
```

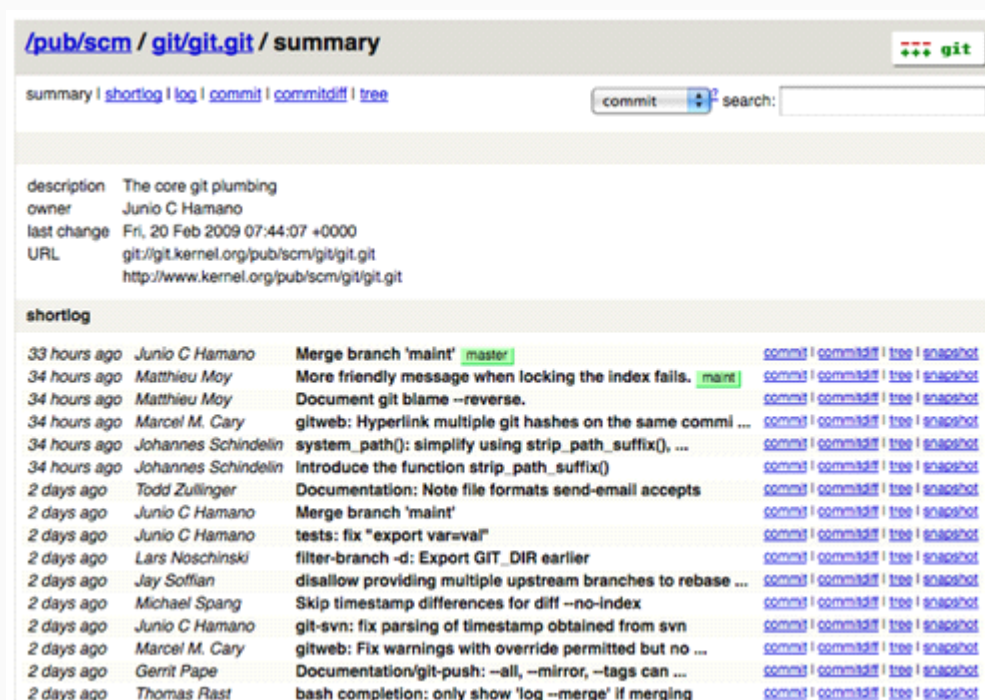
重启 Apache 之后，就可以通过项目的 URL 来克隆该目录下的仓库了。

```
$ git clone http://git.gitserver/project.git
```

这一招可以让你在几分钟内为相当数量的用户架设好基于 HTTP 的读取权限。另一个提供非授权访问的简单方法是开启一个 Git 守护进程，不过这将要求该进程作为后台进程常驻——接下来的这一节就要讨论这方面的细节。

4.6 GitWeb

现在我们的项目已经有了可读可写和只读的连接方式，不过如果能有一个简单的 web 界面访问就更好了。Git 自带一个叫做 GitWeb 的 CGI 脚本，运行效果可以到 <http://git.kernel.org> 这样的站点体验下（见图 4-1）。



The screenshot displays the GitWeb interface for the `/pub/scm/git/git.git` repository. The page has a header with the repository path and a 'git' logo. Below the header, there are navigation links: `summary`, `shortlog`, `log`, `commit`, `commitdiff`, and `tree`. A search bar is also present. The main content area shows the repository's description, owner, last change, and URL. Below this, a 'shortlog' section lists recent commits with their timestamps, authors, and descriptions. Each commit entry includes links for `commit`, `commitdiff`, `tree`, and `snapshot`.

Time	Author	Commit Message	Links
33 hours ago	Junio C Hamano	Merge branch 'maint' master	commit commitdiff tree snapshot
34 hours ago	Matthieu Moy	More friendly message when locking the index fails. maint	commit commitdiff tree snapshot
34 hours ago	Matthieu Moy	Document git blame --reverse.	commit commitdiff tree snapshot
34 hours ago	Marcel M. Cary	gitweb: Hyperlink multiple git hashes on the same commit ...	commit commitdiff tree snapshot
34 hours ago	Johannes Schindelin	system_path(): simplify using strip_path_suffix(), ...	commit commitdiff tree snapshot
34 hours ago	Johannes Schindelin	Introduce the function strip_path_suffix()	commit commitdiff tree snapshot
2 days ago	Todd Zullinger	Documentation: Note file formats send-email accepts	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	Merge branch 'maint'	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	tests: fix "export var=val"	commit commitdiff tree snapshot
2 days ago	Lars Noschinski	filter-branch -d: Export GIT_DIR earlier	commit commitdiff tree snapshot
2 days ago	Jay Soffian	disallow providing multiple upstream branches to rebase ...	commit commitdiff tree snapshot
2 days ago	Michael Spang	Skip timestamp differences for diff --no-index	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	git-svn: fix parsing of timestamp obtained from svn	commit commitdiff tree snapshot
2 days ago	Marcel M. Cary	gitweb: Fix warnings with override permitted but no ...	commit commitdiff tree snapshot
2 days ago	Gerrit Pape	Documentation/git-push: --all, --mirror, --tags can ...	commit commitdiff tree snapshot
2 days ago	Thomas Rast	bash completion: only show 'log --merge' if merging	commit commitdiff tree snapshot

Figure 4-1. 基于网页的 **GitWeb** 用户界面如果想看看自己项目的效果，不妨用 **Git** 自带的一个命令，可以使用类似 **lighttpd** 或 **webrick** 这样轻量级的服务器启动一个临时进程。如果是在 **Linux** 主机上，通常都预装了 **lighttpd**，可以到项目目录中键入 **git instaweb** 来启动。如果用的是 **Mac**，**Leopard** 预装了 **Ruby**，所以 **webrick** 应该是最好的选择。如果要用 **lighttpd** 以外的程序来启动 **git instaweb**，可以通过 **--httpd** 选项指定：

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

这会在 **1234** 端口开启一个 **HTTPD** 服务，随之在浏览器中显示该页，十分简单。关闭服务时，只需在原来的命令后面加上 **--stop** 选项就可以了：

```
$ git instaweb --httpd=webrick --stop
```

如果需要为团队或者某个开源项目长期运行 **GitWeb**，那么 **CGI** 脚本就要由正常的网页服务来运行。一些 **Linux** 发行版可以通过 **apt** 或 **yum** 安装一个叫做 **gitweb** 的软件包，不妨首先尝试一下。我们将快速介绍一下手动安装 **GitWeb** 的流程。首先，你需要 **Git** 的源码，其中带有 **GitWeb**，并能生成定制的 **CGI** 脚本：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
    prefix=/usr gitweb/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

注意，通过指定 **GITWEB_PROJECTROOT** 变量告诉编译命令 **Git** 仓库的位置。然后，设置 **Apache** 以 **CGI** 方式运行该脚本，添加一个 **VirtualHost** 配置：

```
ServerName gitserver DocumentRoot /var/www/gitweb Options ExecCGI
+FollowSymLinks +SymLinksIfOwnerMatch AllowOverride All order allow,deny Allow
from all AddHandler cgi-script cgi DirectoryIndex gitweb.cgi
```

不难想象，**GitWeb** 可以使用任何兼容 **CGI** 的网页服务来运行；如果偏向使用其他 **web** 服务器，配置也不会很麻烦。现在，通过 **http://gitserver** 就可以在线访问仓库了，在 **http://git.server** 上还可以通过 **HTTP** 克隆和获取仓库的内容。

4.7 Gitosis

把所有用户的公钥保存在 `authorized_keys` 文件的做法，只能凑和一阵子，当用户数量达到几百人的规模时，管理起来就会十分痛苦。每次改删用户都必须登录服务器不去说，这种做法还缺少必要的权限管理 — 每个人都对所有项目拥有完整的读写权限。

幸好我们还可以选择应用广泛的 **Gitosis** 项目。简单地说，**Gitosis** 就是一套用来管理 `authorized_keys` 文件和实现简单连接限制的脚本。有趣的是，用来添加用户和设定权限的并非通过网页程序，而只是管理一个特殊的 **Git** 仓库。你只需要在这个特殊仓库内做好相应的设定，然后推送到服务器上，**Gitosis** 就会随之改变运行策略，听起来就很酷，对吧？

Gitosis 的安装算不上傻瓜化，但也不算太难。用 **Linux** 服务器架设起来最简单 — 以下例子中，我们使用装有 **Ubuntu 8.10** 系统的服务器。

Gitosis 的工作依赖于某些 **Python** 工具，所以首先要安装 **Python** 的 `setuptools` 包，在 **Ubuntu** 上称为 `python-setuptools`：

```
$ apt-get install python-setuptools
```

接下来，从 **Gitosis** 项目主页克隆并安装：

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

这会安装几个供 **Gitosis** 使用的工具。默认 **Gitosis** 会把 `/home/git` 作为存储所有 **Git** 仓库的根目录，这没什么不好，不过我们之前已经把项目仓库都放在 `/opt/git` 里面了，所以为了方便起见，我们可以做一个符号连接，直接划转过去，而不必重新配置：

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis 将会帮我们管理用户公钥，所以先把当前控制文件改名备份，以便稍后重新添加，准备好让 **Gitosis** 自动管理 `authorized_keys` 文件：

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

接下来，如果之前把 **git** 用户的登录 `shell` 改为 `git-shell` 命令的话，先恢复 ‘**git**’ 用户的登录 `shell`。改过之后，大家仍然无法通过该帐号登录（译注：因为 `authorized_keys` 文件已经没有了。），不过不用担心，这会交给 **Gitosis** 来实现。所以现在先打开 `/etc/passwd` 文件，把这行：

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

改回：

```
git:x:1000:1000::/home/git:/bin/sh
```

好了，现在可以初始化 **Gitosis** 了。你可以用自己的公钥执行 **gitosis-init** 命令，要是公钥不在服务器上，先临时复制一份：

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

这样该公钥的拥有者就能修改用于配置 **Gitosis** 的那个特殊 **Git** 仓库了。接下来，需要手工对该仓库中的 **post-update** 脚本加上可执行权限：

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

基本上就算是好了。如果设定过程没出什么差错，现在可以试一下用初始化 **Gitosis** 的公钥的拥有者身份 **SSH** 登录服务器，应该会看到类似下面这样：

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

说明 **Gitosis** 认出了该用户的身份，但由于没有运行任何 **Git** 命令，所以它切断了连接。那么，现在运行一个实际的 **Git** 命令 — 克隆 **Gitosis** 的控制仓库：

```
# 在你本地计算机上
$ git clone git@gitserver:gitosis-admin.git
```

这会得到一个名为 **gitosis-admin** 的工作目录，主要由两部分组成：

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

gitosis.conf 文件是用来设置用户、仓库和权限的控制文件。**keydir** 目录则是保存所有具有访问权限用户公钥的地方 — 每人一个。在 **keydir** 里的文件名（比如上面的 **scott.pub**）应该跟你的不一样 — **Gitosis** 会自动从使用 **gitosis-init** 脚本导入的公钥尾部的描述中获取该名字。

看一下 **gitosis.conf** 文件的内容，它应该只包含与刚刚克隆的 **gitosis-admin** 相关的信息：

```
$ cat gitosis.conf
[gitosis]
```

```
[group gitosis-admin]
writable = gitosis-admin
members = scott
```

它显示用户 **scott** — 初始化 **Gitosis** 公钥的拥有者 — 是唯一能管理 **gitosis-admin** 项目的人。

现在我们来添加一个新项目。为此我们要建立一个名为 **mobile** 的新段落，在其中罗列手机开发团队的开发者，以及他们拥有写权限的项目。由于 ‘**scott**’ 是系统中的唯一用户，我们把他设为唯一用户，并允许他读写名为 **iphone_project** 的新项目：

```
[group mobile]
writable = iphone_project
members = scott
```

修改完之后，提交 **gitosis-admin** 里的改动，并推送到服务器使其生效：

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"

1 files changed, 4 insertions(+), 0 deletions(-)

$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
fb27aec..8962da8 master -> master
```

在新工程 **iphone_project** 里首次推送数据到服务器前，得先设定该服务器地址为远程仓库。但你不用事先到服务器上手工创建该项目的裸仓库— **Gitosis** 会在第一次遇到推送时自动创建：

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
* [new branch]      master -> master
```

请注意，这里不用指明完整路径（实际上，如果加上反而没用），只需要一个冒号加项目名

字即可 — **Gitis** 会自动帮你映射到实际位置。

要和朋友们在一个项目上协同工作，就得重新添加他们的公钥。不过这次不用在服务器上一个一个手工添加到`~/.ssh/authorized_keys` 文件末端，而只需管理 `keydir` 目录中的公钥文件。文件的命名将决定在 `gitis.conf` 中对用户的标识。现在我们为 **John**，**Josie** 和 **Jessica** 添加公钥：

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

然后把他们都加进 ‘mobile’ 团队，让他们对 `iphone_project` 具有读写权限：

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

如果你提交并推送这个修改，四个用户将同时具有该项目的读写权限。

Gitis 也具有简单的访问控制功能。如果想让 **John** 只有读权限，可以这样做：

```
[group mobile]
writable = iphone_project
members = scott josie jessica
```

```
[group mobile_ro]
readonly = iphone_project
members = john
```

现在 **John** 可以克隆和获取更新，但 **Gitis** 不会允许他向项目推送任何内容。像这样的组可以随意创建，多少不限，每个都可以包含若干不同的用户和项目。甚至还可以指定某个组为成员之一（在组名前加上`@` 前缀），自动继承该组的成员：

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
writable = iphone_project
members = @mobile_committers

[group mobile_2]
writable = another_iphone_project
```

```
members = @mobile_committers john
```

如果遇到意外问题，试试看把 `loglevel=DEBUG` 加到 `[gitosis]` 的段落（译注：把日志设置为调试级别，记录更详细的运行信息。）。如果一不小心搞错了配置，失去了推送权限，也可以手工修改服务器上的 `/home/git/.gitosis.conf` 文件 — `Gitosis` 实际是从该文件读取信息的。它在得到推送数据时，会把新的 `gitosis.conf` 存到该路径上。所以如果你手工编辑该文件的话，它会一直保持到下次向 `gitosis-admin` 推送新版本的配置内容为止。

4.8 Gitolite

Note: the latest copy of this section of the ProGit book is always available within the [gitolite documentation](#). The author would also like to humbly state that, while this section is accurate, and can (and often has) been used to install gitolite without reading any other documentation, it is of necessity not complete, and cannot completely replace the enormous amount of documentation that gitolite comes with.

Git has started to become very popular in corporate environments, which tend to have some additional requirements in terms of access control. Gitolite was originally created to help with those requirements, but it turns out that it's equally useful in the open source world: the Fedora Project controls access to their package management repositories (over 10,000 of them!) using gitolite, and this is probably the largest gitolite installation anywhere too.

Gitolite allows you to specify permissions not just by repository, but also by branch or tag names within each repository. That is, you can specify that certain people (or groups of people) can only push certain “refs” (branches or tags) but not others.

Installing

Installing Gitolite is very easy, even if you don't read the extensive documentation that comes with it. You need an account on a Unix server of some kind; various Linux flavours, and Solaris 10, have been tested. You do not need root access, assuming git, perl, and an openssh compatible ssh server are already installed. In the examples below, we will use the gitolite account on a host called gitserver.

Gitolite is somewhat unusual as far as “server” software goes – access is via ssh, and so every userid on the server is a potential “gitolite host”. As a result, there is a notion of “installing” the software itself, and then “setting up” a user as a “gitolite host”.

Gitolite has 4 methods of installation. People using Fedora or Debian systems can obtain an RPM or a DEB and install that. People with root access can install it manually. In these two methods, any user on the system can then become a “gitolite host”.

People without root access can install it within their own userids. And finally, gitolite can be installed by running a script on the workstation, from a bash shell. (Even the bash that comes with msysgit will do, in case you’re wondering.)

We will describe this last method in this article; for the other methods please see the documentation.

You start by obtaining public key based access to your server, so that you can log in from your workstation to the server without getting a password prompt. The following method works on Linux; for other workstation OSs you may have to do this manually. We assume you already had a key pair generated using ssh-keygen.

```
$ ssh-copy-id -i ~/.ssh/id_rsa gitolite@gitserver
```

This will ask you for the password to the gitolite account, and then set up public key access. This is **essential** for the install script, so check to make sure you can run a command without getting a password prompt:

```
$ ssh gitolite@gitserver pwd
/home/gitolite
```

Next, you clone Gitolite from the project’s main site and run the “easy install” script (the third argument is your name as you would like it to appear in the resulting gitolite-admin repository):

```
$ git clone git://github.com/sitaramc/gitolite
$ cd gitolite/src
$ ./gl-easy-install -q gitolite gitserver sitaram
```

And you’re done! Gitolite has now been installed on the server, and you now have a brand new repository called gitolite-admin in the home directory of your workstation. You administer your gitolite setup by making changes to this repository and pushing.

That last command does produce a fair amount of output, which might be interesting to read. Also, the first time you run this, a new keypair is created; you will have to choose a passphrase or hit enter for none. Why a second keypair is needed, and how it is used, is explained in the “ssh troubleshooting” document that comes with Gitolite. (Hey the documentation has to be good for something!)

Repos named gitolite-admin and testing are created on the server by default. If you wish to clone either of these locally (from an account that has SSH console access to the gitolite account via authorized_keys), type:

```
$ git clone gitolite:gitolite-admin
$ git clone gitolite:testing
```

To clone these same repos from any other account:

```
$ git clone gitolite@servername:gitolite-admin
$ git clone gitolite@servername:testing
```

Customising the Install

While the default, quick, install works for most people, there are some ways to customise the install if you need to. If you omit the -q argument, you get a “verbose” mode install – detailed information on what the install is doing at each step. The verbose mode also allows you to change certain server-side parameters, such as the location of the actual repositories, by editing an “rc” file that the server uses. This “rc” file is liberally commented so you should be able to make any changes you need quite easily, save it, and continue. This file also contains various settings that you can change to enable or disable some of gitolite’s advanced features.

Config File and Access Control Rules

Once the install is done, you switch to the gitolite-admin repository (placed in your HOME directory) and poke around to see what you got:

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/sitaram.pub
$ cat conf/gitolite.conf
#gitolite conf
# please see conf/example.conf for details on syntax and features

repo gitolite-admin
    RW+                = sitaram

repo testing
```

```
RW+                = @all
```

Notice that “sitaram” (the last argument in the `gl-easy-install` command you gave earlier) has read-write permissions on the `gitolite-admin` repository as well as a public key file of the same name.

The config file syntax for gitolite is liberally documented in `conf/example.conf`, so we’ll only mention some highlights here.

You can group users or repos for convenience. The group names are just like macros; when defining them, it doesn’t even matter whether they are projects or users; that distinction is only made when you use the “macro”.

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admins         = scott      # Adams, not Chacon, sorry :)
@interns        = ashok      # get the spelling right, Scott!
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

You can control permissions at the “ref” level. In the following example, interns can only push the “int” branch. Engineers can push any branch whose name starts with “eng-“, and tags that start with “rc” followed by a digit. And the admins can do anything (including rewind) to any ref.

```
repo @oss_repos

  RW int$          = @interns
  RW eng-          = @engineers
  RW refs/tags/rc[0-9] = @engineers
  RW+              = @admins
```

The expression after the `RW` or `RW+` is a regular expression (regex) that the refname (ref) being pushed is matched against. So we call it a “refex”! Of course, a refex can be far more powerful than shown here, so don’t overdo it if you’re not comfortable with perl regexes.

Also, as you probably guessed, Gitolite prefixes `refs/heads/` as a syntactic convenience if the refex does not begin with `refs/`.

An important feature of the config file’s syntax is that all the rules for a repository need not be in one place. You can keep all the common stuff together, like the rules for `@oss_repos`

shown above, then add specific rules for specific cases later on, like so:

```
repo gitolite
    RW+                = sitaram
```

That rule will just get added to the ruleset for the gitolite repository.

At this point you might be wondering how the access control rules are actually applied, so let's go over that briefly.

There are two levels of access control in gitolite. The first is at the repository level; if you have read (or write) access to any ref in the repository, then you have read (or write) access to the repository.

The second level, applicable only to "write" access, is by branch or tag within a repository. The username, the access being attempted (W or +), and the refname being updated are known. The access rules are checked in order of appearance in the config file, looking for a match for this combination (but remember that the refname is regex-matched, not merely string-matched). If a match is found, the push succeeds. A fallthrough results in access being denied.

Advanced Access Control with "deny" rules

So far, we've only seen permissions to be one of R, RW, or RW+. However, gitolite allows another permission: -, standing for "deny". This gives you a lot more power, at the expense of some complexity, because now fallthrough is not the only way for access to be denied, so the order of the rules now matters!

Let us say, in the situation above, we want engineers to be able to rewind any branch except master and integ. Here's how to do that:

```
RW master integ      = @engineers
-   master integ      = @engineers
RW+                   = @engineers
```

Again, you simply follow the rules top down until you hit a match for your access mode, or a deny. Non-rewind push to master or integ is allowed by the first rule. A rewind push to those refs does not match the first rule, drops down to the second, and is therefore denied. Any push (rewind or non-rewind) to refs other than master or integ won't match the first two rules anyway, and the third rule allows it.

Restricting pushes by files changed

In addition to restricting what branches a user can push changes to, you can also restrict

what files they are allowed to touch. For example, perhaps the Makefile (or some other program) is really not supposed to be changed by just anyone, because a lot of things depend on it or would break if the changes are not done just right. You can tell gitolite:

```
repo foo

RW          = @junior_devs @senior_devs

RW NAME/    = @senior_devs
- NAME/Makefile = @junior_devs
RW NAME/    = @junior_devs
```

This powerful feature is documented in `conf/example.conf`.

Personal Branches

Gitolite also has a feature called “personal branches” (or rather, “personal branch namespace”) that can be very useful in a corporate environment.

A lot of code exchange in the git world happens by “please pull” requests. In a corporate environment, however, unauthenticated access is a no-no, and a developer workstation cannot do authentication, so you have to push to the central server and ask someone to pull from there.

This would normally cause the same branch name clutter as in a centralised VCS, plus setting up permissions for this becomes a chore for the admin.

Gitolite lets you define a “personal” or “scratch” namespace prefix for each developer (for example, `refs/personal/` /*); see the “personal branches” section in `doc/3-faq-tips-etc.mkd` for details.

“Wildcard” repositories

Gitolite allows you to specify repositories with wildcards (actually perl regexes), like, for example `assignments/s[0-9][0-9]/a[0-9][0-9]`, to pick a random example. This is a very powerful feature, which has to be enabled by setting `$GL_WILDREPOS = 1`; in the `rc` file. It allows you to assign a new permission mode (“C”) which allows users to create repositories based on such wild cards, automatically assigns ownership to the specific user who created it, allows him/her to hand out R and RW permissions to other users to collaborate, etc. This feature is documented in `doc/4-wildcard-repositories.mkd`.

Other Features

We’ll round off this discussion with a sampling of other features, all of which, and many

more, are described in great detail in the “faqs, tips, etc” and other documents.

Logging: Gitolite logs all successful accesses. If you were somewhat relaxed about giving people rewind permissions (RW+) and some kid blew away “master”, the log file is a life saver, in terms of easily and quickly finding the SHA that got hosed.

Git outside normal PATH: One extremely useful convenience feature in gitolite is support for git installed outside the normal \$PATH (this is more common than you think; some corporate environments or even some hosting providers refuse to install things system-wide and you end up putting them in your own directories). Normally, you are forced to make the client-side git aware of this non-standard location of the git binaries in some way. With gitolite, just choose a verbose install and set \$GIT_PATH in the “rc” files. No client-side changes are required after that :-)

Access rights reporting: Another convenient feature is what happens when you try and just ssh to the server. Gitolite shows you what repos you have access to, and what that access may be. Here’s an example:

```
hello sitaram, the gitolite version here is v1.5.4-19-ga3397d4
the gitolite config gives you the following access:

R      anu-wsd
R      entrans
R W    git-notes
R W    gitolite
R W    gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

Delegation: For really large installations, you can delegate responsibility for groups of repositories to various people and have them manage those pieces independently. This reduces the load on the main admin, and makes him less of a bottleneck. This feature has its own documentation file in the doc/ directory.

Gitweb support: Gitolite supports gitweb in several ways. You can specify which repos are visible via gitweb. You can set the “owner” and “description” for gitweb from the gitolite config file. Gitweb has a mechanism for you to implement access control based on HTTP authentication, so you can make it use the “compiled” config file that gitolite produces, which means the same access control rules (for read access) apply for gitweb and gitolite.

Mirroring: Gitolite can help you maintain multiple mirrors, and switch between them easily

if the primary server goes down.

4.9 Git 守护进程

对于提供公共的，非授权的只读访问，我们可以抛弃 HTTP 协议，改用 Git 自己的协议，这主要是出于性能和速度的考虑。Git 协议远比 HTTP 协议高效，因而访问速度也快，所以它能节省很多用户的时间。

重申一下，这一点只适用于非授权的只读访问。如果建在防火墙之外的服务器上，那么它所提供的服务应该只是那些公开的只读项目。如果是在防火墙之内的服务器上，可用于支撑大量参与人员或自动系统（用于持续集成或编译的主机）只读访问的项目，这样可以省去逐一配置 SSH 公钥的麻烦。

但不管哪种情形，Git 协议的配置设定都很简单。基本上，只要以守护进程的形式运行该命令即可：

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

这里的 `--reuseaddr` 选项表示在重启服务前，不等之前的连接超时就立即重启。而 `--base-path` 选项则允许克隆项目时不必给出完整路径。最后面的路径告诉 Git 守护进程允许开放给用户访问的仓库目录。假如有防火墙，则需要为该主机的 9418 端口设置为允许通信。

以守护进程的形式运行该进程的方法有很多，但主要还得看用的是什么操作系统。在 Ubuntu 主机上，可以用 Upstart 脚本达成。编辑该文件：

```
/etc/event.d/local-git-daemon
```

加入以下内容：

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

出于安全考虑，强烈建议用一个对仓库只有读取权限的用户身份来运行该进程 — 只需要简单地新建一个名为 `git-ro` 的用户（译注：新建用户默认对仓库文件不具备写权限，但这取决于仓库目录的权限设定。务必确认 `git-ro` 对仓库只能读不能写。），并用它的身份来启动进

程。这里为了简化，后面我们还是用之前运行 **Gitosis** 的用户 `'git'`。

这样一来，当你重启计算机时，**Git** 进程也会自动启动。要是进程意外退出或者被杀掉，也会自行重启。在设置完成后，不重启计算机就启动该守护进程，可以运行：

```
initctl start local-git-daemon
```

而在其他操作系统上，可以用 **xinetd**，或者 **sysvinit** 系统的脚本，或者其他类似的脚本——只要能让那个命令变为守护进程并可监控。

接下来，我们必须告诉 **Gitosis** 哪些仓库允许通过 **Git** 协议进行匿名只读访问。如果每个仓库都设有各自的段落，可以分别指定是否允许 **Git** 进程开放给用户匿名读取。比如允许通过 **Git** 协议访问 **iphone_project**，可以把下面两行加到 **gitosis.conf** 文件的末尾：

```
[repo iphone_project]
daemon = yes
```

在提交和推送完成后，运行中的 **Git** 守护进程就会响应来自 **9418** 端口对该项目的访问请求。

如果不考虑 **Gitosis**，单单起了 **Git** 守护进程的话，就必须到每一个允许匿名只读访问的仓库目录内，创建一个特殊名称的空文件作为标志：

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

该文件的存在，表明允许 **Git** 守护进程开放对该项目的匿名只读访问。

Gitosis 还能设定哪些项目允许放在 **GitWeb** 上显示。先打开 **GitWeb** 的配置文件 **/etc/gitweb.conf**，添加以下四行：

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

接下来，只要配置各个项目在 **Gitosis** 中的 **gitweb** 参数，便能达成是否允许 **GitWeb** 用户浏览该项目。比如，要让 **iphone_project** 项目在 **GitWeb** 里出现，把 **repo** 的设定改成下面的样子：

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

在提交并推送过之后，**GitWeb** 就会自动开始显示 **iphone_project** 项目的细节和历史。

4.10 Git 托管服务

如果不想经历自己架设 Git 服务器的麻烦，网络上有几个专业的仓库托管服务可供选择。这样做有几大优点：托管账户的建立通常比较省时，方便项目的启动，而且不涉及服务器的维护和监控。即使内部创建并运行着自己的服务器，同时为开源项目提供一个公共托管站点还是有好处的——让开源社区更方便地找到该项目，并给予帮助。

目前，可供选择的托管服务数量繁多，各有利弊。在 Git 官方 wiki 上的 Githosting 页面有一个最新的托管服务列表：

<http://git.or.cz/gitwiki/GitHosting>

由于本书无法全部一一介绍，而本人（译注：指本书作者 Scott Chacon。）刚好在其中一家公司工作，所以接下来我们将会介绍如何在 GitHub 上建立新账户并启动项目。至于其他托管服务大体也是这么一个过程，基本的想法都是差不多的。

GitHub 是目前为止最大的开源 Git 托管服务，并且还是少数同时提供公共代码和私有代码托管服务的站点之一，所以你可以在上面同时保存开源和商业代码。事实上，本书就是放在 GitHub 上合作编著的。（译注：本书的翻译也是放在 GitHub 上广泛协作的。）

GitHub

GitHub 和大多数的代码托管站点在处理项目命名空间的方式上略有不同。GitHub 的设计更侧重于用户，而不是完全基于项目。也就是说，如果我在 GitHub 上托管一个名为 grit 的项目的话，它的地址不会是 `github.com/grit`，而是按在用户底下 `github.com/shacon/grit`（译注：本书作者 Scott Chacon 在 GitHub 上的用户名是 shacon.）。不存在所谓某个项目的官方版本，所以假如第一作者放弃了某个项目，它可以无缝转移到其它用户的名下。

GitHub 同时也是一个向使用私有仓库的用户收取费用的商业公司，但任何人都可以方便快捷地申请到一个免费账户，并在上面托管数量无限的开源项目。接下来我们快速介绍一下 GitHub 的基本使用。

建立新账户

首先注册一个免费账户。访问 Pricing and Signup 页面 <http://github.com/plans> 并点击 Free account 里的 Sign Up 按钮（见图 4-2），进入注册页面。

分布式 Git

为了便于项目中的所有开发者分享代码，我们准备好了一台服务器存放远程 Git 仓库。经过前面几章的学习，我们已经学会了一些基本的本地工作流程中所需用到的命令。接下来，我们要学习下如何利用 Git 来组织和完成分布式工作流程。

特别是，当作为项目贡献者时，我们该怎么做才能方便维护者采纳更新；或者作为项目维护者时，又该怎样有效管理大量贡献者的提交。

5.1 分布式工作流程

同传统的集中式版本控制系统（CVCS）不同，开发者之间的协作方式因着 Git 的分布式特性而变得更为灵活多样。在集中式系统上，每个开发者就像是连接在集线器上的节点，彼此的工作方式大体相像。而在 Git 网络中，每个开发者同时扮演着节点和集线器的角色，这就是说，每一个开发者都可以将自己的代码贡献到另外一个开发者的仓库中，或者建立自己的公共仓库，让其他开发者基于自己的工作开始，为自己的仓库贡献代码。于是，Git 的分布式协作便可以衍生出种种不同的工作流程，我会在接下来的章节介绍几种常见的应用方式，并分别讨论各自的优缺点。你可以选择其中的一种，或者结合起来，应用到你自己的项目中。

集中式工作流

通常，集中式工作流程使用的都是单点协作模型。一个存放代码仓库的中心服务器，可以接受所有开发者提交的代码。所有的开发者都是普通的节点，作为中心集线器的消费者，平时的工作就是和中心仓库同步数据（见图 5-1）。

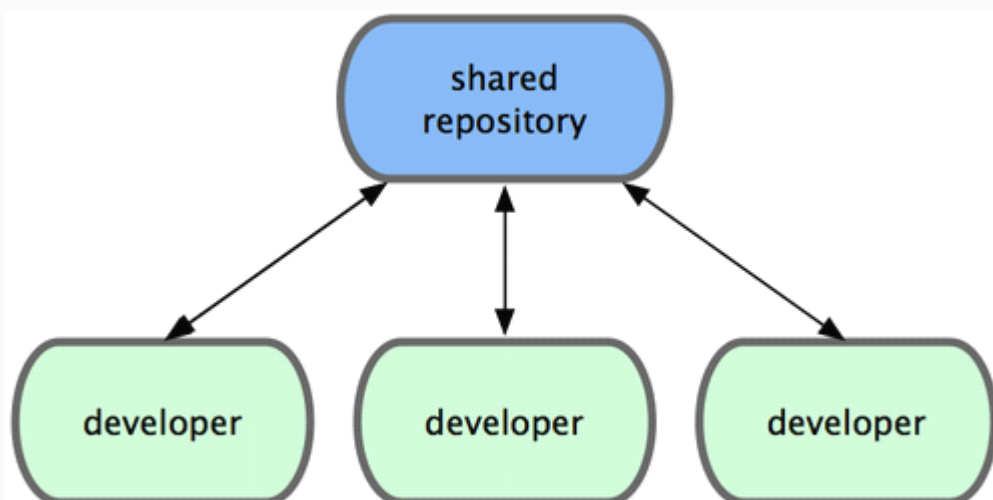


图 5-1. 集中式工作流如果两个开发者从中心仓库克隆代码下来，同时作了一些修订，那么只有第一个开发者可以顺利地把数据推送到共享服务器。第二个开发者在提交他的修订之前，必须先下载合并服务器上的数据，解决冲突之后才能推送数据到共享服务器上。在 Git 中这么用也决无问题，这就好比是在用 Subversion（或其他 CVCS）一样，可以很好地工作。

如果你的团队不是很大，或者大家都已经习惯了使用集中式工作流程，完全可以采用这种简单的模式。只需要配置好一台中心服务器，并给每个人推送数据的权限，就可以开展工作了。但如果提交代码时有冲突，Git 根本就不会让用户覆盖他人代码，它直接驳回第二个人的提交操作。这就等于告诉提交者，你所作的修订无法通过快进（fast-forward）来合并，你必须先拉取最新数据下来，手工解决冲突合并后，才能继续推送新的提交。绝大多数人都熟悉和了解这种模式的工作方式，所以使用也非常广泛。

集成管理员工作流

由于 Git 允许使用多个远程仓库，开发者便可以建立自己的公共仓库，往里面写数据并共享给他人，而同时又可以从中提取他们的更新过来。这种情形通常都会有一个代表着官方发布的项目仓库（blessed repository），开发者们由此仓库克隆出一个自己的公共仓库（developer public），然后将自己的提交推送上去，请求官方仓库的维护者拉取更新合并到主项目。维护者在自己的本地也有个克隆仓库（integration manager），他可以将你的公共仓库作为远程仓库添加进来，经过测试无误后合并到主干分支，然后再推送到官方仓库。工作流程看起来就像图 5-2 所示：

- 1 项目维护者可以推送数据到公共仓库 blessed repository。
2. 贡献者克隆此仓库，修订或编写新代码。
- 3 维护者在自己本地的 integration manger 仓库中，将贡献者的仓库加为远程仓库，合并更新并做测试。
- 4 维护者将合并后的更新推送到主仓库 blessed repository。

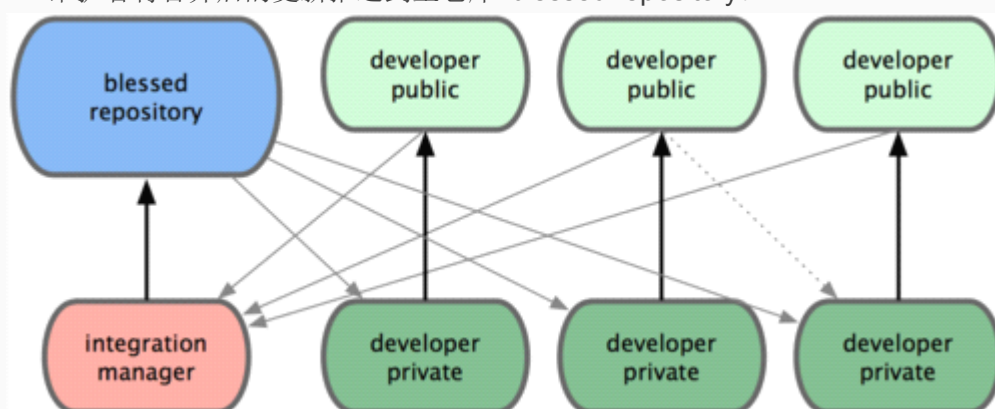


图 5-2. 集成管理员工作流在 GitHub 网站上使用得最多的就是这种工作流。人们可以复制（fork 亦即克隆）某个项目到自己的列表中，成为自己的公共仓库。随后将自己的更新提交到这个仓库，所有人都可以看到你的每次更新。这么做最主要的优点在于，你可以按照自己的节奏继续工作，而不必等待维护者处理你提交的更新；而维护者也可以按照自己的节奏，任何时候都可以过来处理接纳你的贡献。

司令官与副官工作流

这其实是上一种工作流的变体。一般超大型的项目才会用到这样的工作方式，像是拥有数百协作开发者的 Linux 内核项目就是如此。各个集成管理员分别负责集成项目中的特定部分，所以称为副官（lieutenant）。而所有这些集成管理员头上还有一位负责统筹的总集成管理员，称为司令官（dictator）。司令官维护的仓库用于提供所有协作者拉取最新集成的项目代码。整个流程看起来如图 5-3 所示：

- 5 一般的开发者在自己的特性分支上工作，并不定期地根据主干分支（dictator 上的 master）衍合。
- 6 副官（lieutenant）将普通开发者的特性分支合并到自己的 master 分支中。
- 7 司令官（dictator）将所有副官的 master 分支并入自己的 master 分支。
- 8 司令官 dictator 将集成后的 master 分支推送到共享仓库 blessed repository 中，以便所有其他开发者以此为基础进行衍合。

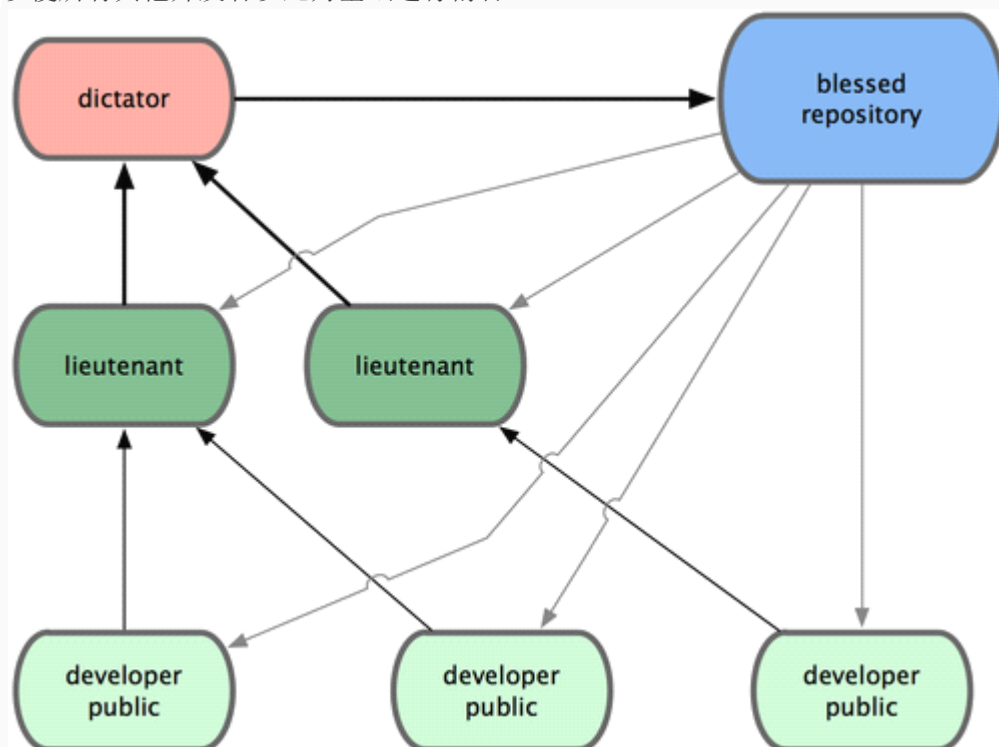


图 5-3. 司令官与副官 workflow 这种工作流程并不常用，只有当项目极为庞杂，或者需要多级别管理时，才会体现出优势。利用这种方式，项目总负责人（即司令官）可以把大量分散的集成工作委托给不同的小组负责人分别处理，最后再统筹起来，如此各人的职责清晰明确，也不易出错（译注：此乃分而治之）。

以上介绍的是常见的分布式系统可以应用的工作流程，当然不止于 **Git**。在实际的开发工作中，你可能会遇到各种为了满足特定需求而有所变化的工作方式。我想现在你应该已经清楚，接下来自己需要用哪种方式开展工作了。下节我还会再举些例子，看看各式 workflow 中的每个角色具体应该如何操作。

5.2 为项目作贡献

接下来，我们来学习一下作为项目贡献者，会有哪些常见的工作模式。

不过要说清楚整个协作过程真的很难，**Git** 如此灵活，人们的协作方式便可以各式各样，没有固定不变的范式可循，而每个项目的具体情况又多少会有些不同，比如说参与者的规模，所选择的工作流程，每个人的提交权限，以及 **Git** 以外贡献等等，都会影响到具体操作的细节。

首当其冲的是参与者规模。项目中有多少开发者是经常提交代码的？经常又是多久呢？大多数两至三人的小团队，一天大约只有几次提交，如果不是什么热门项目的话就更少了。可是在大公司里，或者大项目中，参与者可以多到上千，每天都会有十几个上百个补丁提交上来。这种差异带来的影响是显著的，越是多的人参与进来，就越难保证每次合并正确无误。你正在工作的代码，可能会因为合并进来其他人的更新而变得过时，甚至受创无法运行。而已经提交上去的更新，也可能在等着审核合并的过程中变得过时。那么，我们该怎样做才能确保代码是最新的，提交的补丁也是可用的呢？

接下来便是项目所采用的 workflow。是集中式的，每个开发者都具有等同的写权限？项目是否有专人负责检查所有补丁？是不是所有补丁都做过同行复阅（**peer-review**）再通过审核的？你是否参与审核过程？如果使用副官系统，那你是不是限定于只能向此副官提交？

还有你的提交权限。有或没有向主项目提交更新的权限，结果完全不同，直接决定最终采用怎样的 workflow。如果不能直接提交更新，那该如何贡献自己的代码呢？是不是该有个什么策略？你每次贡献代码会有多少量？提交频率呢？

所有以上这些问题都会或多或少影响到最终采用的 workflow。接下来，我会在一系列由简入繁的具体用例中，逐一阐述。此后在实践时，应该可以借鉴这里的例子，略作调整，以满足实际需要构建自己的工作流。

提交指南

开始分析特定用例之前，先来了解下如何撰写提交说明。一份好的提交指南可以帮助协作更轻松更有效地配合。**Git** 项目本身就提供了一份文档（**Git** 项目源代码目录中 **Documentation/SubmittingPatches**），列数了大量提示，从如何编撰提交说明到提交补丁，不一而足。

首先，请不要在更新中提交多余的白字符（**whitespace**）。**Git** 有种检查此类问题的方法，在提交之前，先运行 **git diff --check**，会把可能的多余白字符修正列出来。下面的示例，我已经把终端中显示为红色的白字符用 **X** 替换掉：

```
$ git diff --check

lib/simplegit.rb:5: trailing whitespace.
+   @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+   def command(git_cmd)XXXX
```

这样在提交之前你就可以看到这类问题，及时解决以免困扰其他开发者。

接下来，请将每次提交限定于完成一次逻辑功能。并且可能的话，适当地分解为多次小更新，以便每次小型提交都更易于理解。请不要在周末穷追猛打一次性 解决五个问题，而最后拖到周一再提交。就算是这样也请尽可能利用暂存区域，将之前的改动分解为每次修复一个问题，再分别提交和加注说明。如果针对两个问题 改动的是同一个文件，可以试试看 **git add --patch** 的方式将部分内容置入暂存区域（我们会在第六章再详细介绍）。无论是五次小提交还是混杂在一起的大提交，最终分支末端的项目快照应该还是一样的，但分解开 来之后，更便于其他开发者复阅。这么做也方便自己将来取消某个特定问题的修复。我们将在第六章介绍一些重写提交历史，同暂存区域交互的技巧和工具，以便最 终得到一个干净有意义，且易于理解的提交历史。

最后需要谨记的是提交说明的撰写。写得好可以让大家协作起来更轻松。一般来说，提交说明最好限制在一行以内，**50** 个字符以下，简明扼要地描述更新内容，空开一行后，再展开详细注解。**Git** 项目本身需要开发者撰写详尽注解，包括本次修订的因由，以及前后不同实现之间的比较，我们也该借鉴这种做法。另外，提交说明应该用祈使现在式语态，比如，不要说成 “**I added tests for**” 或 “**Adding tests for**” 而应该用 “**Add tests for**”。下面是来自 **tpope.net** 的 **Tim Pope** 原创的提交说明格式模版，供参考：

本次更新的简要描述（50 个字符以内）

如果必要，此处展开详尽阐述。段落宽度限定在 72 个字符以内。

某些情况下，第一行的简要描述将用作邮件标题，其余部分作为邮件正文。

其间的空行是必要的，以区分两者（当然没有正文另当别论）。

如果并在一起，`rebase` 这样的工具就可能会迷惑。

另起空行后，再进一步补充其他说明。

- 可以使用这样的条目列举式。
 - 一般以单个空格紧跟短划线或者星号作为每项条目的起始符。每个条目间用一空行隔开。
- 不过这里按自己项目的约定，可以略作变化。

如果你的提交说明都用这样的格式来书写，好多事情就可以变得十分简单。**Git** 项目本身就是这样要求的，我强烈建议你到 **Git** 项目仓库下运行 `git log --no-merges` 看看，所有提交历史的说明是怎样撰写的。（译注：如果现在还没有克隆 **git** 项目源代码，是时候 `git clone git://git.kernel.org/pub/scm/git/git.git` 了。）

为简单起见，在接下来的例子（及本书随后的所有演示）中，我都不会用这种格式，而使用 `-m` 选项提交 `git commit`。不过请还是按照我之前讲的做，别学我这里偷懒的方式。

私有的小型团队

我们从最简单的情况开始，一个私有项目，与你一起协作的还有另外一到两位开发者。这里说私有，是指源代码不公开，其他人无法访问项目仓库。而你和其他开发者则都具有推送数据到仓库的权限。

这种情况下，你们可以用 **Subversion** 或其他集中式版本控制系统类似的工作流来协作。你仍然可以得到 **Git** 带来的其他好处：离线提交，快速分支与合并等等，但工作流程还是差不多的。主要区别在于，合并操作发生在客户端而非服务器上。让我们来看看，两个开发者一起使用同一个共享仓库，会发生些什么。第一个人，**John**，克隆了仓库，作了些更新，在本地提交。（下面的例子中省略了常规提示，用... 代替以节约版面。）

```
# John's Machine
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
1 files changed, 1 insertions(+), 1 deletions(-)
```

第二个开发者，**Jessica**，一样这么做：克隆仓库，提交更新：

```
# Jessica's Machine
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
1 files changed, 1 insertions(+), 0 deletions(-)
```

现在，**Jessica** 将她的工作推送到服务器上：

```
# Jessica's Machine
$ git push origin master
...
To jessica@github:simplegit.git
1edee6b..fbff5bc master -> master
```

John 也尝试推送自己的工作上去：

```
# John's Machine
$ git push origin master
To john@github:simplegit.git
! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@github:simplegit.git'
```

John 的推送操作被驳回，因为 **Jessica** 已经推送了新的数据上去。请注意，特别是你用惯了 **Subversion** 的话，这里其实修改的是两个文件，而不是同一个文件的同一个地方。**Subversion** 会在服务器端自动合并提交上来的更新，而 **Git** 则必须先在本地上合并后才能推送。于是，**John** 不得不先把 **Jessica** 的更新拉下来：

```
$ git fetch origin
...
From john@github:simplegit
+ 049d078...fbff5bc master -> origin/master
```

此刻，**John** 的本地仓库如图 5-4 所示：

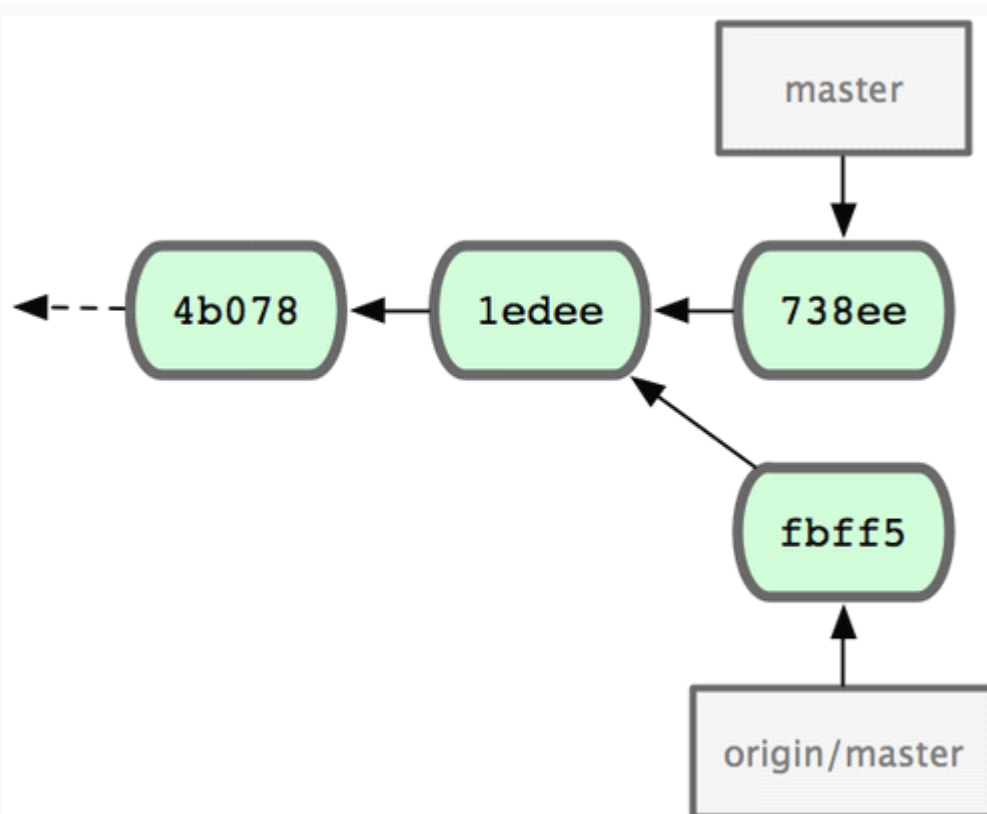


图 5-4. John 的仓库历史虽然 John 下载了 Jessica 推送到服务器的最近更新（fbff5），但目前只是 origin/master 指针指向它，而当前的本地分支 master 仍然指向自己的更新（738ee），所以需要先把她的提交合并过来，才能继续推送数据：

```
$ git merge origin/master
Merge made by recursive.
  TODO | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

还好，合并过程非常顺利，没有冲突，现在 John 的提交历史如图 5-5 所示：

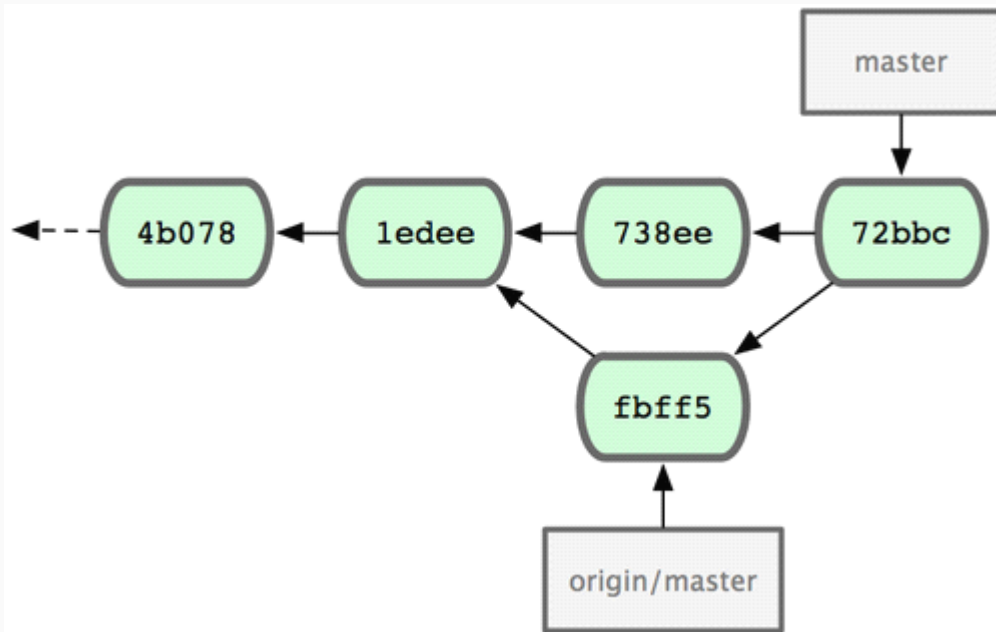


图 5-5. 合并 origin/master 后 John 的仓库历史现在，John 应该再测试一下代码是否仍然正常工作，然后将合并结果（72bbc）推送到服务器上：

```

$ git push origin master
...
To john@github:john@github:simplegit.git
fbff5bc..72bbc59 master -> master
  
```

最终，John 的提交历史变为图 5-6 所示：

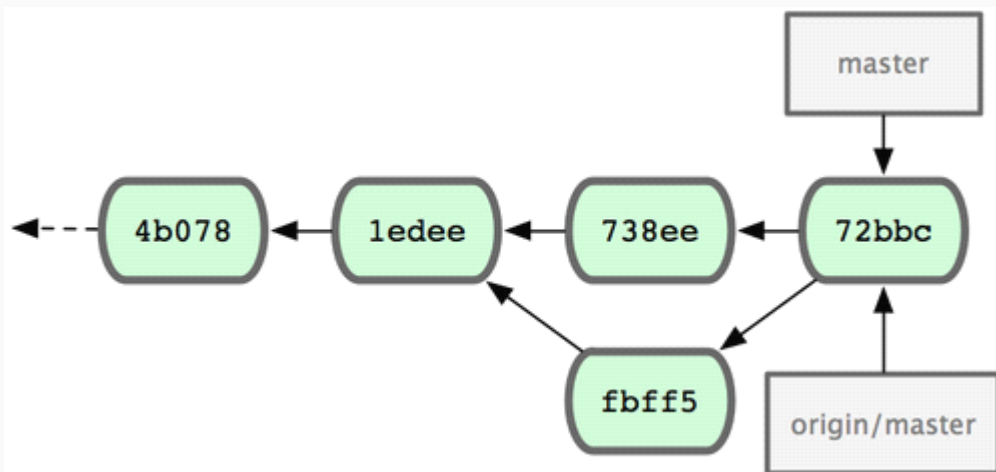


图 5-6. 推送后 John 的仓库历史而在这段时间，Jessica 已经开始在另一个特性分支工作了。她创建了 issue54 并提交了三更新。她还没有下载 John 提交的合并结果，所以提

交历史如图 5-7 所示：

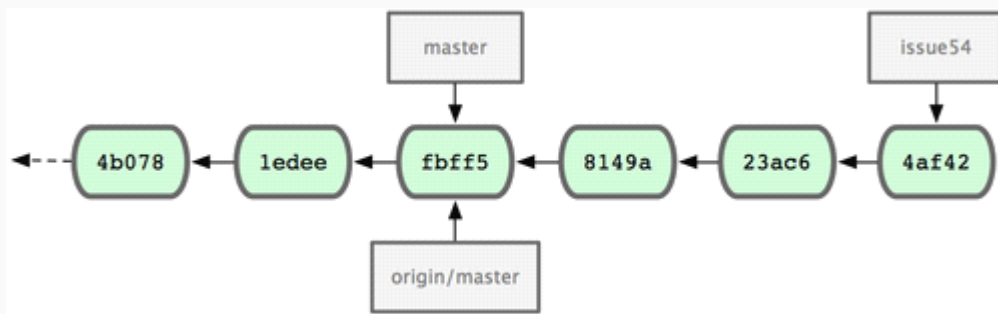


图 5-7. Jessica 的提交历史 Jessica 想要先和服务器上的数据同步，所以先下载数据：

```
# Jessica's Machine
$ git fetch origin
...
From jessica@github:simplegit
 fbff5bc..72bbc59 master    -> origin/master
```

于是 Jessica 的本地仓库历史多出了 John 的两次提交 (738ee 和 72bbc)，如图 5-8 所示：

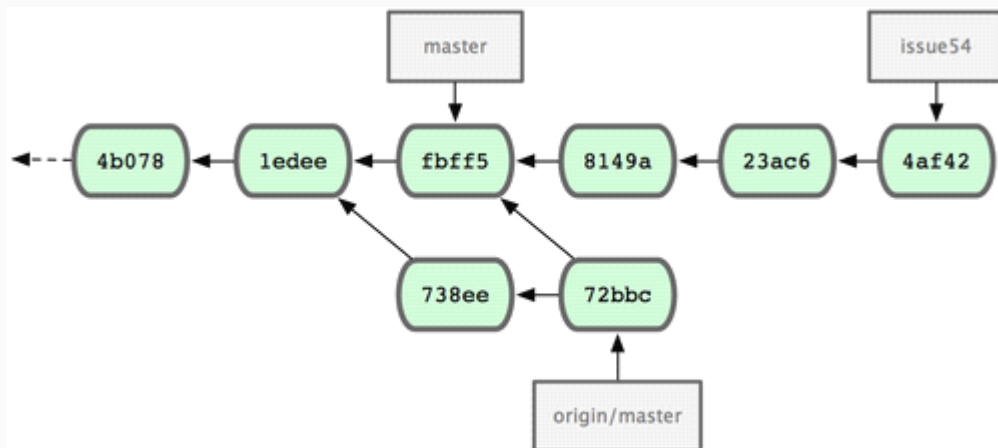


图 5-8. 获取 John 的更新之后 Jessica 的提交历史此时，Jessica 在特性分支上的工作已经完成，但她想在推送数据之前，先确认下要并进来的数据究竟是什么，于是运行 git log 查看：

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith
```



```
Date: Fri May 29 16:01:27 2009 -0700 removed invalid default value
```

现在，**Jessica** 可以将特性分支上的工作并到 **master** 分支，然后再并入 **John** 的工作（**origin/master**）到自己的 **master** 分支，最后再推送回服务器。当然，得先切回主分支才能集成所有数据：

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

要合并 **origin/master** 或 **issue54** 分支，谁先谁后都没有关系，因为它们都在上游（**upstream**）（译注：想像分叉的更新像是汇流成河的源头，所以上游 **upstream** 是指最新的提交），所以无所谓先后顺序，最终合并后的内容快照都是一样的，而仅是提交历史看起来会有些先后差别。**Jessica** 选择先合并 **issue54**：

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |    1 +
lib/simplegit.rb |    6 +++++-
2 files changed, 6 insertions(+), 1 deletions(-)
```

正如图所示，没有冲突发生，仅是一次简单快进。现在 **Jessica** 开始合并 **John** 的工作（**origin/master**）：

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)
```

所有的合并都非常干净。现在 **Jessica** 的提交历史如图 5-9 所示：

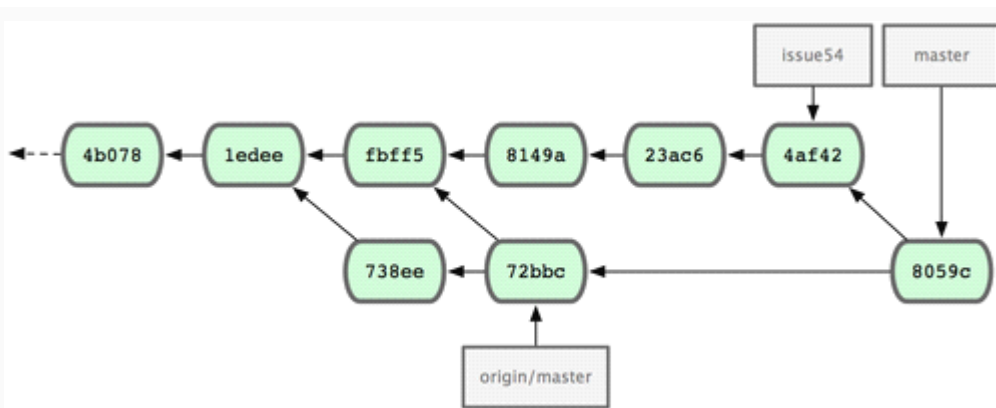


图 5-9. 合并 John 的更新后 Jessica 的提交历史现在 Jessica 已经可以在自己的 master 分支中访问 origin/master 的最新改动了，所以她应该可以成功推送最后的合并结果到服务器上（假设 John 此时没再推送新数据上来）：

```

$ git push origin master
...
To jessica@github:simplegit.git
  72bbc59..8059c15 master -> master

```

至此，每个开发者都提交了若干次，且成功合并了对方的工作成果，最新的提交历史如图 5-10 所示：

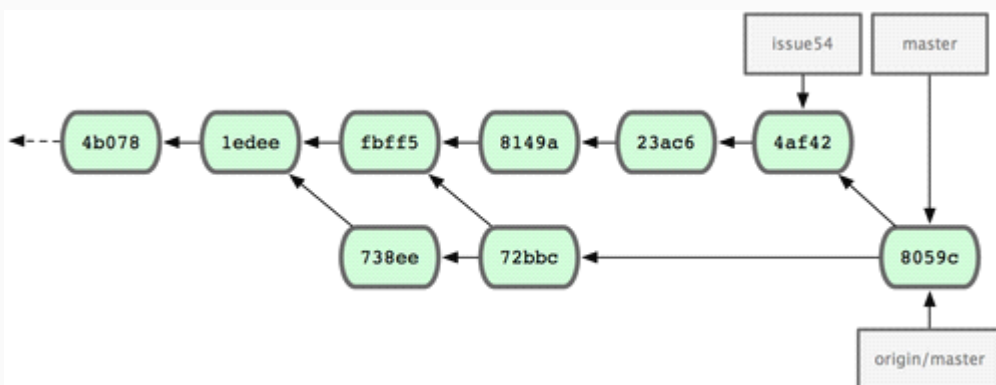


图 5-10. Jessica 推送数据后的提交历史以上就是最简单的协作方式之一：先在自己的特性分支中工作一段时间，完成后合并到自己的 master 分支；然后下载合并 origin/master 上的更新（如果有的话），再推回远程服务器。一般的协作流程如图 5-11 所示：

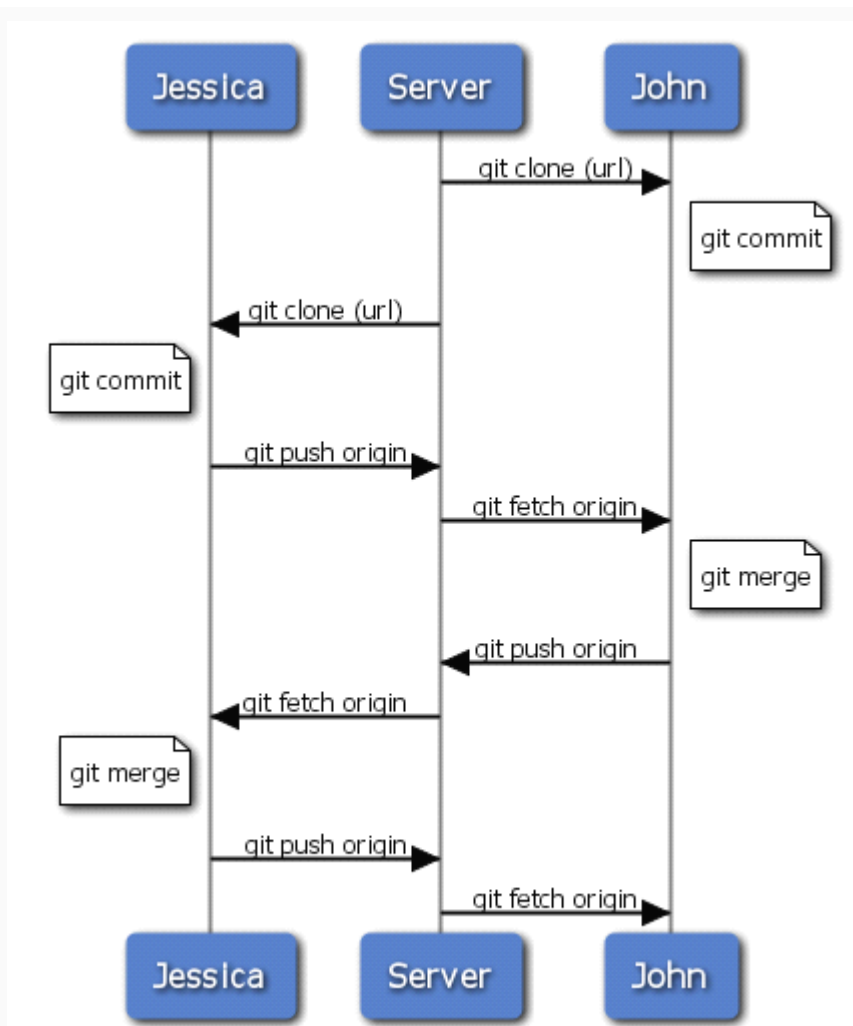


图 5-11. 多用户共享仓库协作方式的一般工作流程时序**私有团队间协作**

现在我们来查看更大一点规模的私有团队协作。如果有几个小组分头负责若干特性的开发和集成，那他们之间的协作过程是怎样的。

假设 John 和 Jessica 一起负责开发某项特性 A，而同时 Jessica 和 Josie 一起负责开发另一项功能 B。公司使用典型的集成管理员式工作流，每个组都有一名管理员负责集成本组代码，及更新项目主仓库的 master 分支。所有开发都在代表小组的分支上进行。

让我们跟随 Jessica 的视角看看她的工作流程。她参与开发两项特性，同时和不同小组的开发者一起协作。克隆生成本地仓库后，她打算先着手开发特性 A。于是创建了新的 featureA 分支，继而编写代码：

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch "featureA"
$ vim lib/simplegit.rb
```

```
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
1 files changed, 1 insertions(+), 1 deletions(-)
```

此刻，她需要分享目前的进展给 **John**，于是她将自己的 **featureA** 分支提交到服务器。由于 **Jessica** 没有权限推送数据到主仓库的 **master** 分支（只有集成管理员有此权限），所以只能将此分支推上去同 **John** 共享协作：

```
$ git push origin featureA
...
To jessica@github:simplegit.git
* [new branch]      featureA -> featureA
```

Jessica 发邮件给 **John** 让他上来看 **featureA** 分支上的进展。在等待他的反馈之前，**Jessica** 决定继续工作，和 **Josie** 一起开发 **featureB** 上的特性 **B**。当然，先创建此分支，分叉点以服务器上的 **master** 为起点：

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

随后，**Jessica** 在 **featureB** 上提交了若干更新：

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
1 files changed, 5 insertions(+), 0 deletions(-)
```

现在 **Jessica** 的更新历史如图 5-12 所示：

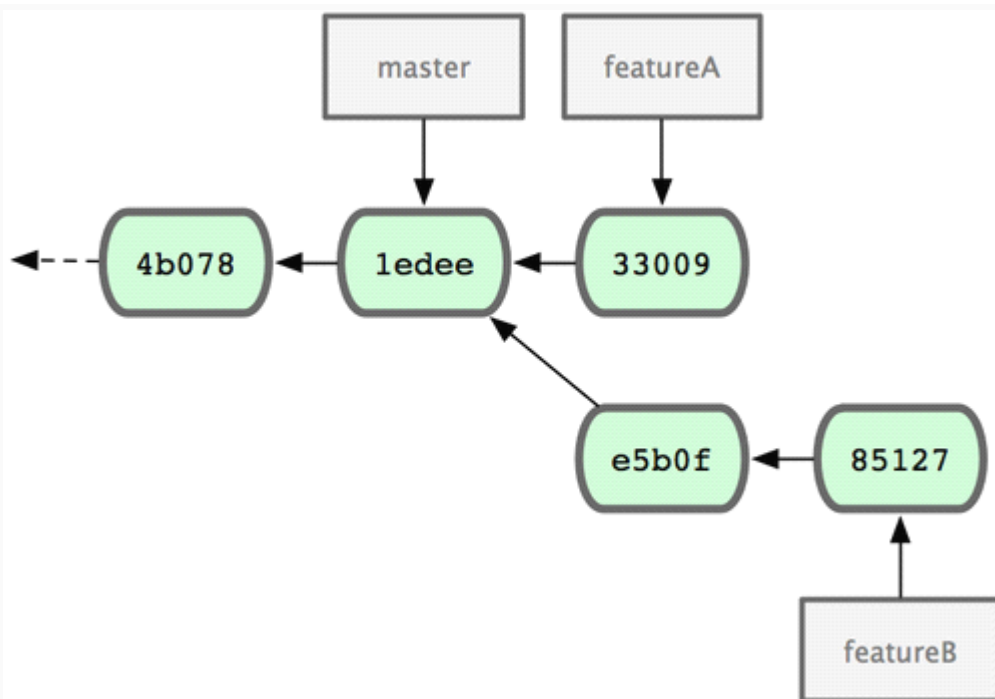


图 5-12. Jessica 的更新历史 Jessica 正准备推送自己的进展上去，却收到 Josie 的来信，说是她已经将自己的工作推到服务器上的 **featureBee** 分支了。这样，Jessica 就必须先将 Josie 的代码合并到自己本地分支中，才能再一起推送回服务器。她用 `git fetch` 下载 Josie 的最新代码：

```
$ git fetch origin
...
From jessica@github:simplegit
* [new branch]      featureBee -> origin/featureBee
```

然后 Jessica 使用 `git merge` 将此分支合并到自己分支中：

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb | 4 ++++
1 files changed, 4 insertions(+), 0 deletions(-)
```

合并很顺利，但另外有个小问题：她要推送自己的 **featureB** 分支到服务器上的 **featureBee** 分支上去。当然，她可以使用冒号 (:) 格式指定目标分支：

```
$ git push origin featureB:featureBee
...
To jessica@github:simplegit.git
```

```
fba9af8..cd685d1 featureB -> featureBee
```

我们称此为 `_refspec_`。更多有关于 `Git refspec` 的讨论和使用方式会在第九章作详细阐述。

接下来, **John** 发邮件给 **Jessica** 告诉她, 他看了之后作了些修改, 已经推回服务器 `featureA` 分支, 请她过目下。于是 **Jessica** 运行 `git fetch` 下载最新数据:

```
$ git fetch origin
...
From jessica@github:simplegit
 3300904..aad881d featureA -> origin/featureA
```

接下来便可以用 `git log` 查看更新了些什么:

```
$ git log origin/featureA ^featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith
```

```
Date: Fri May 29 19:57:33 2009 -0700 changed log output to 30 from 25
```

最后, 她将 **John** 的工作合并到自己的 `featureA` 分支中:

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb | 10 ++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica 稍做一番修整后同步到服务器:

```
$ git commit -am 'small tweak'
[featureA ed774b3] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@github:simplegit.git
 3300904..ed774b3 featureA -> featureA
```

现在的 **Jessica** 提交历史如图 5-13 所示:

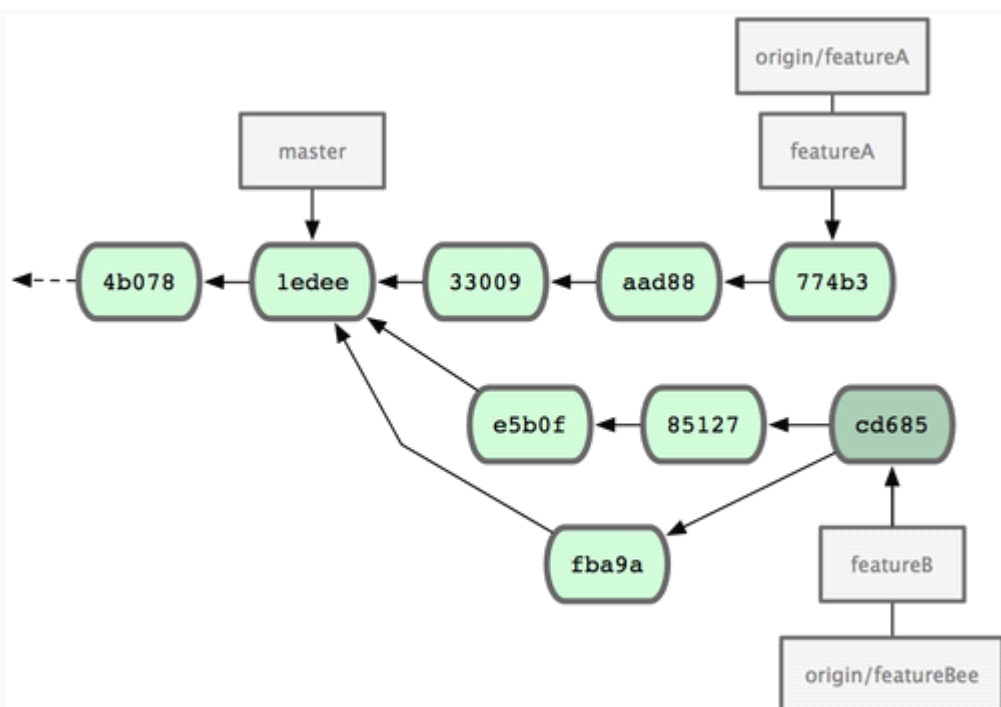


图 5-13. 在特性分支中提交更新后的提交历史现在，Jessica，Josie 和 John 通知集成管理员服务器上的 **featureA** 及 **featureBee** 分支已经准备好，可以并入主线了。在管理员完成集成工作后，主分支上便多出一个新的合并提交（5399e），用 **fetch** 命令更新到本地后，提交历史如图 5-14 所示：

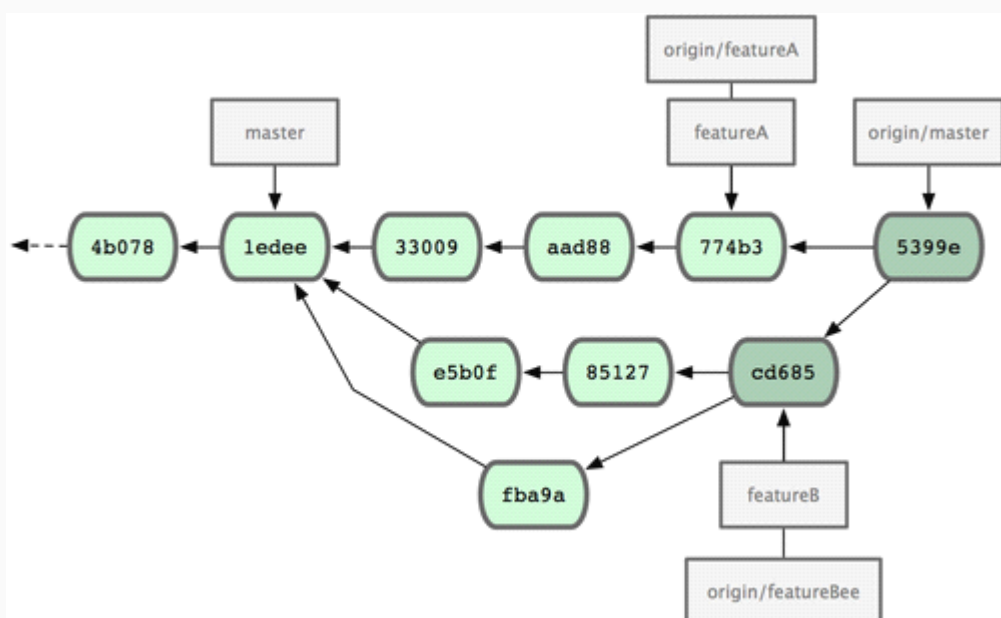


图 5-14. 合并特性分支后的 Jessica 提交历史许多开发小组改用 Git 就是因为它允许多个小组间并行工作，而在稍后恰当时机再行合并。通过共享远程分支的方式，无需干扰整体项目代码便可以开展工作，因此使用 Git 的小型团队间协作可以变得非常灵活自由。以上

工作流程的时序如图 5-15 所示：

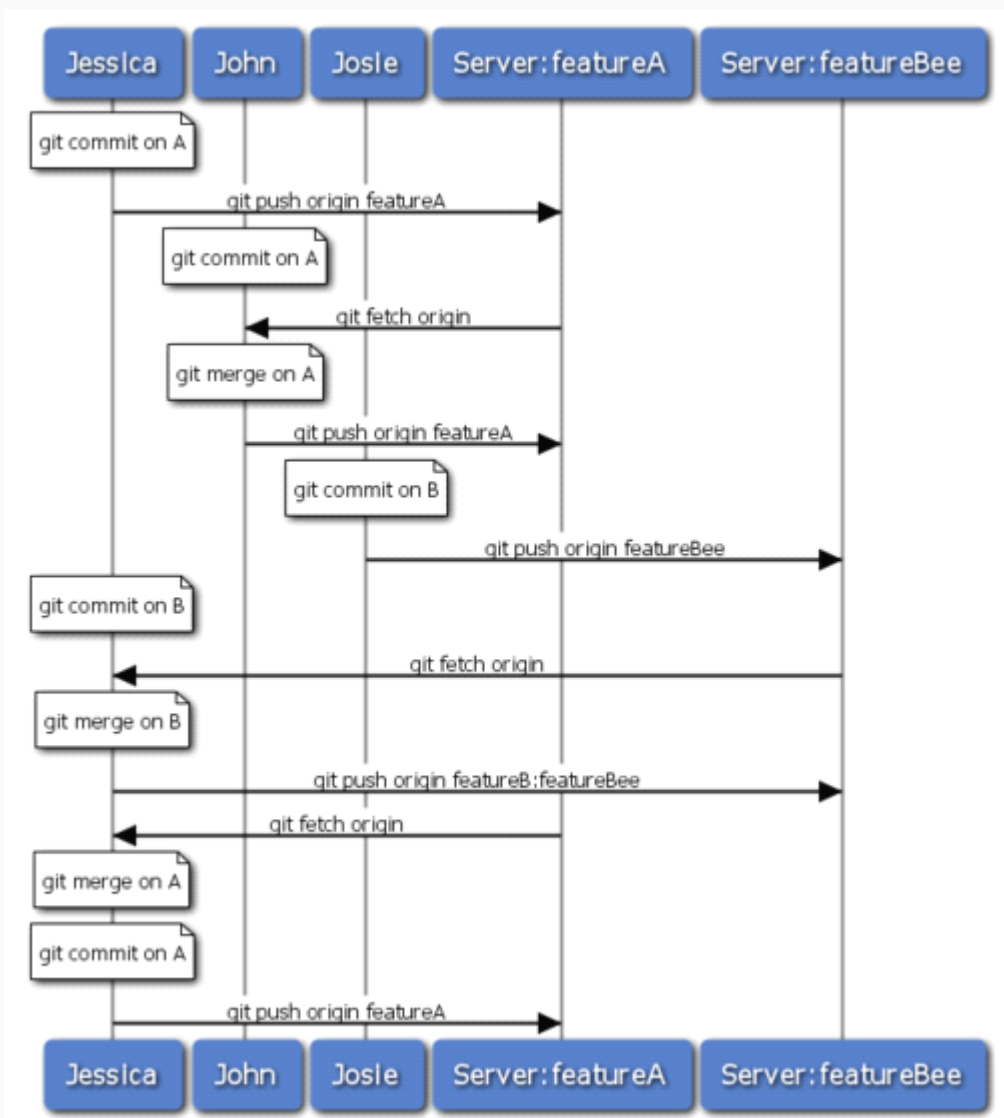


图 5-15. 团队间协作工作流程基本时序公开的小型项目

上面说的是私有项目协作，但要给公开项目作贡献，情况就有些不同了。因为你没有直接更新主仓库分支的权限，得寻求其它方式把工作成果交给项目维护人。下面会介绍两种方法，第一种使用 git 托管服务商提供的仓库复制功能，一般称作 **fork**，比如 **repo.or.cz** 和 **GitHub** 都支持这样的操作，而且许多项目管理员都希望大家使用这样的方式。另一种方法是通过电子邮件寄送文件补丁。

但不管哪种方式，起先我们总需要克隆原始仓库，而后创建特性分支开展工作。基本工作流程如下：

```
$ git clone (url)
$ cd project
```



```
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

你可能想到用 **rebase -i** 将所有更新先变作单个提交，又或者想重新安排提交之间的差异补丁，以方便项目维护者审阅 – 有关交互式衍合操作的细节见第六章。

在完成了特性分支开发，提交给项目维护者之前，先到原始项目的页面上点击“**Fork**”按钮，创建一个自己可写的公共仓库（译注：即下面的 **url** 部分，参照后续的例子，应该是 **git://githost/simplegit.git**）。然后将此仓库添加为本地的第二个远端仓库，姑且称为 **myfork**：

```
$ git remote add myfork (url)
```

你需要将本地更新推送到这个仓库。要是将远端 **master** 合并到本地再推回去，还不如把整个特性分支推上去来得干脆直接。而且，假若项目维护者未采纳你的贡献的话（不管是直接合并还是 **cherry pick**），都不用回退（**rewind**）自己的 **master** 分支。但若维护者合并或 **cherry-pick** 了你的工作，最后总还可以从他们的更新中同步这些代码。好吧，现在先把 **featureA** 分支整个推上去：

```
$ git push myfork featureA
```

然后通知项目管理员，让他来抓取你的代码。通常我们把这件事叫做 **pull request**。可以直接用 **GitHub** 等网站提供的“**pull request**”按钮自动发送请求通知；或手工把 **git request-pull** 命令输出结果电邮给项目管理员。

request-pull 命令接受两个参数，第一个是本地特性分支开始前的原始分支，第二个是请求对方来抓取的 **Git** 仓库 **URL**（译注：即下面 **myfork** 所指的，自己可写的公共仓库）。比如现在 **Jessica** 准备要给 **John** 发一个 **pull request**，她之前在自己的特性分支上提交了两次更新，并把分支整个推到了服务器上，所以运行该命令会看到：

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
    John Smith (1):
        added a new function

are available in the git repository at:

    git://githost/simplegit.git featureA

Jessica Smith (2):
```

```
add limit to log function  
change log output to 30 from 25
```

```
lib/simplegit.rb | 10 +++++++--
```

```
1 files changed, 9 insertions(+), 1 deletions(-)
```

输出的内容可以直接发邮件给管理者，他们就会明白这是从哪次提交开始旁支出去的，该到哪里去抓取新的代码，以及新的代码增加了哪些功能等等。

像这样随时保持自己的 **master** 分支和官方 **origin/master** 同步，并将自己的工作限制在特性分支上的做法，既方便又灵活，采纳和丢弃都轻而易举。就算原始主干发生变化，我们也能重新衍合提供新的补丁。比如现在要开始第二项特性的开发，不要在原来已推送的特性分支上继续，还是按原始 **master** 开始：

```
$ git checkout -b featureB origin/master  
$ (work)  
$ git commit  
$ git push myfork featureB  
$ (email maintainer)  
$ git fetch origin
```

现在，A、B 两个特性分支各不相扰，如同竹筒里的两颗豆子，队列中的两个补丁，你随时都可以分别从头写过，或者衍合，或者修改，而不用担心特性代码的交叉混杂。如图 5-16 所示：

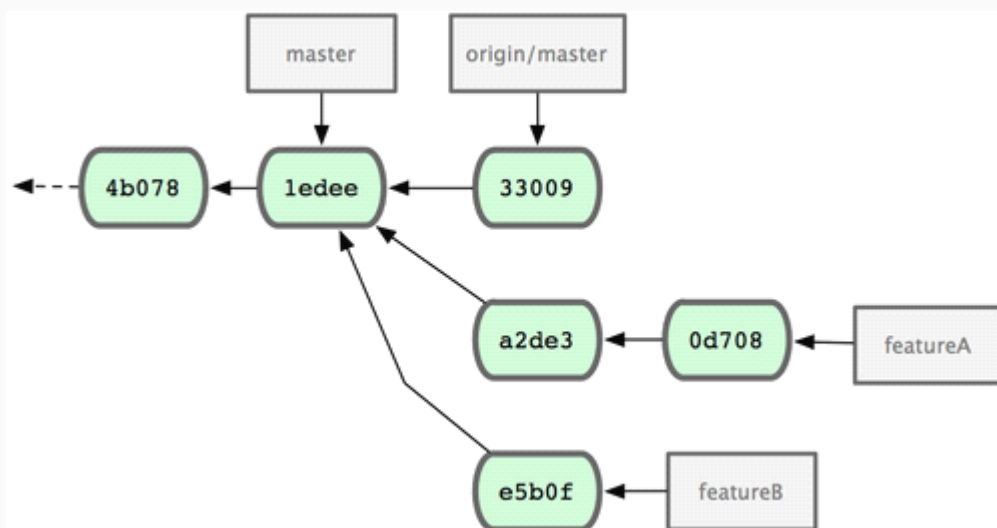


图 5-16. **featureB** 以后的提交历史假设项目管理员接纳了许多别人提交的补丁后，准备要采纳你提交的第一个分支，却发现因为代码基准不一致，合并工作无法正确干净地完成。这就需要你再次衍合到最新的 **origin/master**，解决相关冲突，然后重新提交你的修改：

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

自然，这会重写提交历史，如图 5-17 所示：

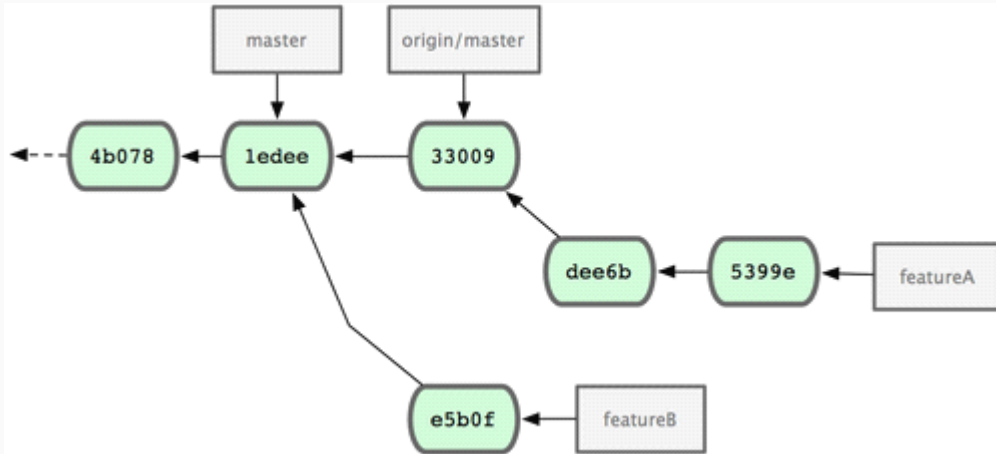


图 5-17. **featureA** 重新衍合后的提交历史注意，此时推送分支必须使用 **-f** 选项（译注：表示 **force**，不作检查强制重写）替换远程已有的 **featureA** 分支，因为新的 **commit** 并非原来的后续更新。当然你也可以直接推送到另一个新的分支上去，比如称作 **featureAv2**。

再考虑另一种情形：管理员看过第二个分支后觉得思路新颖，但想请你改下具体实现。我们只需以当前 **origin/master** 分支为基准，开始一个新的特性分支 **featureBv2**，然后把原来的 **featureB** 的更新拿过来，解决冲突，按要求重新实现部分代码，然后将此特性分支推送上去：

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

这里的 **--squash** 选项将目标分支上的所有更改全拿来应用到当前分支上，而 **--no-commit** 选项告诉 **Git** 此时无需自动生成和记录（合并）提交。这样，你就可以在原来代码基础上，继续工作，直到最后一起提交。

好了，现在可以请管理员抓取 **featureBv2** 上的最新代码了，如图 5-18 所示：

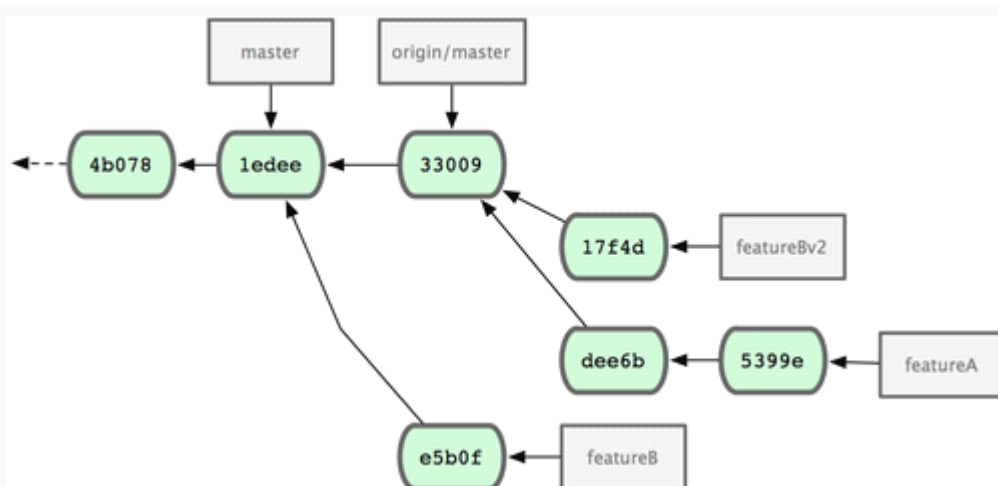


图 5-18. featureBv2 之后的提交历史公开的大型项目

许多大型项目都会立有一套自己的接受补丁流程，你应该注意下其中细节。但多数项目都允许通过开发者邮件列表接受补丁，现在我们来看具体例子。

整个工作流程类似上面的情形：为每个补丁创建独立的特性分支，而不同之处在于如何提交这些补丁。不需要创建自己可写的公共仓库，也不用将自己的更新推送到自己的服务器，你只需将每次提交的差异内容以电子邮件的方式依次发送到邮件列表中即可。

```

$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit

```

如此一番后，有了两个提交要发到邮件列表。我们可以用 `git format-patch` 命令来生成 `mbox` 格式的文件然后作为附件发送。每个提交都会封装为一个 `.patch` 后缀的 `mbox` 文件，但其中只包含一封邮件，邮件标题就是提交消息（译注：额外有前缀，看例子），邮件内容包含补丁正文和 `Git` 版本号。这种方式的妙处在于接受补丁时仍可保留原来的提交消息，请看接下来的例子：

```

$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch

```

`format-patch` 命令依次创建补丁文件，并输出文件名。上面的 `-M` 选项允许 `Git` 检查是否有对文件重命名的提交。我们来看看补丁文件的内容：

```

$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001

```

From: Jessica Smith

```
Date: Sun, 6 Apr 2008 10:17:23 -0700 Subject: [PATCH 1/2] add limit to log
function Limit log functionality to the first 20 --- lib/simplegit.rb | 2 +- 1
files changed, 1 insertions(+), 1 deletions(-) diff --git a/lib/simplegit.rb
b/lib/simplegit.rb index 76f47bc..f9815f1 100644 --- a/lib/simplegit.rb +++
b/lib/simplegit.rb @@ -14,7 +14,7 @@ class SimpleGit end def log(treeish = 'master')
- command("git log #{treeish}") + command("git log -n 20 #{treeish}") end def
ls_tree(treeish = 'master') -- 1.6.2.rc1.20.g8c5b.dirty
```

如果有额外信息需要补充，但又不想放在提交消息中说明，可以编辑这些补丁文件，在第一个 `---` 行之前添加说明，但不要修改下面的补丁正文，比如例子中的 **Limit log functionality to the first 20** 部分。这样，其它开发者能阅读，但在采纳补丁时不会将此合并进来。

你可以用邮件客户端软件发送这些补丁文件，也可以直接在命令行发送。有些所谓智能的邮件客户端软件会自作主张帮你调整格式，所以粘贴补丁到邮件正文时，有可能会丢失换行符和若干空格。Git 提供了一个通过 IMAP 发送补丁文件的工具。接下来我会演示如何通过 Gmail 的 IMAP 服务器发送。另外，在 Git 源代码中有个 **Documentation/SubmittingPatches** 文件，可以仔细阅读，看看其它邮件程序的相关导引。

首先在 `~/.gitconfig` 文件中配置 `imap` 项。每个选项都可用 `git config` 命令分别设置，当然直接编辑文件添加以下内容更便捷：

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

如果你的 IMAP 服务器没有启用 SSL，就无需配置最后那两行，并且 `host` 应该以 `imap://` 开头而不再是有 `s` 的 `imaps://`。保存配置文件后，就能用 `git send-email` 命令把补丁作为邮件依次发送到指定的 IMAP 服务器上的文件夹中（译注：这里就是 Gmail 的 **[Gmail]/Drafts** 文件夹。但如果你的语言设置不是英文，此处的文件夹 **Drafts** 字样会变为对应的语言。）：

```
$ git send-email *.patch
```

```
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith
```

```
] Emails will be sent from: Jessica Smith Who should the emails be sent to?
jessica@example.com Message-ID to be used as In-Reply-To for the first email?
y
```

接下来，Git 会根据每个补丁依次输出类似下面的日志：

```
(mbox) Adding cc: Jessica Smith
```

```
from \line 'From: Jessica Smith ' OK. Log says: Sendmail: /usr/sbin/sendmail
-i jessica@example.com From: Jessica Smith To: jessica@example.com Subject:
[PATCH 1/2] added limit to log function Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com> X-Mailer:
git-send-email 1.6.2.rc1.20.g8c5b.dirty In-Reply-To: References: Result: OK
```

最后，到 Gmail 上打开 Drafts 文件夹，编辑这些邮件，修改收件人地址为邮件列表地址，另外给要抄送的人也加到 Cc 列表中，最后发送。

小结

本节主要介绍了常见 Git 项目协作的工作流程，还有一些帮助处理这些工作的命令和工具。接下来我们要看看如何维护 Git 项目，并成为合格的项目管理员，或是集成经理。

5.3 项目的管理

既然是相互协作，在贡献代码的同时，也免不了要维护管理自己的项目。像是怎么处理别人用 format-patch 生成的补丁，或是集成远端仓库上某个分支上的变化等等。但无论是管理代码仓库，还是帮忙审核收到的补丁，都需要同贡献者约定某种长期可持续的工作方式。

使用特性分支进行工作

如果想要集成新的代码进来，最好局限在特性分支上做。临时的特性分支可以让你随意尝试，进退自如。比如碰上无法正常工作的补丁，可以先搁在那边，直到有时间仔细核查修复为止。创建的分支可以用相关的主题关键字命名，比如 ruby_client 或者其它类似的描述性词语，

以帮助将来回忆。**Git** 项目本身还时常把分支名称分置于不同命名空间下，比如 `sc/ruby_client` 就说明这是 **sc** 这个人贡献的。现在从当前主干分支为基础，新建临时分支：

```
$ git branch sc/ruby_client master
```

另外，如果你希望立即转到分支上去工作，可以用 `checkout -b`：

```
$ git checkout -b sc/ruby_client master
```

好了，现在已经准备妥当，可以试着将别人贡献的代码合并进来了。之后评估一下有没有问题，最后再决定是否真的要并入主干。

采纳来自邮件的补丁

如果收到一个通过电邮发来的补丁，你应该先把它应用到特性分支上进行评估。有两种应用补丁的方法：`git apply` 或者 `git am`。

使用 `apply` 命令应用补丁

如果收到的补丁文件是用 `git diff` 或由其它 **Unix** 的 `diff` 命令生成，就该用 `git apply` 命令来应用补丁。假设补丁文件存在 `/tmp/patch-ruby-client.patch`，可以这样运行：

```
$ git apply /tmp/patch-ruby-client.patch
```

这会修改当前工作目录下的文件，效果基本与运行 `patch -p1` 打补丁一样，但它更为严格，且不会出现混乱。如果是 `git diff` 格式描述的补丁，此命令还会相应地添加，删除，重命名文件。当然，普通的 `patch` 命令是不会这么做的。另外请注意，`git apply` 是一个事务性操作的命令，也就是说，要么所有补丁都打上去，要么全部放弃。所以不会出现 `patch` 命令那样，一部分文件打上了补丁而另一部分却没有，这样一种不上不下的修订状态。所以总的来说，`git apply` 要比 `patch` 严谨许多。因为仅仅是更新当前的文件，所以此命令不会自动生成提交对象，你得手工缓存相应文件的更新状态并执行提交命令。

在实际打补丁之前，可以先用 `git apply --check` 查看补丁是否能够干净顺利地应用到当前分支中：

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

如果没有任何输出，表示我们可以顺利采纳该补丁。如果有问题，除了报告错误信息之外，该命令还会返回一个非零的状态，所以在 `shell` 脚本里可用于检测状态。

使用 `am` 命令应用补丁

如果贡献者也用 **Git**，且擅于制作 `format-patch` 补丁，那你的合并工作将会非常轻松。因为这些补丁中除了文件内容差异外，还包含了作者信息和提交消息。所以请鼓励贡献者用

`format-patch` 生成补丁。对于传统的 `diff` 命令生成的补丁，则只能用 `git apply` 处理。

对于 `format-patch` 制作的新式补丁，应当使用 `git am` 命令。从技术上来说，`git am` 能够读取 `mbox` 格式的文件。这是种简单的纯文本文件，可以包含多封电邮，格式上用 `From` 加空格以及随便什么辅助信息所组成的行作为分隔行，以区分每封邮件，就像这样：

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith
```

```
Date: Sun, 6 Apr 2008 10:17:23 -0700 Subject: [PATCH 1/2] add limit to log
function Limit log functionality to the first 20
```

这是 `format-patch` 命令输出的开头几行，也是一个有效的 `mbox` 文件格式。如果有人用 `git send-email` 给你发了一个补丁，你可以将此邮件下载到本地，然后运行 `git am` 命令来应用这个补丁。如果你的邮件客户端能将多封电邮导出为 `mbox` 格式的文件，就可以用 `git am` 一次性应用所有导出的补丁。

如果贡献者将 `format-patch` 生成的补丁文件上传到类似 `Request Ticket` 一样的任务处理系统，那么可以先下载到本地，继而使用 `git am` 应用该补丁：

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

你会看到它被干净地应用到本地分支，并自动创建了新的提交对象。作者信息取自邮件头 `From` 和 `Date`，提交消息则取自 `Subject` 以及正文中补丁之前的内容。来看具体实例，采纳之前展示的那个 `mbox` 电邮补丁后，最新的提交对象为：

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith
```

```
AuthorDate: Sun Apr 6 10:17:23 2008 -0700 Commit: Scott Chacon CommitDate:
Thu Apr 9 09:19:06 2009 -0700 add limit to log function Limit log functionality
to the first 20
```

`Commit` 部分显示的是采纳补丁的人，以及采纳的时间。而 `Author` 部分则显示的是原作者，以及创建补丁的时间。

有时,我们也会遇到打不上补丁的情况。这多半是因为主干分支和补丁的基础分支相差太远,但也可能是因为某些依赖补丁还未应用。这种情况下, **git am** 会报错并询问该怎么做:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.

When you have resolved this problem run "git am --resolved".

If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Git 会在有冲突的文件里加入冲突解决标记,这同合并或衍合操作一样。解决的办法也一样,先编辑文件消除冲突,然后暂存文件,最后运行 **git am --resolved** 提交修正结果:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

如果想让 **Git** 更智能地处理冲突,可以用 **-3** 选项进行三方合并。如果当前分支未包含该补丁的基础代码或其祖先,那么三方合并就会失败,所以该选项默认为关闭状态。一般来说,如果该补丁是基于某个公开的提交制作而成的话,你总是可以通过同步来获取这个共同祖先,所以用三方合并选项可以解决很多麻烦:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

像上面的例子,对于打过的补丁我又再打一遍,自然会产生冲突,但因为加上了 **-3** 选项,所以它很聪明地告诉我,无需更新,原有的补丁已经应用。

对于一次应用多个补丁时所用的 **mbox** 格式文件,可以用 **am** 命令的交互模式选项 **-i**,这样就会在打每个补丁前停住,询问该如何操作:

```
$ git am -3 -i mbox
Commit Body is:
-----
```

```
seeing if this helps the gem
```

```
-----  
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

在多个补丁要打的情况下，这是个非常好的办法，一方面可以预览下补丁内容，同时也可以有选择性的接纳或跳过某些补丁。

打完所有补丁后，如果测试下来新特性可以正常工作，那就可以安心地将当前特性分支合并到长期分支中去了。

检出远程分支

如果贡献者有自己的 **Git** 仓库，并将修改推送到此仓库中，那么当你拿到仓库的访问地址和对应分支的名称后，就可以加为远程分支，然后在本地进行合并。

比如，**Jessica** 发来一封邮件，说在她代码库中的 **ruby-client** 分支上已经实现了某个非常棒的新功能，希望我们能帮忙测试一下。我们可以先把她的仓库加为远程仓库，然后抓取数据，完了再将她所说的分支检出到本地来测试：

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

若是不久她又发来邮件，说还有个很棒的功能实现在另一分支上，那我们只需重新抓取下最新数据，然后检出那个分支到本地就可以了，无需重复设置远程仓库。

这种做法便于同别人保持长期的合作关系。但前提是要求贡献者有自己的服务器，而我们也需为每个人建一个远程分支。有些贡献者提交代码补丁并不是很频繁，所以通过邮件接收补丁效率会更高。同时我们自己也不会希望建上百来个分支，却只从每个分支取一两个补丁。但若是用脚本程序来管理，或直接使用代码仓库托管服务，就可以简化此过程。当然，选择何种方式取决于你和贡献者的喜好。

使用远程分支的另外一个好处是能够得到提交历史。不管代码合并是不是会有问题，至少我们知道该分支的历史分叉点，所以默认会从共同祖先开始自动进行三方合并，无需 **-3** 选项，也不用像打补丁那样祈祷存在共同的基准点。

如果只是临时合作，只需用 **git pull** 命令抓取远程仓库上的数据，合并到本地临时分支就可以了。一次性的抓取动作自然不会把该仓库地址加为远程仓库。

```
$ git pull git://github.com/onetimeguy/project.git  
From git://github.com/onetimeguy/project  
* branch          HEAD      -> FETCH_HEAD  
Merge made by recursive.
```

决断代码取舍

现在特性分支上已合并好了贡献者的代码，是时候决断取舍了。本节将回顾一些之前学过的命令，以看清将要合并到主干的是哪些代码，从而理解它们到底做了些什么，是否真的要并入。

一般我们会先看下，特性分支上都有哪些新增的提交。比如在 **contrib** 特性分支上打了两个补丁，仅查看这两个补丁的提交信息，可以用 **--not master** 选项指定要屏蔽的分支 **master**，这样就会剔除重复的提交历史：

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon
```

```
Date: Fri Oct 24 09:53:59 2008 -0700 seeing if this helps the gem commit
7482e0d16d04bea79d0dba8988cc78df655f16a0 Author: Scott Chacon Date: Mon Oct 22
19:38:36 2008 -0700 updated the gemspec to hopefully work better
```

还可以查看每次提交的具体修改。请牢记，在 **git log** 后加 **-p** 选项将展示每次提交的内容差异。

如果想看当前分支同其他分支合并时的完整内容差异，有个小窍门：

```
$ git diff master
```

虽然能得到差异内容，但请记住，结果有可能和我们的预期不同。一旦主干 **master** 在特性分支创建之后有所修改，那么通过 **diff** 命令来比较的，是最新主干上的提交快照。显然，这不是我们所要的。比方在 **master** 分支中某个文件里添了一行，然后运行上面的命令，简单的比较最新快照所得到的结论只能是，特性分支中删除了这一行。

这个很好理解：如果 **master** 是特性分支的直接祖先，不会产生任何问题；如果它们的提交历史在不同的分叉上，那么产生的内容差异，看起来就像是增加了特性分支上的新代码，同时删除了 **master** 分支上的新代码。

实际上我们真正想要看的，是新加入到特性分支的代码，也就是合并时会并入主干的代码。所以，准确地讲，我们应该比较特性分支和它同 **master** 分支的共同祖先之间的差异。

我们可以手工定位它们的共同祖先，然后与之比较：

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
```

```
$ git diff 36c7db
```

但这么做很麻烦，所以 **Git** 提供了便捷的 ... 语法。对于 **diff** 命令，可以把 ... 加在原始分支（拥有共同祖先）和当前分支之间：

```
$ git diff master...contrib
```

现在看到的，就是实际将要引入的新代码。这是一个非常有用的命令，应该牢记。

代码集成

一旦特性分支准备停当，接下来的问题就是如何集成到更靠近主线的分支中。此外还要考虑维护项目的总体步骤是什么。虽然有很多选择，不过我们这里只介绍其中一部分。

合并流程

一般最简单的情形，是在 **master** 分支中维护稳定代码，然后在特性分支上开发新功能，或是审核测试别人贡献的代码，接着将它并入主干，最后删除这个特性分支，如此反复。来看示例，假设当前代码库中有两个分支，分别为 **ruby_client** 和 **php_client**，如图 5-19 所示。然后先把 **ruby_client** 合并进主干，再合并 **php_client**，最后的提交历史如图 5-20 所示。

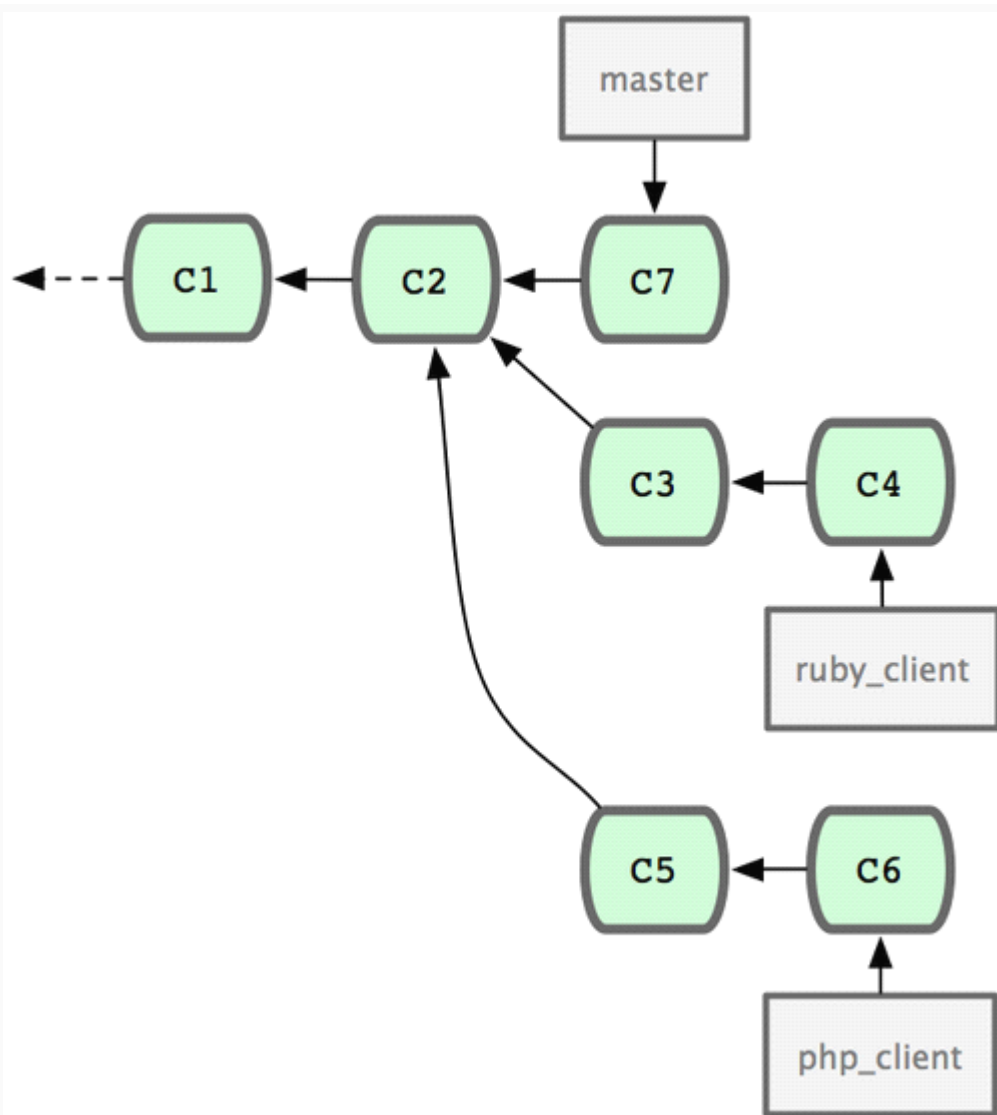


图 5-19. 多个特性分支

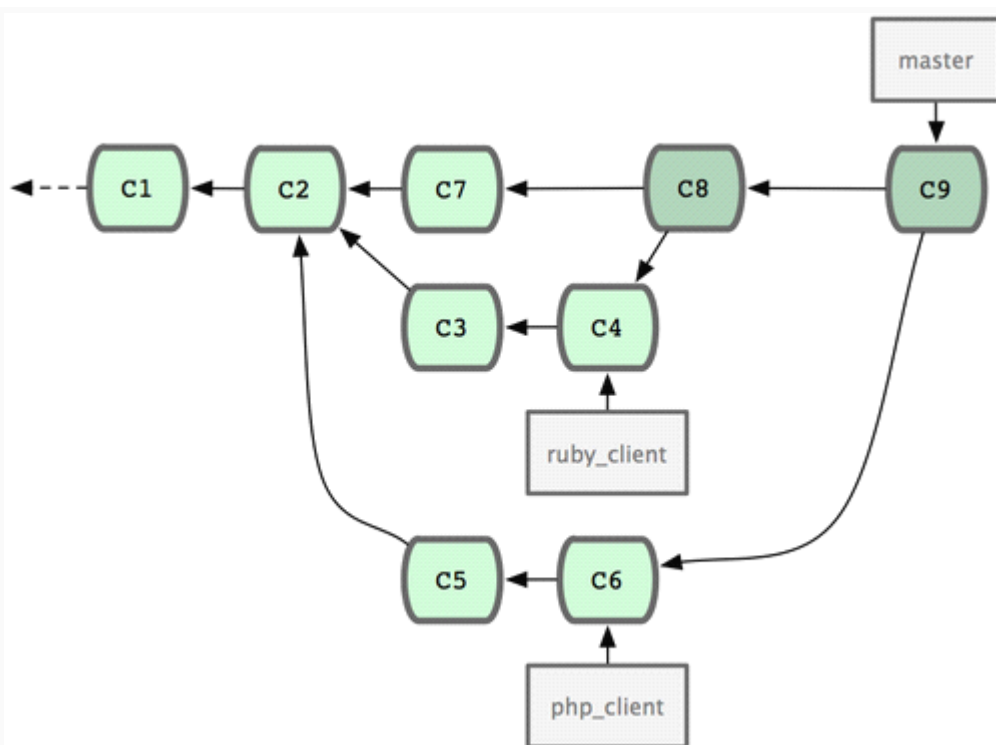


图 5-20. 合并特性分支之后这是最简单的流程，所以在处理大一些的项目时可能会有问题。

对于大型项目，至少需要维护两个长期分支 `master` 和 `develop`。新代码（图 5-21 中的 `ruby_client`）将首先并入 `develop` 分支（图 5-22 中的 C8），经过一个阶段，确认 `develop` 中的代码已稳定到可发行时，再将 `master` 分支快进到稳定点（图 5-23 中的 C8）。而平时这两个分支都会被推送到公开的代码库。

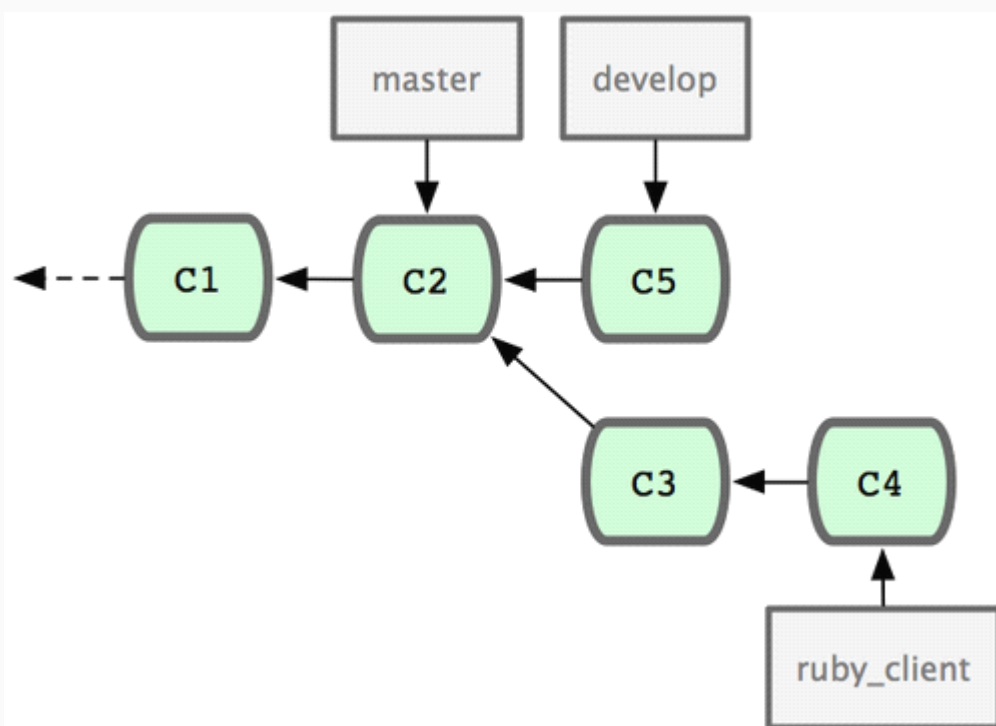


图 5-21. 特性分支合并前

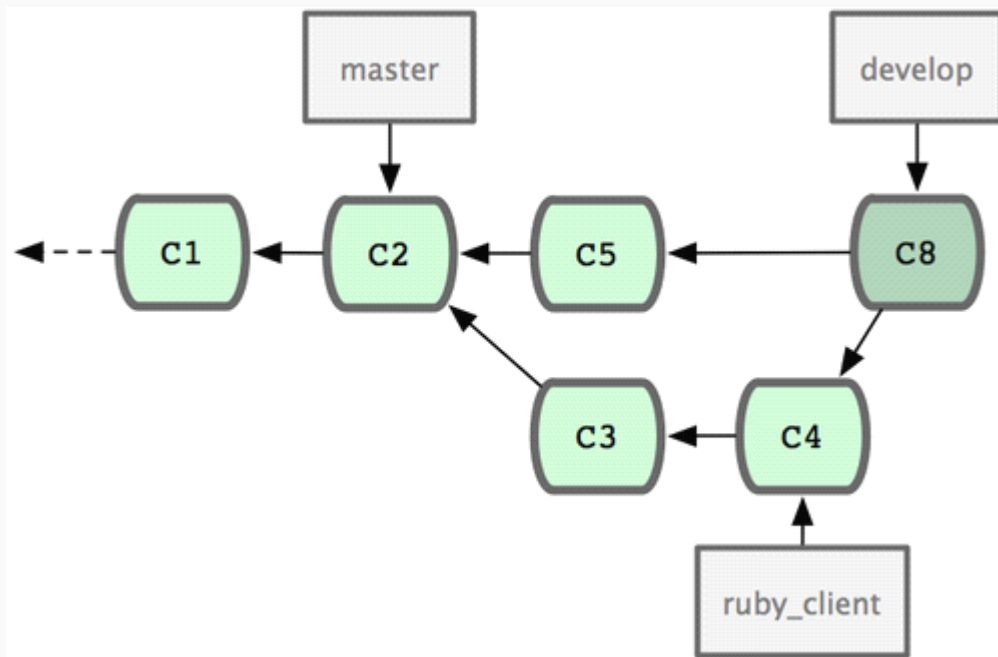


图 5-22. 特性分支合并后

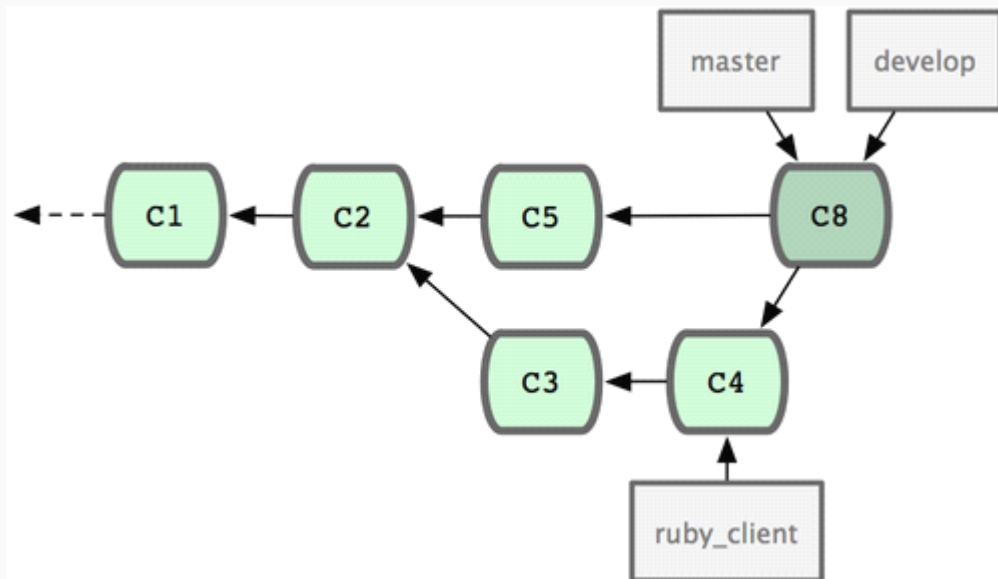


图 5-23. 特性分支发布后这样，在人们克隆仓库时就有两种选择：既可检出最新稳定版本，确保正常使用；也能检出开发版本，试用最前沿的新特性。你也可以扩展这个概念，先将所有新代码合并到临时特性分支，等到该分支稳定下来并通过测试后，再并入 **develop** 分支。然后，让时间检验一切，如果这些代码确实可以正常工作相当长一段时间，那就有理由相信它已经足够稳定，可以放心并入主干分支发布。

大项目的合并流程

Git 项目本身有四个长期分支：用于发布的 `master` 分支、用于合并基本稳定特性的 `next` 分支、用于合并仍需改进特性的 `pu` 分支（`pu` 是 `proposed updates` 的缩写），以及用于除错维护的 `maint` 分支（`maint` 取自 `maintenance`）。维护者可以按照之前介绍的方法，将贡献者的代码引入为不同的特性分支（如图 5-24 所示），然后测试评估，看哪些特性能稳定工作，哪些还需改进。稳定的特性可以并入 `next` 分支，然后再推送到公共仓库，以供其他人试用。

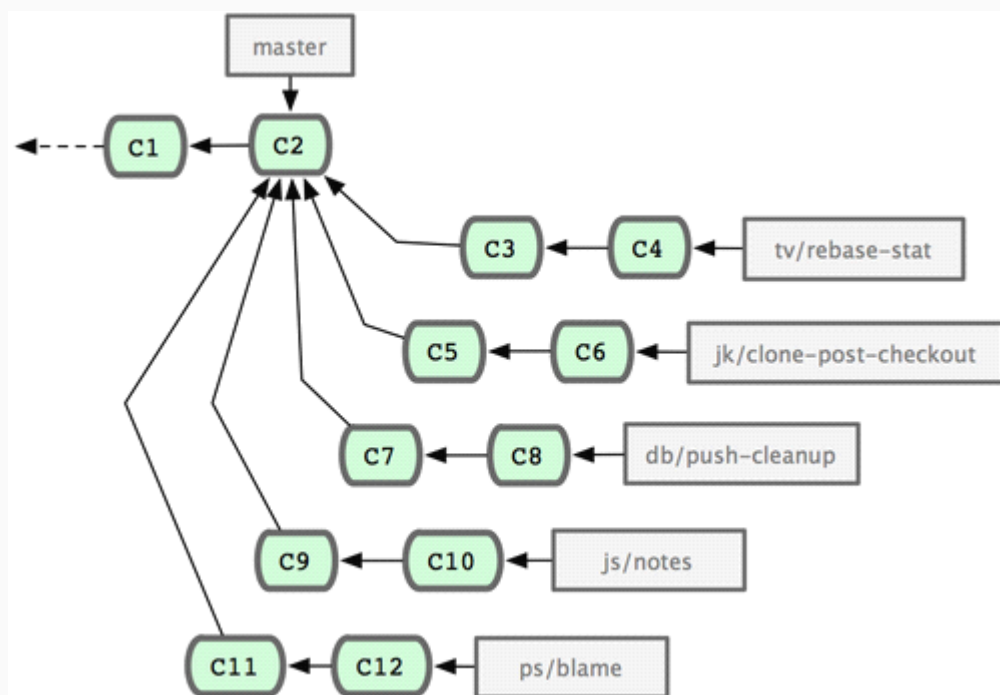


图 5-24. 管理复杂的并行贡献仍需改进的特性可以先并入 `pu` 分支。直到它们完全稳定后再并入 `master`。同时一并检查下 `next` 分支，将足够稳定的特性也并入 `master`。所以一般来说，`master` 始终是在快进，`next` 偶尔做下衍合，而 `pu` 则是频繁衍合，如图 5-25 所示：

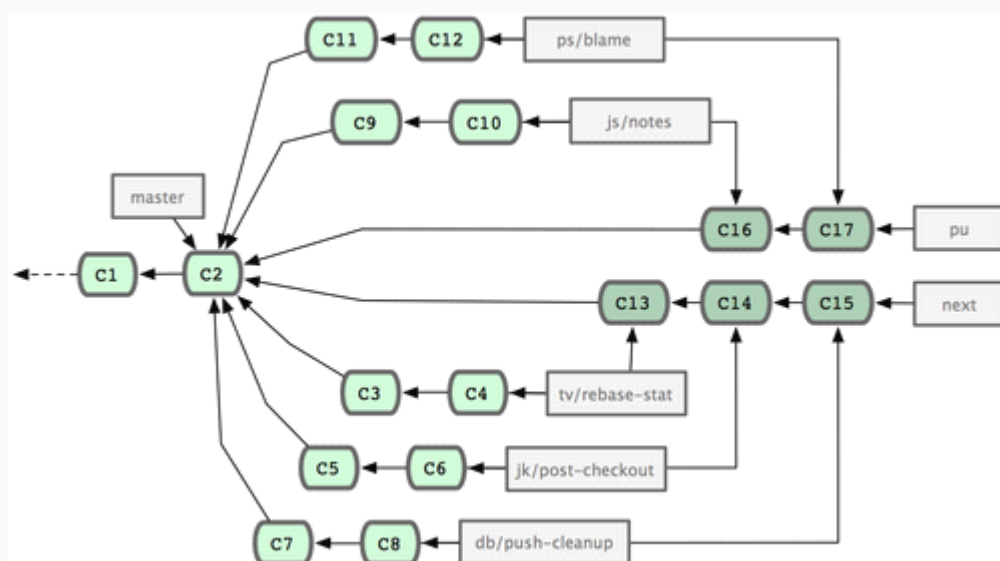


图 5-25. 将特性并入长期分支并入 **master** 后的特性分支，已经无需保留分支索引，放心删除好了。**Git** 项目还有一个 **maint** 分支，它是以最近一次发行版为基础分化而来的，用于维护除错补丁。所以克隆 **Git** 项目仓库后会得到这四个分支，通过检出不同分支可以了解各自进展，或是试用前沿特性，或是贡献代码。而维护者则通过管理这些分支，逐步有序地并入第三方贡献。

衍合与挑拣（cherry-pick）的流程

一些维护者更喜欢衍合或者挑拣贡献者的代码，而不是简单的合并，因为这样能够保持线性的提交历史。如果你完成了一个特性的开发，并决定将它引入到主干代码中，你可以转到那个特性分支然后执行衍合命令，好在你主干分支上（也可能是 **develop** 分支之类的）重新提交这些修改。如果这些代码工作得很好，你就可以快进 **master** 分支，得到一个线性的提交历史。

另一个引入代码的方法是挑拣。挑拣类似于针对某次特定提交的衍合。它首先提取某次提交的补丁，然后试着应用在当前分支上。如果某个特性分支上有多个 **commits**，但你想引入其中之一就可以使用这种方法。也可能仅仅是因为你喜欢用挑拣，讨厌衍合。假设你有一个类似图 5-26 的工程。

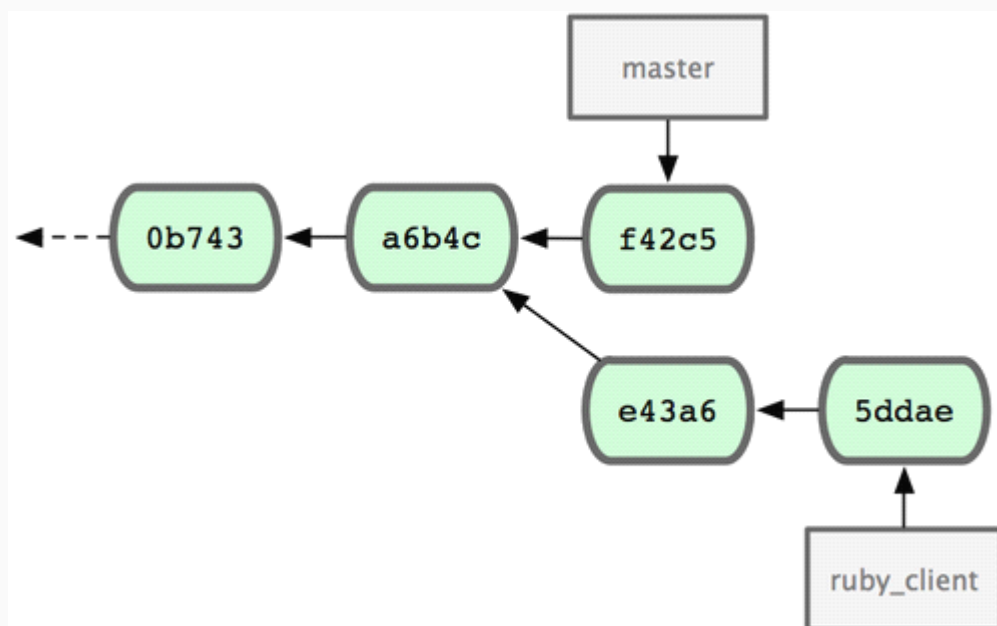


图 5-26. 挑拣（cherry-pick）之前的历史如果你希望拉取 **e43a6**到你的主干分支，可以这样：

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
```

```
3 files changed, 17 insertions(+), 3 deletions(-)
```

这将会引入 **e43a6** 的代码，但是会得到不同的 SHA-1 值，因为应用日期不同。现在你的历史看起来像图 5-27。

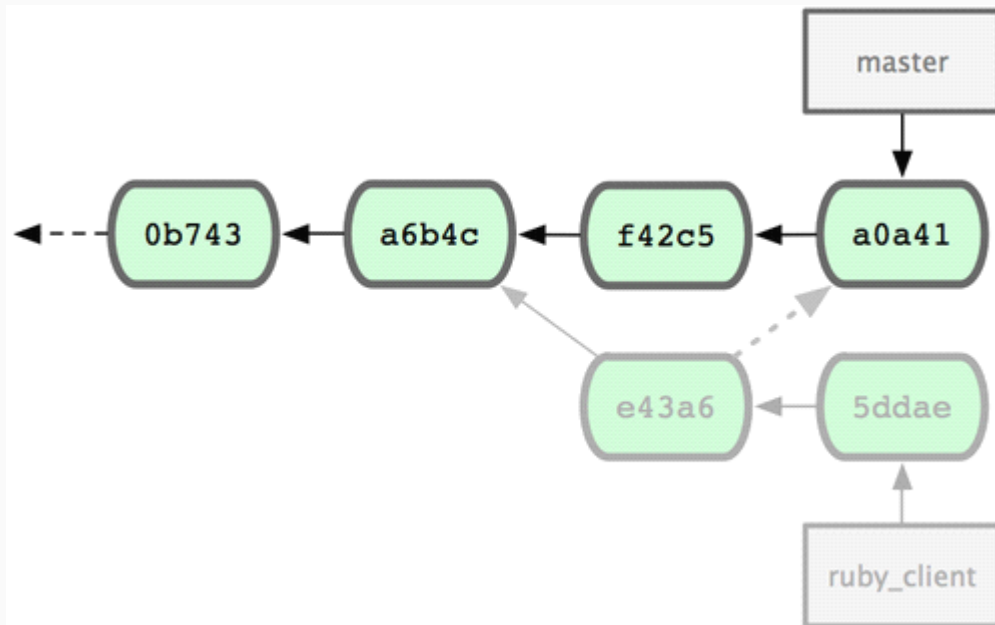


图 5-27. 挑拣（cherry-pick）之后的历史现在，你可以删除这个特性分支并丢弃你不想引入的那些 commit。

给发行版签名

你可以删除上次发布的版本并重新打标签，也可以像第二章所说的那样建立一个新的标签。如果你决定以维护者的身份给发行版签名，应该这样做：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon"
```

```
" 1024-bit DSA key, ID F721C45A, created 2009-02-09"
```

完成签名之后，如何分发 PGP 公钥（public key）是个问题。（译者注：分发公钥是为了验证标签）。还好，Git 的设计者想到了解决办法：可以把 key（既公钥）作为 blob 变量写入 Git 库，然后把它的内容直接写在标签里。gpg --list-keys 命令可以显示出你所拥有的 key：

```
$ gpg --list-keys
```

```
/Users/schacon/.gnupg/pubring.gpg
-----
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid Scott Chacon
```

```
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

然后，导出 **key** 的内容并经由管道符传递给 **git hash-object**，之后钥匙会以 **blob** 类型写入 **Git** 中，最后返回这个 **blob** 量的 **SHA-1** 值：

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

现在你的 **Git** 已经包含了这个 **key** 的内容了，可以通过不同的 **SHA-1** 值指定不同的 **key** 来创建标签。

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

在运行 **git push --tags** 命令之后，**maintainer-gpg-pub** 标签就会公布给所有人。如果有人想要校验标签，他可以使用如下命令导入你的 **key**：

```
$ git show maintainer-gpg-pub | gpg --import
```

人们可以用这个 **key** 校验你签名的所有标签。另外，你也可以在标签信息里写入一个操作向导，用户只需要运行 **git show** 查看标签信息，然后按照你的向导就能完成校验。

生成内部版本号

因为 **Git** 不会为每次提交自动附加类似 **v123** 的递增序列，所以如果你想要得到一个便于理解的提交号可以运行 **git describe** 命令。**Git** 将会返回一个字符串，由三部分组成：最近一次标定的版本号，加上自那次标定之后的提交次数，再加上一段 **SHA-1** 值 of the commit you're describing:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

这个字符串可以作为快照的名字，方便人们理解。如果你的 **Git** 是你自己下载源码然后编译安装的，你会发现 **git --version** 命令的输出和这个字符串差不多。如果在一个刚刚打完标签的提交上运行 **describe** 命令，只会得到这次标定的版本号，而没有后面两项信息。

git describe 命令只适用于有标注的标签（通过 **-a** 或者 **-s** 选项创建的标签），所以发行版的标签都应该是带有标注的，以保证 **git describe** 能够正确的执行。你也可以把这个字符串作

为 `checkout` 或者 `show` 命令的目标，因为他们最终都依赖于一个简短的 SHA-1 值，当然如果这个 SHA-1 值失效他们也跟着失效。最近 Linux 内核为了保证 SHA-1 值的唯一性，将位数由 8 位扩展到 10 位，这就导致扩展之前的 `git describe` 输出完全失效了。

准备发布

现在可以发布一个新的版本了。首先要将代码的压缩包归档，方便那些可怜的还没有使用 Git 的人们。可以使用 `git archive`：

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

这个压缩包解压出来的是一个文件夹，里面是你项目的最新代码快照。你也可以用类似的方法建立一个 zip 压缩包，在 `git archive` 加上 `--format=zip` 选项：

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

现在你有了一个 `tar.gz` 压缩包和一个 `zip` 压缩包，可以把他们上传到你网站上或者用 `e-mail` 发给别人。

制作简报

是时候通知邮件列表里的朋友们来检验你的成果了。使用 `git shortlog` 命令可以方便快捷的制作一份修改日志（`changelog`），告诉大家上次发布之后又增加了哪些特性和修复了哪些 `bug`。实际上这个命令能够统计给定范围内的所有提交；假如你上一次发布的版本是 `v1.0.1`，下面的命令将给出自从上次发布之后的所有提交的简介：

```
$ git shortlog --no-merges master --not v1.0.1
```

Chris Wanstrath (8):

- Add support for annotated tags to Grit::Tag
- Add packed-refs annotated tag support.
- Add Grit::Commit#to_patch
- Update version and History.txt
- Remove stray `puts`
- Make ls_tree ignore nils

Tom Preston-Werner (4):

- fix dates in history
- dynamic version method
- Version bump to 1.0.2
- Regenerated gemspec for version 1.0.2

这就是自从 v1.0.1 版本以来的所有提交的简介，内容按照作者分组，以便你能快速的发 e-mail 给他们。

5.4 小结

你学会了如何使用 **Git** 为项目做贡献，也学会了如何使用 **Git** 维护你的项目。恭喜！你已经成为一名高效的开发者。在下一篇你将学到更强大的工具来处理更加复杂的问题，之后你会变成一位 **Git** 大师。

Git 工具

现在，你已经学习了管理或者维护 Git 仓库，实现代码控制所需的大多数日常命令和工作流程。你已经完成了跟踪和提交文件的基本任务，并且发挥了暂存区和轻量级的特性分支及合并的威力。

接下来你将领略到一些 Git 可以实现的非常强大的功能，这些功能你可能并不会在日常操作中使用，但在某些时候你也许会需要。

6.1 修订版本（Revision）选择

Git 允许你通过几种方法来指明特定的或者一定范围内的提交。了解它们并不是必需的，但是了解一下总没坏处。

单个修订版本

显然你可以使用给出的 SHA-1 值来指明一次提交，不过也有更加人性化的方法来做同样的事。本节概述了指明单个提交的诸多方法。

简短的 SHA

Git 很聪明，它能够通过你提供的前几个字符来识别你想要的那次提交，只要你提供的那部分 SHA-1 不短于四个字符，并且没有歧义——也就是说，当前仓库中只有一个对象以这段 SHA-1 开头。

例如，想要查看一次指定的提交，假设你运行 `git log` 命令并找到你增加了功能的那次提交：

```
$ git log

commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

```
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

假设是 `1c002dd....`。如果你想 `git show` 这次提交，下面的命令是等价的（假设简短的版本没有歧义）：

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git 可以为你的 **SHA-1** 值生成出简短且唯一的缩写。如果你传递 `--abbrev-commit` 给 `git log` 命令，输出结果里就会使用简短且唯一的值；它默认使用七个字符来表示，不过必要时为了避免 **SHA-1** 的歧义，会增加字符数：

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

通常在一个项目中，使用八到十个字符来避免 **SHA-1** 歧义已经足够了。最大的 Git 项目之一，Linux 内核，目前也只需要最长 40 个字符中的 12 个字符来保持唯一性。

关于 **SHA-1** 的简短说明

许多人可能会担心一个问题：在随机的偶然情况下，在他们的仓库里会出现两个具有相同 **SHA-1** 值的对象。那会怎么样呢？

如果你真的向仓库里提交了一个跟之前的某个对象具有相同 **SHA-1** 值的对象，Git 将会发现之前的那个对象已经存在在 Git 数据库中，并认为它已经被写入了。如果什么时候你想再次检出那个对象时，你会总是得到先前的那个对象的数据。

不过，你应该了解到，这种情况发生的概率是多么微小。**SHA-1** 摘要长度是 20 字节，也就是 160 位。为了保证有 50% 的概率出现一次冲突，需要 2^{80} 个随机哈希的对象（计算冲突机率的公式是 $p = (n(n-1)/2) * (1/2^{160})$ ）。 2^{80} 是 1.2×10^{24} ，也就是一亿亿亿，那是地球上沙粒总数的 1200 倍。

现在举例说一下怎样才能产生一次 **SHA-1** 冲突。如果地球上 65 亿的人类都在编程，每人每秒都在产生等价于整个 Linux 内核历史（一百万个 Git 对象）的代码，并将之提交到一个巨大的 Git 仓库里面，那将花费 5 年的时间才会产生足够的对象，使其拥有 50% 的概率产生一次 **SHA-1** 对象冲突。这要比你编程团队的成员同一个晚上在互不相干的意外中

被狼袭击并杀死的机率还要小。

分支引用

指明一次提交的最直接的方法要求有一个指向它的分支引用。这样，你就可以在任何需要一个提交对象或者 **SHA-1** 值的 **Git** 命令中使用该分支名称了。如果你想要显示一个分支的最后一次提交的对象，例如假设 **topic1** 分支指向 **ca82a6d**，那么下面的命令是等价的：

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

如果你想知道某个分支指向哪个特定的 **SHA**，或者想看任何一个例子中被简写的 **SHA-1**，你可以使用一个叫做 **rev-parse** 的 **Git** 探测工具。在第 9 章你可以看到关于探测工具的更多信息；简单来说，**rev-parse** 是为了底层操作而不是日常操作设计的。不过，有时你想看 **Git** 现在到底处于什么状态时，它可能会很有用。这里你可以对你的分支运行 **rev-parse**。

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

引用日志里的简称

在你工作的同时，**Git** 在后台的工作之一就是保存一份引用日志——一份记录最近几个月你的 **HEAD** 和分支引用的日志。

你可以使用 **git reflog** 来查看引用日志：

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd... HEAD@{2}: commit: added some blame and merge stuff
1c36188... HEAD@{3}: rebase -i (squash): updating HEAD
95df984... HEAD@{4}: commit: # This is a combination of two commits.
1c36188... HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5... HEAD@{6}: rebase -i (pick): updating HEAD
```

每次你的分支顶端因为某些原因被修改时，**Git** 就会为你将信息保存在这个临时历史记录里面。你也可以使用这份数据来指明更早的分支。如果你想查看仓库中 **HEAD** 在五次前的值，你可以使用引用日志的输出中的 **@{n}** 引用：

```
$ git show HEAD@{5}
```

你也可以使用这个语法来查看一定时间前分支指向哪里。例如，想看你的 **master** 分支昨天在哪，你可以输入

```
$ git show master@{yesterday}
```


它就会显示昨天分支的顶端在哪。这项技术只对还在你引用日志里的数据有用，所以不能用来查看比几个月前还早的提交。

想要看类似于 `git log` 输出格式的引用日志信息，你可以运行 `git log -g`：

```
$ git log -g master

commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

需要注意的是，日志引用信息只存在于本地——这是一个你在仓库里做过什么的日志。其他人的仓库拷贝里的引用和你的相同；而你新克隆一个仓库的时候，引用日志是空的，因为你在仓库里还没有操作。只有你克隆了一个项目至少两个月，`git show HEAD@{2.months.ago}` 才会有用——如果你是五分钟前克隆的仓库，将不会有结果返回。

祖先引用

另一种指明某次提交的常用方法是通过它的祖先。如果你在引用最后加上一个 `^`，Git 将其理解为此次提交的父提交。假设你的工程历史是这样的：

```
$ git log --pretty=format:'%h %s' --graph

* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
```

```
* 9b29157 add open3_detach to gemspec file list
```

那么，想看上一次提交，你可以使用 **HEAD^**，意思是“HEAD 的父提交”：

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

你也可以在 **^** 后添加一个数字——例如，**d921970^2** 意思是“d921970 的第二父提交”。这种语法只在合并提交时有用，因为合并提交可能有多个父提交。第一父提交是你合并时所在分支，而第二父提交是你所合并的分支：

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
```

```
Some rdoc changes
```

另外一个指明祖先提交的方法是 **~**。这也是指向第一父提交，所以 **HEAD~** 和 **HEAD^** 是等价的。当你指定数字的时候就明显不一样了。**HEAD~2** 是指“第一父提交的第一父提交”，也就是“祖父提交”——它会根据你指定的次数检索第一父提交。例如，在上面列出的历史记录里面，**HEAD~3** 会是

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

也可以写成 **HEAD^^**，同样是第一父提交的第一父提交的第一父提交：

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

你也可以混合使用这些语法——你可以通过 **HEAD~3^2** 指明先前引用的第二父提交（假设它是一个合并提交）。

提交范围

现在你已经可以指明单次的提交，让我们来看看怎样指明一定范围的提交。这在你管理分支的时候尤显重要——如果你有很多分支，你可以指明范围来圈定一些问题的答案，比如：“这个分支上我有哪些工作还没合并到主分支的？”

双点

最常用的指明范围的方法是双点的语法。这种语法主要是让 **Git** 区分出可从一个分支中获得而不能从另一个分支中获得的提交。例如，假设你有类似于图 6-1 的提交历史。

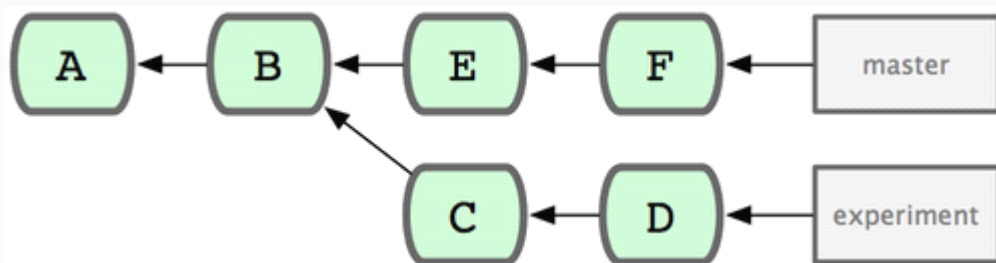


图 6-1. 范围选择的提交历史实例你想要查看你的试验分支上哪些没有被提交到主分支，那么你就可以使用 **master..experiment** 来让 **Git** 显示这些提交的日志——这句话的意思是“所有可从 **experiment** 分支中获得而不能从 **master** 分支中获得的提交”。为了使例子简单明了，我使用了图标中提交对象的字母来代替真实日志的输出，所以会显示：

```
$ git log master..experiment
```

```
D
```

```
C
```

另一方面，如果你想看相反的——所有在 **master** 而不在 **experiment** 中的分支——你可以

交换分支的名字。`experiment..master` 显示所有可在 `master` 获得而在 `experiment` 中不能的提交：

```
$ git log experiment..master
```

```
F
```

```
E
```

这在你想保持 `experiment` 分支最新和预览你将合并的提交的时候特别有用。这个语法的另一种常见用途是查看你将把什么推送到远程：

```
$ git log origin/master..HEAD
```

这条命令显示任何在你当前分支上而不在远程 `origin` 上的提交。如果你运行 `git push` 并且你的当前分支正在跟踪 `origin/master`，被 `git log origin/master..HEAD` 列出的提交就是将被传输到服务器上的提交。你也可以留空语法中的一边来让 `Git` 来假定它是 `HEAD`。例如，输入 `git log origin/master..` 将得到和上面的例子一样的结果——`Git` 使用 `HEAD` 来代替不存在的一边。

多点

双点语法就像速记一样有用；但是你也可能会想针对两个以上的分支来指明修订版本，比如查看哪些提交被包含在某些分支中的一个，但是不在你当前的分支上。`Git` 允许你在引用前使用 `^` 字符或者 `--not` 指明你不希望提交被包含其中的分支。因此下面三个命令是等同的：

```
$ git log refA..refB
```

```
$ git log ^refA refB
```

```
$ git log refB --not refA
```

这样很好，因为它允许你在查询中指定多于两个的引用，而这是双点语法所做不到的。例如，如果你想查找所有从 `refA` 或 `refB` 包含的但是不被 `refC` 包含的提交，你可以输入下面中的一个

```
$ git log refA refB ^refC
```

```
$ git log refA refB --not refC
```

这建立了一个非常强大的修订版本查询系统，应该可以帮助你解决分支里包含了什么这个问题。

三点

最后一种主要的范围选择语法是三点语法，这个可以指定被两个引用中的一个包含但又不被两者同时包含的分支。回过头来看一下图6-1里所列的提交历史的例子。如果你想查看 `master` 或者 `experiment` 中包含的但不是两者共有的引用，你可以运行

```
$ git log master...experiment
```

F
E
D
C

这个再次给出你普通的 **log** 输出但是只显示那四次提交的信息，按照传统的提交日期排列。这种情形下，**log** 命令的一个常用参数是 **--left-right**，它会显示每个提交到底处于哪一侧的分支。这使得数据更加有用。

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

有了以上工具，让 **Git** 知道你要察看哪些提交就容易得多了。

6.2 交互式暂存

Git 提供了很多脚本来辅助某些命令行任务。这里，你将看到一些交互式命令，它们帮助你方便地构建只包含特定组合和部分文件的提交。在你修改了一大批文件然后决定将这些变更分布在几个各有侧重的提交而不是单个又大又乱的提交时，这些工具非常有用。用这种方法，你可以确保你的提交在逻辑上划分为相应的变更集，以便于供和你一起工作的开发者审阅。如果你运行 **git add** 时加上 **-i** 或者 **--interactive** 选项，**Git** 就进入了一个交互式的 **shell** 模式，显示一些类似于下面的信息：

```
$ git add -i

          staged      unstaged path
1:    unchanged      +0/-1  TODO
2:    unchanged      +1/-1  index.html
3:    unchanged      +5/-1  lib/simplegit.rb

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

What now>
```

你会看到这个命令以一个完全不同的视图显示了你的暂存区——主要是你通过 **git status** 得到的那些信息但是稍微简洁但信息更加丰富一些。它在左侧列出了你暂存的变更，在右侧列

出了未被暂存的变更。

在这之后是一个命令区。这里你可以做很多事情，包括暂存文件，撤回文件，暂存部分文件，加入未被追踪的文件，查看暂存文件的差别。

暂存和撤回文件

如果你在 **What now>** 的提示后输入 **2** 或者 **u**，这个脚本会提示你那些文件你想要暂存：

```
What now> 2

      staged      unstaged path
1:   unchanged    +0/-1  TODO
2:   unchanged    +1/-1  index.html
3:   unchanged    +5/-1  lib/simplegit.rb
```

Update>>

如果想暂存 **TODO** 和 **index.html**，你可以输入相应的编号：

```
Update>> 1,2

      staged      unstaged path
* 1:   unchanged    +0/-1  TODO
* 2:   unchanged    +1/-1  index.html
  3:   unchanged    +5/-1  lib/simplegit.rb
```

Update>>

每个文件旁边的*表示选中的文件将被暂存。如果你在 **update>>**提示后直接敲入回车，Git 会替你把所有选中的内容暂存：

Update>>

updated 2 paths

*** Commands ***

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

What now> 1

```
      staged      unstaged path
1:      +0/-1      nothing  TODO
2:      +1/-1      nothing  index.html
3:   unchanged    +5/-1  lib/simplegit.rb
```

现在你可以看到 **TODO** 和 **index.html** 文件被暂存了同时 **simplegit.rb** 文件仍然未被暂存。如果这时你想要撤回 **TODO** 文件，就使用 **3** 或者 **r**（代表 **revert**，恢复）选项：

```

*** Commands ***

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

What now> 3

      staged      unstaged path
1:      +0/-1      nothing TODO
2:      +1/-1      nothing index.html
3:   unchanged      +5/-1 lib/simplegit.rb

Revert>> 1

      staged      unstaged path
* 1:      +0/-1      nothing TODO
2:      +1/-1      nothing index.html
3:   unchanged      +5/-1 lib/simplegit.rb

Revert>> [enter]
reverted one path

```

再次查看 **Git** 的状态，你会看到你已经撤回了 **TODO** 文件

```

*** Commands ***

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

What now> 1

      staged      unstaged path
1:   unchanged      +0/-1 TODO
2:      +1/-1      nothing index.html
3:   unchanged      +5/-1 lib/simplegit.rb

```

要查看你暂存内容的差异，你可以使用 **6** 或者 **d**（表示 **diff**）命令。它会显示你暂存文件的列表，你可以选择其中的几个，显示其被暂存的差异。这跟你在命令行下指定 **git diff --cached** 非常相似：

```

*** Commands ***

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

What now> 6

      staged      unstaged path
1:      +1/-1      nothing index.html

Review diff>> 1

diff --git a/index.html b/index.html

```

```

index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

通过这些基本命令，你可以使用交互式增加模式更加方便地处理暂存区。

暂存补丁

只让 **Git** 暂存文件的某些部分而忽略其他也是有可能的。例如，你对 **simplegit.rb** 文件作了两处修改但是只想暂存其中一个而忽略另一个，在 **Git** 中实现这一点非常容易。在交互式的提示符下，输入**5**或者**p**（表示 **patch**，补丁）。**Git** 会询问哪些文件你希望部分暂存；然后对于被选中文件的每一节，他会逐个显示文件的差异区块并询问你是否希望暂存他们：

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-   command("git log -n 25 #{treeish}")
+   command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,]?

```

此处你有很多选择。输入**?**可以显示列表：

```

Stage this hunk [y,n,a,d,/,j,J,g,e,]? ?
y - stage this hunk

```



```

n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help

```

如果你想暂存各个区块，通常你会输入 **y** 或者 **n**，但是暂存特定文件里的全部区块或者暂时跳过对一个区块的处理同样也很有用。如果你暂存了文件的一个部分而保留另外一个部分不被暂存，你的状态输出看起来会是这样：

```

What now> 1

      staged      unstaged path
1:    unchanged      +0/-1 TODO
2:        +1/-1      nothing index.html
3:        +1/-1      +4/-0 lib/simplegit.rb

```

`simplegit.rb` 的状态非常有意思。它显示有几行被暂存了，有几行没有。你部分地暂存了这个文件。在这时，你可以退出交互式脚本然后运行 **git commit** 来提交部分暂存的文件。

最后你也可以不通过交互式增加的模式来实现部分文件暂存——你可以在命令行下通过 **git add -p** 或者 **git add --patch** 来启动同样的脚本。

6.3 储藏 (Stashing)

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是 **git stash** 命令。

“储藏”“可以获取你工作目录的中间状态——也就是你修改过的被追踪的文件和暂存的变更——并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

储藏你的工作

为了演示这一功能，你可以进入你的项目，在一些文件上进行工作，有可能还暂存其中一个

变更。如果你运行 **git status**，你可以看到你的中间状态：

```
$ 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:   lib/simplegit.rb
#
```

现在你想切换分支，但是你还不想提交你正在进行中的工作；所以你储藏这些变更。为了往堆栈推送一个新的储藏，只要运行 **git stash**：

```
$ git stash

Saved working directory and index state \
    "WIP on master: 049d078 added the index file"

HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

你的工作目录就干净了：

```
$ git status

# On branch master

nothing to commit (working directory clean)
```

这时，你可以方便地切换到其他分支工作；你的变更都保存在栈上。要查看现有的储藏，你可以使用 **git stash list**：

```
$ git stash list

stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

在这个案例中，之前已经进行了两次储藏，所以你可以访问到三个不同的储藏。你可以重新应用你刚刚实施的储藏，所采用的命令就是之前在原始的 **stash** 命令的帮助输出里提示的：**git stash apply**。如果你想应用更早的储藏，你可以通过名字指定它，像这样：**git stash apply stash@{2}**。如果你不指明，Git 默认使用最近的储藏并尝试应用它：

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

你可以看到 **Git** 重新修改了你所储藏的那些当时尚未提交的文件。在这个案例里，你尝试应用储藏的工作目录是干净的，并且属于同一分支；但是一个干净的工作目录和应用到相同的分支上并不是应用储藏的必要条件。你可以在其中一个分支上保留一份储藏，随后切换到另外一个分支，再重新应用这些变更。在工作目录里包含已修改和未提交的文件时，你也可以应用储藏——**Git** 会给出归并冲突如果有任何变更无法干净地被应用。

对文件的变更被重新应用，但是被暂存的文件没有重新被暂存。想那样的话，你必须在运行 **git stash apply** 命令时带上一个 **--index** 的选项来告诉命令重新应用被暂存的变更。如果你这么做的，你应该已经回到你原来的位置：

```
$ git stash apply --index
# 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:   lib/simplegit.rb
#
```

apply 选项只尝试应用储藏的工作——储藏的内容仍然在栈上。要移除它，你可以运行 **git stash drop**，加上你希望移除的储藏的名字：

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
$ git stash drop stash@{0}
```

```
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

你也可以运行 `git stash pop` 来重新应用储藏，同时立刻将其从堆栈中移走。

Un-applying a Stash

In some use case scenarios you might want to apply stashed changes, do some work, but then un-apply those changes that originally came from the stash. Git does not provide such a `stash unapply` command, but it is possible to achieve the effect by simply retrieving the patch associated with a stash and applying it in reverse:

```
$ git stash show -p stash@{0} | git apply -R
```

Again, if you don't specify a stash, Git assumes the most recent stash:

```
$ git stash show -p | git apply -R
```

You may want to create an alias and effectively add a `stash-unapply` command to your git. For example:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

从储藏中创建分支

如果你储藏了一些工作，暂时不去理会，然后继续在你储藏工作的分支上工作，你在重新应用工作时可能会碰到一些问题。如果尝试应用的变更是针对一个你那之后修改过的文件，你会碰到一个归并冲突并且必须去化解它。如果你想用更方便的方法来重新检验你储藏的变更，你可以运行 `git stash branch`，这会创建一个新的分支，检出你储藏工作时的所处的提交，重新应用你的工作，如果成功，将会丢弃储藏。

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# 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:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

这是一个很棒的捷径来恢复储藏的工作然后在新的分支上继续当时的工作。

6.4 重写历史

很多时候，在 **Git** 上工作的时候，你也许会由于某种原因想要修订你的提交历史。**Git** 的一个卓越之处就是它允许你在最后可能的时刻再作决定。你可以在你即将提交暂存区时决定什么文件归入哪一次提交，你可以使用 **stash** 命令来决定你暂时搁置的工作，你可以重写已经发生的提交以使它们看起来是另外一种样子。这个包括改变提交的次序、改变说明或者修改提交中包含的文件，将提交归并、拆分或者完全删除——这一切在你尚未开始将你的工作和别人共享前都是可以的。

在这一节中，你会学到如何完成这些很有用的任务以使你的提交历史在你将其共享给别人之前变成你想要的样子。

改变最近一次提交

改变最近一次提交也许是最常见的重写历史的行为。对于你的最近一次提交，你经常想做两件基本事情：改变提交说明，或者改变你刚刚通过增加，改变，删除而记录的快照。

如果你只想修改最近一次提交说明，这非常简单：

```
$ git commit --amend
```

这会把你带入文本编辑器，里面包含了你最近一次提交说明，供你修改。当你保存并退出编辑器，这个编辑器会写入一个新的提交，里面包含了那个说明，并且让它成为你的新的最近一次提交。

如果你完成提交后又想修改被提交的快照，增加或者修改其中的文件，可能因为你最初提交时，忘了添加一个新建的文件，这个过程基本上一样。你通过修改文件然后对其运行 **git add** 或对一个已被记录的文件运行 **git rm**，随后的 **git commit --amend** 会获取你当前的暂存区并将它作为新提交对应的快照。

使用这项技术的时候你必须小心，因为修正会改变提交的 **SHA-1** 值。这个很像是一次非常小的 **rebase**——不要在你最近一次提交被推送后还去修正它。

修改多个提交说明

要修改历史中更早的提交，你必须采用更复杂的工具。**Git** 没有一个修改历史的工具，但是你可以使用 **rebase** 工具来衍合一系列的提交到它们原来所在的 **HEAD** 上而不是移到新的

上。依靠这个交互式的 **rebase** 工具，你就可以停留在每一次提交后，如果你想修改或改变说明、增加文件或任何其他事情。你可以通过给 **git rebase** 增加 **-i** 选项来以交互方式地运行 **rebase**。你必须通过告诉命令衍合到哪次提交，来指明你需要重写的提交的回溯深度。

例如，你想修改最近三次的提交说明，或者其中任意一次，你必须给 **git rebase -i** 提供一个参数，指明你想要修改的提交的父提交，例如 **HEAD~2** 或者 **HEAD~3**。可能记住 **~3** 更加容易，因为你想修改最近三次提交；但是请记住你事实上所指的是四次提交之前，即你想修改的提交的父提交。

```
$ git rebase -i HEAD~3
```

再次提醒这是一个衍合命令——**HEAD~3..HEAD** 范围内的每一次提交都会被重写，无论你是否修改说明。不要涵盖你已经推送到中心服务器的提交——这么做会使其他开发者产生混乱，因为你提供了同样变更的不同版本。

运行这个命令会为你的文本编辑器提供一个提交列表，看起来像下面这样

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# 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.
#
```

很重要的一点是你得注意这些提交的顺序与你通常通过 **log** 命令看到的是相反的。如果你运行 **log**，你会看到下面这样的结果：

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

请注意这里的倒序。交互式的 **rebase** 给了你一个即将运行的脚本。它会从你在命令行上指明的提交开始(**HEAD~3**)然后自上至下重播每次提交里引入的变更。它将最早的列在顶上而

不是最近的，因为这是第一个需要重播的。

你需要修改这个脚本来让它停留在你想修改的变更上。要做到这一点，你只要将你想修改的每一次提交前面的 **pick** 改为 **edit**。例如，只想修改第三次提交说明的话，你就像下面这样修改文件：

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

当你保存并退出编辑器，**Git** 会倒回至列表中的最后一次提交，然后把你送到命令行中，同时显示以下信息：

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

这些指示很明确地告诉你该干什么。输入

```
$ git commit --amend
```

修改提交说明，退出编辑器。然后，运行

```
$ git rebase --continue
```

这个命令会自动应用其他两次提交，你就完成任务了。如果你将更多行的 **pick** 改为 **edit**，你就能对你想修改的提交重复这些步骤。**Git** 每次都会停下，让你修正提交，完成后继续运行。

重排提交

你也可以使用交互式的衍合来彻底重排或删除提交。如果你想删除“added cat-file”这个提交并且修改其他两次提交引入的顺序，你将 **rebase** 脚本从这个

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

改为这个：

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

当你保存并退出编辑器，Git 将分支倒回至这些提交的父提交，应用310154e，然后 f7f3f6d，接着停止。你有效地修改了这些提交的顺序并且彻底删除了”added cat-file”这次提交。

压制(Squashing)提交

交互式的衍合工具还可以将一系列提交压制为单一提交。脚本在 rebase 的信息里放了一些有用的指示：

```
#
# 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.
#
```

如果不用”pick”或者”edit”，而是指定”squash”，Git 会同时应用那个变更和它之前的变更并将提交说明归并。因此，如果你想将这三个提交合并为单一提交，你可以将脚本修改成这样：

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

当你保存并退出编辑器，Git 会应用全部三次变更然后将你送回编辑器来归并三次提交说明。

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:
```



```
added cat-file
```

当你保存之后，你就拥有了一个包含前三次提交的全部变更的单一提交。

拆分提交

拆分提交就是撤销一次提交，然后多次部分地暂存或提交直到结束。例如，假设你想将三次提交中的中间一次拆分。将”updated README formatting and added blame”拆分成两次提交：第一次为”updated README formatting”，第二次为”added blame”。你可以在 `rebase -i` 脚本中修改你想拆分的提交前的指令为”edit”：

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

然后，这个脚本就将你带入命令行，你重置那次提交，提取被重置的变更，从中创建多次提交。当你保存并退出编辑器，Git 倒回到列表中第一次提交的父提交，应用第一次提交（f7f3f6d），应用第二次提交（310154e），然后将你带到控制台。那里你可以用 `git reset HEAD^` 对那次提交进行一次混合的重置，这将撤销那次提交并且将修改的文件撤回。此时你可以暂存并提交文件，直到你拥有多次提交，结束后，运行 `git rebase --continue`。

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git 在脚本中应用了最后一次提交（a5f4a0d），你的历史看起来就像这样了：

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

再次提醒，这会修改你列表中的提交的 SHA 值，所以请确保这个列表里不包含你已经推送到共享仓库的提交。

核弹级选项: filter-branch

如果你想用脚本的方式修改大量的提交，还有一个重写历史的选项可以用——例如，全局性地修改电子邮件地址或者将一个文件从所有提交中删除。这个命令是 `filter-branch`，这个会

大面积地修改你的历史，所以你很有可能不该去用它，除非你的项目尚未公开，没有其他人正在你准备修改的提交的基础上工作。尽管如此，这个可以非常有用。你会学习一些常见用法，借此对它的功能有所认识。

从所有提交中删除一个文件

这个经常发生。有些人不经思考使用 `git add .`，意外地提交了一个巨大的二进制文件，你想将它从所有地方删除。也许你不小心提交了一个包含密码的文件，而你想让你的项目开源。`filter-branch` 大概会是你用来清理整个历史的工具。要从整个历史中删除一个名叫 `password.txt` 的文件，你可以在 `filter-branch` 上使用 `--tree-filter` 选项：

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` 选项会在每次检出项目时先执行指定的命令然后重新提交结果。在这个例子中，你会在所有快照中删除一个名叫 `password.txt` 的文件，无论它是否存在。如果你想删除所有不小心提交上去的编辑器备份文件，你可以运行类似 `git filter-branch --tree-filter 'rm -f *~' HEAD` 的命令。

你可以观察到 `Git` 重写目录树并且提交，然后将分支指针移到末尾。一个比较好的办法是在一个测试分支上做这些然后在你确定产物真的是你所要的之后，再 `hard-reset` 你的主分支。要在你所有的分支上运行 `filter-branch` 的话，你可以传递一个 `--all` 给命令。

将一个子目录设置为新的根目录

假设你完成了从另外一个代码控制系统的导入工作，得到了一些没有意义的子目录（`trunk`, `tags` 等等）。如果你想让 `trunk` 子目录成为每一次提交的新的项目根目录，`filter-branch` 也可以帮你做到：

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

现在你的项目根目录就是 `trunk` 子目录了。`Git` 会自动地删除不对这个子目录产生影响的提交。

全局性地更换电子邮件地址

另一个常见的案例是你在开始时忘了运行 `git config` 来设置你的姓名和电子邮件地址，也许你想开源一个项目，把你所有的工作电子邮件地址修改为个人地址。无论哪种情况你都可以用 `filter-branch` 来更换多次提交里的电子邮件地址。你必须小心一些，只改变属于你的电子邮件地址，所以你使用 `--commit-filter`：

```
$ git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
        GIT_AUTHOR_NAME="Scott Chacon";
        GIT_AUTHOR_EMAIL="schacon@example.com";
        git commit-tree "$@";
    else
        git commit-tree "$@";
    fi' HEAD
```

这个会遍历并重写所有提交使之拥有你的新地址。因为提交里包含了它们的父提交的SHA-1值，这个命令会修改你的历史中的所有提交，而不仅仅是包含了匹配的电子邮件地址的那些。

6.5 使用 Git 调试

Git 同样提供了一些工具来帮助你调试项目中遇到的问题。由于 Git 被设计为可应用于几乎任何类型的项目，这些工具是通用型，但是在遇到问题时可以经常帮助你查找缺陷所在。

文件标注

如果你在追踪代码中的缺陷想知道这是什么时候为什么被引进来的，文件标注会是你的最佳工具。它会显示文件中对每一行进行修改的最近一次提交。因此，如果你发现自己代码中的一个方法存在缺陷，你可以用 **git blame** 来标注文件，查看那个方法的每一行分别是由谁在哪一天修改的。下面这个例子使用了 **-L** 选项来限制输出范围在第12至22行：

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show
#{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log
#{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame
```

```
#{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

请注意第一个域里是最后一次修改该行的那次提交的 **SHA-1** 值。接下去的两个域是从那次提交中抽取的值——作者姓名和日期——所以你可以方便地获知谁在什么时候修改了这一行。在这后面是行号和文件的内容。请注意[^]**4832fe2**提交的那些行，这些指的是文件最初提交的那些行。那个提交是文件第一次被加入这个项目时存在的，自那以后未被修改过。这会带来小小的困惑，因为你已经至少看到了 **Git** 使用[^]来修饰一个提交的 **SHA** 值的三种不同的意义，但这里确实就是这个意思。

另一件很酷的事情是在 **Git** 中你不需要显式地记录文件的重命名。它会记录快照然后根据现实尝试找出隐式的重命名动作。这其中有一个很有意思的特性就是你可以让它找出所有的代码移动。如果你在 **git blame** 后加上 **-C**，**Git** 会分析你在标注的文件然后尝试找出其中代码片段的原始出处，如果它是从其他地方拷贝过来的话。最近，我在将一个名叫 **GITServerHandler.m** 的文件分解到多个文件中，其中一个是 **GITPackUpload.m**。通过对 **GITPackUpload.m** 执行带 **-C** 参数的 **blame** 命令，我可以看到代码块的原始出处：

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void)
gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMM
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMM
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

这真的非常有用。通常，你会把你拷贝代码的那次提交作为原始提交，因为这是你在这个文件中第一次接触到那儿行。**Git** 可以告诉你编写那些行的原始提交，即便是在另一个文件里。

二分查找

标注文件在你知道问题是哪里引入的时候会有帮助。如果你不知道，并且自上次代码可用的状态已经经历了上百次的提交，你可能就要求助于 **bisect** 命令了。**bisect** 会在你的提交历史

中进行二分查找来尽快地确定哪一次提交引入了错误。

例如你刚刚推送了一个代码发布版本到产品环境中，对代码为什么会表现成那样百思不得其解。你回到你的代码中，还好你可以重现那个问题，但是找不到在哪里。你可以对代码执行 **bisect** 来寻找。首先你运行 **git bisect start** 启动，然后你用 **git bisect bad** 来告诉系统当前的提交已经有问题了。然后你必须告诉 **bisect** 已知的最后一次正常状态是哪次提交，使用 **git bisect good [good_commit]**：

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccec5f9350d878ce677feb13d3b2] error handling on repo
```

Git 发现在你标记为正常的提交(**v1.0**)和当前的错误版本之间大约有**12**次提交，于是它检出中间的一个。在这里，你可以运行测试来检查问题是否存在于这次提交。如果是，那么它是在这个中间提交之前的某一次引入的；如果否，那么问题是在中间提交之后引入的。假设这里是没有错误的，那么你就通过 **git bisect good** 来告诉 **Git** 然后继续你的旅程：

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

现在你在另外一个提交上了，在你刚刚测试通过的和一個错误提交的中点处。你再次运行测试然后发现这次提交是错误的，因此你通过 **git bisect bad** 来告诉 **Git**：

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

这次提交是好的，那么 **Git** 就获得了确定问题引入位置所需的所有信息。它告诉你第一个错误提交的 **SHA-1** 值并且显示一些提交说明以及哪些文件在那次提交里修改过，这样你可以找出缺陷被引入的根源：

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800
```

```
secure this thing
```

```
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

当你完成之后，你应该运行 **git bisect reset** 来重设你的 **HEAD** 到你开始前的地方，否则你会处于一个诡异的地方：

```
$ git bisect reset
```

这是个强大的工具，可以帮助你检查上百的提交，在几分钟内找出缺陷引入的位置。事实上，如果你有一个脚本会在工程正常时返回0，错误时返回非0的话，你可以完全自动地执行 **git bisect**。首先你需要提供已知的错误和正确提交来告诉它二分查找的范围。你可以通过 **bisect start** 命令来列出它们，先列出已知的错误提交再列出已知的正确提交：

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

这样会自动地在每一个检出的提交里运行 **test-error.sh** 直到 **Git** 找出第一个破损的提交。你也可以运行像 **make** 或者 **make tests** 或者任何你所拥有的来为你执行自动化的测试。

6.6 子模块

经常有这样的事情，当你在一个项目上工作时，你需要在其中使用另外一个项目。也许它是一个第三方开发的库或者是你独立开发和并在多个父项目中使用的。这个场景下一个常见的问题产生了：你想将两个项目单独处理但是又需要在其中一个中使用另外一个。

这里有一个例子。假设你在开发一个网站，为之创建 **Atom** 源。你不想编写一个自己的 **Atom** 生成代码，而是决定使用一个库。你可能不得不像 **CPAN install** 或者 **Ruby gem** 一样包含来自共享库的代码，或者将代码拷贝到你的项目树中。如果采用包含库的办法，那么不管用什么办法都很难去定制这个库，部署它就更加困难了，因为你必须确保每个客户都拥有那个库。把代码包含到你自己的项目中带来的问题是，当上游被修改时，任何你进行的定制化的修改都很难归并。

Git 通过子模块处理这个问题。子模块允许你将一个 **Git** 仓库当作另外一个 **Git** 仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。

子模块初步

假设你想把 **Rack** 库（一个 **Ruby** 的 **web** 服务器网关接口）加入到你的项目中，可能既要保持你自己的变更，又要延续上游的变更。首先你要把外部的仓库克隆到你的子目录中。你通过 **git submodule add** 将外部项目加为子模块：

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
```

```
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

现在你就在项目里的 **rack** 子目录下有了一个 **Rack** 项目。你可以进入那个子目录，进行变更，加入你自己的远程可写仓库来推送你的变更，从原始仓库拉取和归并等等。如果你在加入子模块后立刻运行 **git status**，你会看到下面两项：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitmodules
#       new file:   rack
#
```

首先你注意到有一个 **.gitmodules** 文件。这是一个配置文件，保存了项目 **URL** 和你拉取到的本地子目录

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

如果你有多个子模块，这个文件里会有多个条目。很重要的一点是这个文件跟其他文件一样也是处于版本控制之下的，就像你的 **.gitignore** 文件一样。它跟项目里的其他文件一样可以被推送和拉取。这是其他克隆此项目的人获知子模块项目来源的途径。

git status 的输出里所列的另一项目是 **rack**。如果你运行在那上面运行 **git diff**，会发现一些有趣的东西：

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
```

```
+Subproject commit 08d709f78b8c5b0fbbeb7821e37fa53e69afcf433
```

尽管 **rack** 是你工作目录里的子目录，但 **Git** 把它视作一个子模块，当你不在那个目录里时并不记录它的内容。取而代之的是，**Git** 将它记录成来自那个仓库的一个特殊的提交。当你在那个子目录里修改并提交时，子项目会通知那里的 **HEAD** 已经发生变更并记录你当前正在工作的那个提交；通过那样的方法，当其他人克隆此项目，他们可以重新创建一致的环境。

这是关于子模块的重要一点：你记录他们当前确切所处的提交。你不能记录一个子模块的 **master** 或者其他的符号引用。

当你提交时，会看到类似下面的：

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
```

注意 **rack** 条目的 **160000** 模式。这在 **Git** 中是一个特殊模式，基本意思是你将一个提交记录为一个目录项而不是子目录或者文件。

你可以将 **rack** 目录当作一个独立的项目，保持一个指向子目录的最新提交的指针然后反复地更新上层项目。所有的 **Git** 命令都在两个子目录里独立工作：

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100
```

Document version change

克隆一个带子模块的项目

这里你将克隆一个带子模块的项目。当你接收到这样一个项目，你将得到了包含子项目的目录，但里面没有文件：


```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin  3 Apr  9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr  9 09:11 rack
$ ls rack/
$
```

rack 目录存在了，但是是空的。你必须运行两个命令：**git submodule init** 来初始化你的本地配置文件，**git submodule update** 来从那个项目拉取所有数据并检出你上层项目里所列的合适的提交：

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path
'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

现在你的**rack**子目录就处于你先前提交的确切状态了。如果另外一个开发者变更了 **rack** 的代码并提交，你拉取那个引用然后归并之，将得到稍有点怪异的东西：

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack | 2 +-
1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
```

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rack
#
```

你归并来的仅仅上是一个指向你的子模块的指针；但是它并不更新你子模块目录里的代码，所以看起来你的工作目录处于一个临时状态：

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1,1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

事情就是这样，因为你所拥有的子模块的指针并对应于子模块目录的真实状态。为了修复这一点，你必须再次运行 **git submodule update**：

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b master    -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

每次你从主项目中拉取一个子模块的变更都必须这样做。看起来很怪但是管用。

一个常见问题是当开发者对子模块做了一个本地的变更但是并没有推送到公共服务器。然后他们提交了一个指向那个非公开状态的指针然后推送上层项目。当其他开发者试图运行 **git submodule update**，那个子模块系统会找不到所引用的提交，因为它只存在于第一个开发者的系统中。如果发生那种情况，你会看到类似这样的错误：

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
```

```
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path
'rack'
```

你不得不去查看谁最后变更了子模块

```
$ git log -1 rack

commit 85a3eee996800fcfa91e2119372dd4172bf76678

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Apr 9 09:19:14 2009 -0700
```

```
added a submodule reference I will never make public. hahahahaha!
```

然后，你给那个家伙发电子邮件说他一通。

上层项目

有时候，开发者想按照他们的分组获取一个大项目的子目录的子集。如果你是从 **CVS** 或者 **Subversion** 迁移过来的话这个很常见，在那些系统中你已经定义了一个模块或者子目录的集合，而你想延续这种类型的工作流程。

在 **Git** 中实现这个的一个好办法是你将每一个子目录都做成独立的 **Git** 仓库，然后创建一个上层项目的 **Git** 仓库包含多个子模块。这个办法的一个优势是你可以在上层项目中通过标签和分支更为明确地定义项目之间的关系。

子模块的问题

使用子模块并非没有任何缺点。首先，你在子模块目录中工作时必须相对小心。当你运行 **git submodule update**，它会检出项目的指定版本，但是不在分支内。这叫做获得一个分离的头——这意味着 **HEAD** 文件直接指向一次提交，而不是一个符号引用。问题在于你通常并不想在一个分离的头的的环境下工作，因为太容易丢失变更了。如果你先执行了一次 **submodule update**，然后在那个子模块目录里不创建分支就进行提交，然后再次从上层项目里运行 **git submodule update** 同时不进行提交，**Git** 会毫无提示地覆盖你的变更。技术上讲你不会丢失工作，但是你将失去指向它的分支，因此会很难取到。

为了避免这个问题，当你在子模块目录里工作时应使用 **git checkout -b work** 创建一个分支。当你再次在子模块里更新的时候，它仍然会覆盖你的工作，但是至少你拥有一个可以回溯的指针。

切换带有子模块的分支同样也很有技巧。如果你创建一个新的分支，增加了一个子模块，然后切换回不带该子模块的分支，你仍然会拥有一个未被追踪的子模块的目录

```
$ git checkout -b rack

Switched to a new branch "rack"
```

```

$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   rack/

```

你将不得不将它移走或者删除，这样的话当你切换回去的时候必须重新克隆它——你可能会丢失你未推送的本地的变更或分支。

最后一个需要引起注意的是关于从子目录切换到子模块的。如果你已经跟踪了你项目中的一些文件但是想把它们移到子模块去，你必须非常小心，否则 **Git** 会生你的气。假设你的项目中有一个子目录里放了 **rack** 的文件，然后你想将它转换为子模块。如果你删除子目录然后运行 **submodule add**，**Git** 会向你大吼：

```

$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index

```

你必须先将 **rack** 目录撤回。然后你才能加入子模块：

```

$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.

```

```
Resolving deltas: 100% (1952/1952), done.
```

现在假设你在一个分支里那样做了。如果你尝试切换回一个仍然在目录里保留那些文件而不是子模块的分支时——你会得到下面的错误：

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

你必须先移除 **rack** 子模块的目录才能切换到不包含它的分支：

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

然后，当你切换回来，你会得到一个空的 **rack** 目录。你可以运行 **git submodule update** 重新克隆，也可以将 **/tmp/rack** 目录重新移回空目录。

6.7 子树合并

现在你已经看到了子模块系统的麻烦之处，让我们来看一下解决相同问题的另一途径。当 **Git** 归并时，它会检查需要归并的内容然后选择一个合适的归并策略。如果你归并的分支是两个，**Git** 使用一个_递归_策略。如果你归并的分支超过两个，**Git** 采用_章鱼_策略。这些策略是自动选择的，因为递归策略可以处理复杂的三路归并情况——比如多于一个共同祖先的——但是它只能处理两个分支的归并。章鱼归并可以处理多个分支但是但必须更加小心以避免冲突带来的麻烦，因此它被选中作为归并两个以上分支的默认策略。

实际上，你也可以选择其他策略。其中的一个就是_子树_归并，你可以用它来处理子项目问题。这里你会看到如何换用子树归并的方法来实现前一节里所做的 **rack** 的嵌入。

子树归并的思想是你拥有两个工程，其中一个项目映射到另外一个项目的子目录中，反过来也一样。当你指定一个子树归并，**Git** 可以聪明地探知其中一个是另外一个的子树从而实现正确的归并——这相当神奇。

首先你将 **Rack** 应用加入到项目中。你将 **Rack** 项目当作你项目中的一个远程引用，然后将它检出到它自身的分支：

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
```

```
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
* [new branch]      build      -> rack_remote/build
* [new branch]      master     -> rack_remote/master
* [new branch]      rack-0.4   -> rack_remote/rack-0.4
* [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

现在在你的 **rack_branch** 分支中就有了 **Rack** 项目的根目录，而你自己的项目在 **master** 分支中。如果你先检出其中一个然后另外一个，你会看到它们有不同的项目根目录：

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

要将 **Rack** 项目当作子目录拉取到你的 **master** 项目中。你可以在 **Git** 中用 **git read-tree** 来实现。你会在第9章学到更多与 **read-tree** 和它的朋友相关的东西，当前你会知道它读取一个分支的根目录树到当前的暂存区和工作目录。你只要切换回你的 **master** 分支，然后拉取 **rack** 分支到你主项目的 **master** 分支的 **rack** 子目录：

```
$ git read-tree --prefix=rack/ -u rack_branch
```

当你提交的时候，看起来就像你在那个子目录下拥有 **Rack** 的文件——就像你从一个 **tarball** 里拷贝的一样。有意思的是你可以比较容易地归并其中一个分支的变更到另外一个。因此，如果 **Rack** 项目更新了，你可以通过切换到那个分支并执行拉取来获得上游的变更：

```
$ git checkout rack_branch
$ git pull
```

然后，你可以将那些变更归并回你的 **master** 分支。你可以使用 **git merge -s subtree**，它会工作的很好；但是 **Git** 同时会把历史归并到一起，这可能不是你想要的。为了拉取变更并预置提交说明，需要在 **-s subtree** 策略选项的同时使用 **--squash** 和 **--no-commit** 选项。

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

所有 **Rack** 项目的变更都被归并可以进行本地提交。你也可以做相反的事情——在你主分支的 **rack** 目录里进行变更然后归并回 **rack_branch** 分支，然后将它们提交给维护者或者推送到上游。

为了得到 **rack** 子目录和你 **rack_branch** 分支的区别——以决定你是否需要归并它们——你不能使用一般的 **diff** 命令。而是对你想比较的分支运行 **git diff-tree**:

```
$ git diff-tree -p rack_branch
```

或者，为了比较你的 **rack** 子目录和服务器的 **master** 分支，你可以运行

```
$ git diff-tree -p rack_remote/master
```

自定义 Git

到目前为止，我阐述了 Git 基本的运作机制和使用方式，介绍了 Git 提供的许多工具来帮助你简单且有效地使用它。在本章，我将会介绍 Git 的一些重要的配置方法和钩子机制以满足自定义的要求。通过这些工具，它会和你和公司或团队配合得天衣无缝。

7.1 配置 Git

如第一章所言，用 `git config` 配置 Git，要做的第一件事就是设置名字和邮箱地址：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

从现在开始，你会了解到一些类似以上但更为有趣的设置选项来自定义 Git。

先过一遍第一章中提到的 Git 配置细节。Git 使用一系列的配置文件来存储你定义的偏好，它首先会查找 `/etc/gitconfig` 文件，该文件含有 对系统上所有用户及他们所拥有的仓库都生效的配置值（译注： `gitconfig` 是全局配置文件），如果传递 `--system` 选项给 `git config` 命令，Git 会读写这个文件。

接下来 Git 会查找每个用户的 `~/.gitconfig` 文件，你能传递 `--global` 选项让 Git 读写该文件。

最后 Git 会查找由用户定义各个库中 Git 目录下的配置文件（`.git/config`），该文件中的值只对属主库有效。以上阐述的三层配置从一般到特殊层层推进，如果定义的值有冲突，以后面层中定义的为准，例如：在 `.git/config` 和 `/etc/gitconfig` 的较量中，`.git/config` 取得了胜利。虽然你也可以直接手动编辑这些配置文件，但是运行 `git config` 命令将会来得简单些。

客户端基本配置

Git 能够识别的配置项被分为了两大类：客户端和服务端，其中大部分基于你个人工作偏好，属于客户端配置。尽管有数不尽的选项，但我只阐述 其中经常使用或者会对你的工作流产生巨大影响的选项，如果你想观察你当前的 Git 能识别的选项列表，请运行

```
$ git config --help
```

`git config` 的手册页（译注：以 `man` 命令的显示方式）非常细致地罗列了所有可用的配置项。

core.editor

Git 默认会调用你的环境变量 `editor` 定义的值作为文本编辑器，如果没有定义的话，会调用 `Vi` 来创建和编辑提交以及标签信息，你可以使用 `core.editor` 改变默认编辑器：

```
$ git config --global core.editor emacs
```

现在无论你的环境变量 `editor` 被定义成什么，Git 都会调用 `Emacs` 编辑信息。

commit.template

如果把此项指定为你系统上的一个文件，当你提交的时候，Git 会默认使用该文件定义的内容。例如：你创建了一个模板文件\$HOME/.gitmessage.txt，它看起来像这样：

```
subject line
```

```
what happened
```

```
[ticket: X]
```

设置 commit.template，当运行 git commit 时，Git 会在你的编辑器中显示以上的内容，设置 commit.template 如下：

```
$ git config --global commit.template $HOME/.gitmessage.txt
```

```
$ git commit
```

然后当你提交时，在编辑器中显示的提交信息如下：

```
subject line
```

```
what happened
```

```
[ticket: X]
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# modified:   lib/test.rb
```

```
#
```

```
~
```

```
~
```

```
".git/COMMIT_EDITMSG" 14L, 297C
```

如果你有特定的策略要运用在提交信息上，在系统上创建一个模板文件，设置 Git 默认使用它，这样当提交时，你的策略每次都会被运用。

core.pager

core.pager 指定 Git 运行诸如 log、diff 等所使用的分页器，你能设置成用 more 或者任何

你喜欢的分页器（默认用的是 **less**），当然你也可以什么都不用，设置空字符串：

```
$ git config --global core.pager ''
```

这样不管命令的输出量多少，都会在一页显示所有内容。

user.signingkey

如果你要创建经签署的含附注的标签（正如第二章所述），那么把你的 **GPG** 签署密钥设置为配置项会更好，设置密钥 ID 如下：

```
$ git config --global user.signingkey <gpg-key-id>
```

现在你能够签署标签，从而不必每次运行 **git tag** 命令时定义密钥：

```
$ git tag -s <tag-name>
```

core.excludesfile

正如第二章所述，你能在项目库的 **.gitignore** 文件里头用模式来定义那些无需纳入 **Git** 管理的文件，这样它们不会出现在未跟踪列表，也不会在你运行 **git add** 后被暂存。然而，如果你想用项目库之外的文件来定义那些需被忽略的文件的话，用 **core.excludesfile** 通知 **Git** 该文件所处的位置，文件内容和 **.gitignore** 类似。

help.autocorrect

该配置项只在 **Git 1.6.1**及以上版本有效，假如你在 **Git 1.6**中错打了一条命令，会显示：

```
$ git com
```

```
git: 'com' is not a git-command. See 'git --help'.
```

```
Did you mean this?
```

```
commit
```

如果你把 **help.autocorrect** 设置成 **1**（译注：启动自动修正），那么在只有一个命令被模糊匹配到的情况下，**Git** 会自动运行该命令。

Git 中的着色

Git 能够为输出到你终端的内容着色，以便你可以凭直观进行快速、简单地分析，有许多选项能供你使用以符合你的偏好。

color.ui

Git 会按照你需要自动为大部分的输出加上颜色，你能明确地规定哪些需要着色以及怎样着色，设置 **color.ui** 为 **true** 来打开所有的默认终端着色。

```
$ git config --global color.ui true
```

设置好以后，当输出到终端时，Git 会为之加上颜色。其他的参数还有 **false** 和 **always**，**false** 意味着不为输出着色，而 **always** 则表明在任何情况下都要着色，即使 Git 命令被重定向到文件或管道。Git 1.5.5版本引进了此项配置，如果你拥有的版本更老，你必须对颜色有关选项各自进行详细地设置。

你会很少用到 **color.ui = always**，在大多数情况下，如果你想在被重定向的输出中插入颜色码，你能传递**--color**标志给 Git 命令来迫使它这么做，**color.ui = true** 应该是你的首选。

color.*

想要具体到哪些命令输出需要被着色以及怎样着色或者 Git 的版本很老，你就要用到和具体命令有关的颜色配置选项，它们都能被置为 **true**、**false** 或 **always**：

```
color.branch
color.diff
color.interactive
color.status
```

除此之外，以上每个选项都有子选项，可以被用来覆盖其父设置，以达到为输出的各个部分着色的目的。例如，让 **diff** 输出的改变信息以粗体、蓝色前景和黑色背景的形式显示：

```
$ git config --global color.diff.meta "blue black bold"
```

你能设置的颜色值如：**normal**、**black**、**red**、**green**、**yellow**、**blue**、**magenta**、**cyan**、**white**，正如以上例子设置的粗体属性，想要设置字体属性的话，可以选择如：**bold**、**dim**、**ul**、**blink**、**reverse**。

如果你想配置子选项的话，可以参考 **git config** 帮助页。

外部的合并与比较工具

虽然 Git 自己实现了 **diff**，而且到目前为止你一直在使用它，但你能够用一个外部的工具替代它，除此以外，你还能用一个图形化的工具来合并和解决冲突从而不必自己手动解决。有一个不错且免费的工具可以被用来做比较和合并工作，它就是 **P4Merge**（译注：**Perforce** 图形化合并工具），我会展示它的安装过程。

P4Merge 可以在所有主流平台上运行，现在开始大胆尝试吧。对于向你展示的例子，在 **Mac** 和 **Linux** 系统上，我会使用路径名，在 **Windows** 上，**/usr/local/bin** 应该被改为你环境中的可执行路径。

下载 **P4Merge**：

```
http://www.perforce.com/perforce/downloads/component.html
```

首先把你要运行的命令放入外部包装脚本中，我会使用 **Mac** 系统上的路径来指定该脚本的

位置, 在其他系统上, 它应该被放置在二进制文件 **p4merge** 所在的目录中。创建一个 **merge** 包装脚本, 名字叫作 **extMerge**, 让它带参数调用 **p4merge** 二进制文件:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

diff 包装脚本首先确定传递过来7个参数, 随后把其中2个传递给 **merge** 包装脚本, 默认情况下, **Git** 传递以下参数给 **diff**:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

由于你仅仅需要 **old-file** 和 **new-file** 参数, 用 **diff** 包装脚本来传递它们吧。

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

确认这两个脚本是可执行的:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

现在来配置使用你自定义的比较和合并工具吧。这需要许多自定义设置: **merge.tool** 通知 **Git** 使用哪个合并工具; **mergetool.*.cmd** 规定命令运行的方式; **mergetool.trustExitCode** 会通知 **Git** 程序的退出是否指示合并操作成功; **diff.external** 通知 **Git** 用什么命令做比较。因此, 你能运行以下4条配置命令:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

或者直接编辑 **~/.gitconfig** 文件如下:

```
[merge]
    tool = extMerge
[mergetool "extMerge"]
    cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
    trustExitCode = false
[diff]
    external = extDiff
```

设置完毕后，运行 **diff** 命令：

```
$ git diff 32d1776b1^ 32d1776b1
```

命令行居然没有发现 **diff** 命令的输出，其实，Git 调用了刚刚设置的 **P4Merge**，它看起来像图7-1这样：

Figure 7-1. P4Merge.当你设法合并两个分支，结果却有冲突时，运行 **git mergetool**，Git 会调用 **P4Merge** 让你通过图形界面来解决冲突。

设置包装脚本的好处是你能简单地改变 **diff** 和 **merge** 工具，例如把 **extDiff** 和 **extMerge** 改成 **KDiff3**，要做的仅仅是编辑 **extMerge** 脚本文件：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

现在 Git 会使用 **KDiff3**来做比较、合并和解决冲突。

Git 预先设置了许多其他的合并和解决冲突的工具，而你不必设置 **cmd**。可以把合并工具设置为：**kdiff3**、**opendiff**、**tkdiff**、**meld**、**xxdiff**、**emerge**、**vimdiff**、**gvimdiff**。如果你不想用到 **KDiff3**的所有功能，只是想用它来合并，那么 **kdiff3** 正符合你的要求，运行：

```
$ git config --global merge.tool kdiff3
```

如果运行了以上命令，没有设置 **extMerge** 和 **extDiff** 文件，Git 会用 **KDiff3**做合并，让通常内设的比较工具来做比较。

格式化与空白

格式化与空白是许多开发人员在协作时，特别是在跨平台情况下，遇到的令人头疼的细小问题。由于编辑器的不同或者 **Windows** 程序员在跨平台项目中的文件行尾加入了回车换行符，一些细微的空格变化会不经意地进入大家合作的工作或提交的补丁中。不用怕，Git 的一些配置选项会帮助你解决这些问题。

core.autocrlf

假如你正在 **Windows** 上写程序，又或者你正在和其他人合作，他们在 **Windows** 上编程，而你却在其他系统上，在这些情况下，你可能会遇到行尾结束符问题。这是因为 **Windows** 使用回车和换行两个字符来结束一行，而 **Mac** 和 **Linux** 只使用换行一个字符。虽然这是小问题，但它会极大地扰乱跨平台协作。

Git 可以在你提交时自动地把行结束符 CRLF 转换成 LF，而在签出代码时把 LF 转换成 CRLF。用 `core.autocrlf` 来打开此项功能，如果是在 Windows 系统上，把它设置成 `true`，这样当签出代码时，LF 会被转换成 CRLF：

```
$ git config --global core.autocrlf true
```

Linux 或 Mac 系统使用 LF 作为行结束符，因此你不想 Git 在签出文件时进行自动的转换；当一个以 CRLF 为行结束符的文件不小心被引入时你肯定想进行修正，把 `core.autocrlf` 设置成 `input` 来告诉 Git 在提交时把 CRLF 转换成 LF，签出时不转换：

```
$ git config --global core.autocrlf input
```

这样会在 Windows 系统上的签出文件中保留 CRLF，会在 Mac 和 Linux 系统上，包括仓库中保留 LF。

如果你是 Windows 程序员，且正在开发仅运行在 Windows 上的项目，可以设置 `false` 取消此功能，把回车符记录在库中：

```
$ git config --global core.autocrlf false
```

core.whitespace

Git 预先设置了一些选项来探测和修正空白问题，其4种主要选项中的2个默认被打开，另2个被关闭，你可以自由地打开或关闭它们。

默认被打开的2个选项是 `trailing-space` 和 `space-before-tab`，`trailing-space` 会查找每行结尾的空格，`space-before-tab` 会查找每行开头的制表符前的空格。

默认被关闭的2个选项是 `indent-with-non-tab` 和 `cr-at-eol`，`indent-with-non-tab` 会查找8个以上空格（非制表符）开头的行，`cr-at-eol` 让 Git 知道行尾回车符是合法的。

设置 `core.whitespace`，按照你的意图来打开或关闭选项，选项以逗号分割。通过逗号分割的链中去掉选项或在选项前加 `-` 来关闭，例如，如果你想要打开除了 `cr-at-eol` 之外的所有选项：

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

当你运行 `git diff` 命令且为输出着色时，Git 探测到这些问题，因此你也许在提交前能修复它们，当你用 `git apply` 打补丁时同样也会从中受益。如果正准备运用的补丁有特别的空白问题，你可以让 Git 发警告：

```
$ git apply --whitespace=warn <patch>
```

或者让 Git 在打上补丁前自动修正此问题：

```
$ git apply --whitespace=fix <patch>
```

这些选项也能运用于衍合。如果提交了有空白问题的文件但还没推送到上流，你可以运行带有 `--whitespace=fix` 选项的 `rebase` 来让 Git 在重写补丁时自动修正它们。

服务器端配置

Git 服务器端的配置选项并不多，但仍有一些饶有生趣的选项值得你一看。

`receive.fsckObjects`

Git 默认情况下不会在推送期间检查所有对象的一致性。虽然会确认每个对象的有效性以及是否仍然匹配 SHA-1 检验和，但 Git 不会在每次推送时都检查一致性。对于 Git 来说，库或推送的文件越大，这个操作代价就相对越高，每次推送会消耗更多时间，如果想在每次推送时 Git 都检查一致性，设置 `receive.fsckObjects` 为 `true` 来强迫它这么做：

```
$ git config --system receive.fsckObjects true
```

现在 Git 会在每次推送生效前检查库的完整性，确保有问题的客户端没有引入破坏性的数据。

`receive.denyNonFastForwards`

如果对已经被推送的提交历史做衍合，继而再推送，又或者以其它方式推送一个提交历史至远程分支，且该提交历史没在这个远程分支中，这样的推送会被拒绝。这通常是个很好的禁止策略，但有时你在做衍合并确定要更新远程分支，可以在 `push` 命令后加 `-f` 标志来强制更新。

要禁用这样的强制更新功能，可以设置 `receive.denyNonFastForwards`：

```
$ git config --system receive.denyNonFastForwards true
```

稍后你会看到，用服务器端的接收钩子也能达到同样的目的。这个方法可以做更细致的控制，例如：禁用特定的用户做强制更新。

`receive.denyDeletes`

规避 `denyNonFastForwards` 策略的方法之一就是用户删除分支，然后推回新的引用。在更新的 Git 版本中（从 1.6.1 版本开始），把 `receive.denyDeletes` 设置为 `true`：

```
$ git config --system receive.denyDeletes true
```

这样会在推送过程中阻止删除分支和标签 — 没有用户能够这么做。要删除远程分支，必须从服务器手动删除引用文件。通过用户访问控制列表也能这么做，在本章结尾将会介绍这些有趣的方式。

7.2 Git 属性

一些设置项也能被运用于特定的路径中，这样，Git 以对一个特定的子目录或子文件集运用那些设置项。这些设置项被称为 Git 属性，可以在你目录中的.gitattributes 文件内进行设置（通常是你项目的根目录），也可以当你不想让这些属性文件和项目文件一同提交时，在.git/info/attributes 进行设置。

使用属性，你可以对个别文件或目录定义不同的合并策略，让 Git 知道怎样比较非文本文件，在你提交或签出前让 Git 过滤内容。你将在这部分了解到能在自己的项目中使用的属性，以及一些实例。

二进制文件

你可以用 Git 属性让其知道哪些是二进制文件（以防 Git 没有识别出来），以及指示怎样处理这些文件，这点很酷。例如，一些文本文件是由机器产生的，而且无法比较，而一些二进制文件可以比较 — 你将会了解到怎样让 Git 识别这些文件。

识别二进制文件

一些文件看起来像是文本文件，但其实是作为二进制数据被对待。例如，在 Mac 上的 Xcode 项目含有一个以.pbxproj 结尾的文件，它是由记录设置项的 IDE 写到磁盘的 JSON 数据集（纯文本 javascript 数据类型）。虽然技术上看它是由 ASCII 字符组成的文本文件，但你并不认为如此，因为它确实是一个轻量级数据库 — 如果有2人改变了它，你通常无法合并和比较内容，只有机器才能进行识别和操作，于是，你想把它当成二进制文件。

让 Git 把所有 pbxproj 文件当成二进制文件，在.gitattributes 文件中设置如下：

```
*.pbxproj -crlf -diff
```

现在，Git 会尝试转换和修正 CRLF（回车换行）问题，也不会当你在项目中运行 git show 或 git diff 时，比较不同的内容。在 Git 1.6 及之后的版本中，可以用一个宏代替 -crlf -diff：

```
*.pbxproj binary
```

比较二进制文件

在 Git 1.6 及以上版本中，你能利用 Git 属性来有效地比较二进制文件。可以设置 Git 把二进制数据转换成文本格式，用通常的 diff 来比较。

这个特性很酷，而且鲜为人知，因此我会结合实例来讲解。首先，要解决的是最令人头疼的问题：对 Word 文档进行版本控制。很多人对 Word 文档又恨又爱，如果想对其进行版本控制，你可以把文件加入到 Git 库中，每次修改后提交即可。但这样做没有一点实际意义，因为运行 git diff 命令后，你只能得到如下的结果：

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
```



```
Binary files a/chapter1.doc and b/chapter1.doc differ
```

你不能直接比较两个不同版本的 **Word** 文件，除非进行手动扫描，不是吗？ **Git** 属性能很好地解决此问题，把下面的行加到 **.gitattributes** 文件：

```
*.doc diff=word
```

当你要看比较结果时，如果文件扩展名是 **doc**，**Git** 调用 **word** 过滤器。什么是 **word** 过滤器呢？其实就是 **Git** 使用 **strings** 程序，把 **Word** 文档转换成可读的文本文件，之后再进行比较：

```
$ git config diff.word.textconv strings
```

现在如果在两个快照之间比较以 **.doc** 结尾的文件，**Git** 对这些文件运用 **word** 过滤器，在比较前把 **Word** 文件转换成文本文件。

下面展示了一个实例，我把此书的第一章纳入 **Git** 管理，在一个段落中加入了一些文本后保存，之后运行 **git diff** 命令，得到结果如下：

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
    re going to cover how to get it and set it up for the first time if you don
    t already have it on your system.
    In Chapter Two we will go over basic Git usage - how to use Git for the 80%
-s going on, modify stuff and contribute changes. If the book spontaneously
+
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let's see if this works.
```

Git 成功且简洁地显示出我增加的文本 **Let's see if this works**”。虽然有些瑕疵，在末尾显示了一些随机的内容，但确实可以比较了。如果你能找到或自己写个 **Word** 到纯文本的转换器的话，效果可能会更好。**strings** 可以在大部分 **Mac** 和 **Linux** 系统上运行，所以它是处理二进制格式的第一选择。

你还能用这个方法比较图像文件。当比较时，对 **JPEG** 文件运用一个过滤器，它能提炼出 **EXIF** 信息 — 大部分图像格式使用的元数据。如果你下载并安装了 **exiftool** 程序，可以用它参照元数据把图像转换成文本。比较的不同结果将会用文本向你展示：

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

如果在项目中替换了一个图像文件，运行 `git diff` 命令的结果如下：

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 
   ExifTool Version Number      : 7.74
-File Size                     : 70 kB
-File Modification Date/Time   : 2009:04:21 07:02:45-07:00
+File Size                     : 94 kB
+File Modification Date/Time   : 2009:04:21 07:02:43-07:00
+
+File Type                     : PNG
+MIME Type                     : image/png
-Image Width                   : 1058
-Image Height                   : 889
+Image Width                   : 1056
+Image Height                   : 827
+
+Bit Depth                     : 8
+Color Type                     : RGB with Alpha
```

你会发现文件的尺寸大小发生了改变。

关键字扩展

使用 **SVN** 或 **CVS** 的开发人员经常要求关键字扩展。在 **Git** 中，你无法在一个文件被提交后修改它，因为 **Git** 会先对该文件计算校验和。然而，你可以在签出时注入文本，在提交前删除它。 **Git** 属性提供了2种方式这么做。

首先，你能够把 **blob** 的 **SHA-1**校验和自动注入文件的`Id`字段。如果在一个或多个文件上设置了此字段，当下次你签出分支的时候，**Git** 用 **blob** 的 **SHA-1**值替换那个字段。注意，这不是提交对象的 **SHA** 校验和，而是 **blob** 本身的校验和：

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

下次签出文件时，**Git** 入了 **blob** 的 **SHA** 值：

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

然而，这样的显示结果没有多大的实际意义。这个 **SHA** 的值相当地随机，无法区分日期的前后，所以，如果你在 **CVS** 或 **Subversion** 中用过关键字替换，一定会包含一个日期值。

因此，你能写自己的过滤器，在提交文件到暂存区或签出文件时替换关键字。有2种过滤器，“**clean**”和“**smudge**”。在 **.gitattributes** 文件中，你能对特定的路径设置一个过滤器，然后设置处理文件的脚本，这些脚本会在文件签出前（“**smudge**”，见图 7-2）和提交到暂存区前（“**clean**”，见图7-3）被调用。这些过滤器能够做各种有趣的事。

图7-2. 签出时，“smudge”过滤器被触发。

图7-3. 提交到暂存区时，“clean”过滤器被触发。这里举一个简单的例子：在暂存前，用 **indent**（缩进）程序过滤所有 **C** 源代码。在 **.gitattributes** 文件中设置“**indent**”过滤器过滤*.c 文件：

```
*.c    filter=indent
```

然后，通过以下配置，让 **Git** 知道“**indent**”过滤器在遇到“**smudge**”和“**clean**”时分别该做什么：

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

于是，当你暂存*.c 文件时，**indent** 程序会被触发，在把它们签出之前，**cat** 程序会被触发。但 **cat** 程序在这里没什么实际作用。这样的组合，使 **C** 源代码在暂存前被 **indent** 程序过滤，非常有效。

另一个例子是类似 **RCS** 的 **\$Date\$** 关键字扩展。为了演示，需要一个小脚本，接受文件名参数，得到项目的最新提交日期，最后把日期写入该文件。下面用 **Ruby** 脚本来实现：

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:@"%ad" -1`
puts data.gsub('$Date$', 'Date: ' + last_date.to_s + '$')
```

该脚本从 **git log** 命令中得到最新提交日期，找到文件中的所有 **\$Date\$** 字符串，最后把该日期填充到 **\$Date\$** 字符串中 — 此脚本很简单，你可以选择你喜欢的编程语言来实现。把该脚本命名为 **expand_date**，放到正确的路径中，之后需要在 **Git** 中设置一个过滤器（**dater**），让它在签出文件时调用 **expand_date**，在暂存文件时用 **Perl** 清除之：

```
$ git config filter.dater.smudge expand_date
```

```
$ git config filter.dater.clean 'perl -pe  
"s/\\\$Date[^\$]*\\\$/\\\$Date\\\$/'
```

这个 Perl 小程序会删除`$Date$`字符串里多余的字符，恢复`$Date$`原貌。到目前为止，你的过滤器已经设置完毕，可以开始测试了。打开一个文件，在文件中输入`$Date$`关键字，然后设置 Git 属性：

```
$ echo '# $Date$' > date_test.txt  
$ echo 'date*.txt filter=dater' >> .gitattributes
```

如果暂存该文件，之后再签出，你会发现关键字被替换了：

```
$ git add date_test.txt .gitattributes  
$ git commit -m "Testing date expansion in Git"  
$ rm date_test.txt  
$ git checkout date_test.txt  
$ cat date_test.txt  
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

虽说这项技术对自定义应用来说很有用，但还是要小心，因为`.gitattributes`文件会随着项目一起提交，而过滤器（例如：`dater`）不会，所以，过滤器不会在所有地方都生效。当你在设计这些过滤器时要注意，即使它们无法正常工作，也要让整个项目运作下去。

导出仓库

Git 属性在导出项目归档时也能发挥作用。

export-ignore

当产生一个归档时，可以设置 Git 不导出某些文件和目录。如果你不想在归档中包含一个子目录或文件，但想他们纳入项目的版本管理中，你能对应地设置 `export-ignore` 属性。

例如，在 `test/` 子目录中有一些测试文件，在项目的压缩包中包含他们是没有意义的。因此，可以增加下面这行到 Git 属性文件中：

```
test/ export-ignore
```

现在，当运行 `git archive` 来创建项目的压缩包时，那个目录不会在归档中出现。

export-subst

还能对归档做一些简单的关键字替换。在第2章中已经可以看到，可以以`--pretty=format`形式的简码在任何文件中放入`$Format$`字符串。例如，如果想在项目中包含一个叫作 `LAST_COMMIT` 的文件，当运行 `git archive` 时，最后提交日期自动地注入进该文件，可以这样设置：

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

运行 **git archive** 后，打开该文件，会发现其内容如下：

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

合并策略

通过 **Git** 属性，还能对项目中的特定文件使用不同的合并策略。一个非常有用的选项就是，当一些特定文件发生冲突，**Git** 会尝试合并他们，而使用你这边的合并。

如果项目的一个分支有歧义或比较特别，但你想从该分支合并，而且需要忽略其中某些文件，这样的合并策略是有用的。例如，你有一个数据库设置文件 **database.xml**，在2个分支中他们是不同的，你想合并一个分支到另一个，而不弄乱该数据库文件，可以设置属性如下：

```
database.xml merge=ours
```

如果合并到另一个分支，**database.xml** 文件不会有合并冲突，显示如下：

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

这样，**database.xml** 会保持原样。

7.3 Git 挂钩

和其他版本控制系统一样，当某些重要事件发生时，**Git** 以调用自定义脚本。有两组挂钩：客户端和服务端。客户端挂钩用于客户端的操作，如提交和合并。服务端挂钩用于 **Git** 服务器端的操作，如接收被推送的提交。你可以随意地使用这些挂钩，下面会讲解其中一些。

安装一个挂钩

挂钩都被存储在 **Git** 目录下的 **hooks** 子目录中，即大部分项目中的 **.git/hooks**。**Git** 默认会放置一些脚本样本在这个目录中，除了可以作为挂钩使用，这些样本本身是可以独立使用的。所有的样本都是 **shell** 脚本，其中一些还包含了 **Perl** 的脚本，不过，任何正确命名的可执行脚本都可以正常使用 — 可以用 **Ruby** 或 **Python**，或其他。在 **Git 1.6**版本之后，这些样本名都是以 **sample** 结尾，因此，你必须重新命名。在 **Git 1.6**版本之前，这些样本名都是正确的，但这些样本不是可执行文件。

把一个正确命名且可执行的文件放入 **Git** 目录下的 **hooks** 子目录中，可以激活该挂钩脚本，因此，之后他一直被 **Git** 调用。随后会讲解主要的挂钩脚本。

客户端挂钩

有许多客户端挂钩，以下把他们分为：提交 workflow 挂钩、电子邮件 workflow 挂钩及其他客户端挂钩。

提交 workflow 挂钩

有 4 个挂钩被用来处理提交的过程。**pre-commit** 挂钩在键入提交信息前运行，被用来检查即将提交的快照，例如，检查是否有东西被遗漏，确认测试是否运行，以及检查代码。当从该挂钩返回非零值时，**Git** 放弃此次提交，但可以用 **git commit --no-verify** 来忽略。该挂钩可以被用来检查代码错误（运行类似 **lint** 的程序），检查尾部空白（默认挂钩是这么做的），检查新方法（译注：程序的函数）的说明。

prepare-commit-msg 挂钩在提交信息编辑器显示之前，默认信息被创建之后运行。因此，可以有机会在提交作者看到默认信息前进行编辑。该挂钩接收一些选项：拥有提交信息的文件路径，提交类型，如果是一次修订的话，提交的 **SHA-1** 校验和。该挂钩对通常的提交来说不是很有用，只在自动产生的默认提交信息的情况下有作用，如提交信息模板、合并、压缩和修订提交等。可以和提交模板配合使用，以编程的方式插入信息。

commit-msg 挂钩接收一个参数，此参数是包含最近提交信息的临时文件的路径。如果该挂钩脚本以非零退出，**Git** 放弃提交，因此，可以用来在提交通过前验证项目状态或提交信息。本章上一小节已经展示了使用该挂钩核对提交信息是否符合特定的模式。

post-commit 挂钩在整个提交过程完成后运行，他不会接收任何参数，但可以运行 **git log -1 HEAD** 来获得最后的提交信息。总之，该挂钩是作为通知之类使用的。

提交 workflow 的客户端挂钩脚本可以在任何 workflow 中使用，他们经常被用来实施某些策略，但值得注意的是，这些脚本在 **clone** 期间不会被传送。可以在服务器端实施策略来拒绝不符合某些策略的推送，但这完全取决于开发者在客户端使用这些脚本的情况。所以，这些脚本对开发者是有用的，由他们自己设置和维护，而且在任何时候都可以覆盖或修改这些脚本。

E-mail workflow 挂钩

有 3 个可用的客户端挂钩用于 **e-mail** workflow。当运行 **git am** 命令时，会调用他们，因此，如果你没有在工作流中用到此命令，可以跳过本节。如果你通过 **e-mail** 接收由 **git format-patch** 产生的补丁，这些挂钩也许对你有用。

首先运行的是 **applypatch-msg** 挂钩，他接收一个参数：包含被建议提交信息的临时文件名。如果该脚本非零退出，**Git** 放弃此补丁。可以使用这个脚本确认提交信息是否被正确格式化，或让脚本编辑信息以达到标准化。

下一个在 `git am` 运行期间调用是 `pre-applypatch` 挂钩。该挂钩不接收参数，在补丁被运用之后运行，因此，可以被用来在提交前检查快照。你能用此脚本运行测试，检查工作树。如果有什么遗漏，或测试没通过，脚本会以非零退出，放弃此次 `git am` 的运行，补丁不会被提交。

最后在 `git am` 运行期间调用的是 `post-applypatch` 挂钩。你可以用他来通知一个小组或获取的补丁的作者，但无法阻止打补丁的过程。

其他客户端挂钩

`pre-rebase` 挂钩在符合前运行，脚本以非零退出可以中止符合的过程。你可以使用这个挂钩来禁止符合已经推送的提交对象，`Git pre-rebase` 挂钩样本就是这么做的。该样本假定 `next` 是你定义的分支名，因此，你可能要修改样本，把 `next` 改成你定义过且稳定的分支名。

在 `git checkout` 成功运行后，`post-checkout` 挂钩会被调用。他可以用来为你的项目环境设置合适的工作目录。例如：放入大的二进制文件、自动产生的文档或其他一切你不想纳入版本控制的文件。

最后，在 `merge` 命令成功执行后，`post-merge` 挂钩会被调用。他可以用来在 `Git` 无法跟踪的工作树中恢复数据，诸如权限数据。该挂钩同样能够验证在 `Git` 控制之外的文件是否存在，因此，当工作树改变时，你想这些文件可以被复制。

服务器端挂钩

除了客户端挂钩，作为系统管理员，你还可以使用两个服务器端的挂钩对项目实施各种类型的策略。这些挂钩脚本可以在提交对象推送到服务器前被调用，也可以在推送到服务器后被调用。推送到服务器前调用的挂钩可以在任何时候以非零退出，拒绝推送，返回错误消息给客户端，还可以如你所愿设置足够复杂的推送策略。

`pre-receive` 和 `post-receive`

处理来自客户端的推送（`push`）操作时最先执行的脚本就是 `pre-receive`。它从标准输入（`stdin`）获取被推送引用的列表；如果它退出时的返回值不是0，所有推送内容都不会被接受。利用此挂钩脚本可以实现类似保证最新的索引中不包含非 `fast-forward` 类型的这类效果；抑或检查执行推送操作的用户拥有创建，删除或者推送的权限或者他是否对将要修改的每一个文件都有访问权限。

`post-receive` 挂钩在整个过程完结以后运行，可以用来更新其他系统服务或者通知用户。它接受与 `pre-receive` 相同的标准输入数据。应用实例包括给某邮件列表发信，通知实时整合数据的服务器，或者更新软件项目的问题追踪系统——甚至可以通过分析提交信息来决定某个问题是否应该被开启，修改或者关闭。该脚本无法组织推送进程，不过客户端在它完成运行之前将保持连接状态；所以在用它作一些消耗时间的操作之前请三思。

update

`update` 脚本和 `pre-receive` 脚本十分类似。不同之处在于它将为推送者更新的每一个分支运行一次。假如推送者同时向多个分支推送内容，`pre-receive` 只运行一次，相比之下 `update` 则会为每一个更新的分支运行一次。它不会从标准输入读取内容，而是接受三个参数：索引的名字（分支），推送前索引指向的内容的 **SHA-1** 值，以及用户试图推送内容的 **SHA-1** 值。如果 `update` 脚本以退出时返回非零值，只有相应的那一个索引会被拒绝；其余的依然会得到更新。

7.4 Git 强制策略实例

在本节中，我们应用前面学到的知识建立这样一个 **Git** 工作流程：检查提交信息的格式，只接受纯 **fast-forward** 内容的推送，并且指定用户只能修改项目中的特定子目录。我们将写一个客户端脚本来提示开发人员他们推送的内容是否会被拒绝，以及一个服务端脚本来实际执行这些策略。

这些脚本使用 **Ruby** 写成，一半由于它是作者倾向的脚本语言，另外作者觉得它是最接近伪代码的脚本语言；因而即便你不使用 **Ruby** 也能大致看懂。不过任何其他语言也一样适用。所有 **Git** 自带的样例脚本都是用 **Perl** 或 **Bash** 写的。所以从这些脚本中能找到相当多的这两种语言的挂钩样例。

服务端挂钩

所有服务端的工作都在 **hooks**（挂钩）目录的 `update`（更新）脚本中制定。`update` 脚本为每一个得到推送的分支运行一次；它接受推送目标的索引，该分支原来指向的位置，以及被推送的新内容。如果推送是通过 **SSH** 进行的，还可以获取发出此次操作的用户。如果设定所有操作都通过公匙授权的单一帐号（比如 "**git**"）进行，就有必要通过一个 **shell** 包装依据公匙来判断用户的身份，并且设定环境变量来表示该用户的身份。下面假设尝试连接的用户储存在 `$USER` 环境变量里，我们的 `update` 脚本首先搜集一切需要的信息：

```
#!/usr/bin/env ruby
```

```
$refname = ARGV[0]
```

```
$oldrev = ARGV[1]
```

```
$newrev = ARGV[2]
```

```
$user = ENV['USER']
```

```
puts "Enforcing Policies... \n(#{ $refname }) (#{ $oldrev[0,6] })
```

```
(#{ $newrev[0,6] })"
```


没错，我在用全局变量。别鄙视我——这样比较利于演示过程。

指定特殊的提交信息格式

我们的第一项任务是指定每一条提交信息都必须遵循某种特殊的格式。作为演示，假定每一条信息必须包含一条形似“**ref: 1234**”这样的字符串，因为我们需要把每一次提交和项目的问题追踪系统。我们要逐一检查每一条推送上来的提交内容，看看提交信息是否包含这么一个字符串，然后，如果该提交里不包含这个字符串，以非零返回值退出从而拒绝此次推送。

把 `$newrev` 和 `$oldrev` 变量的值传给一个叫做 `git rev-list` 的 `Git plumbing` 命令可以获取所有提交内容的 **SHA-1** 值列表。`git rev-list` 基本类似 `git log` 命令，但它默认只输出 **SHA-1** 值而已，没有其他信息。所以要获取由 **SHA** 值表示的从一次提交到另一次提交之间的所有 **SHA** 值，可以运行：

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

截取这些输出内容，循环遍历其中每一个 **SHA** 值，找出与之对应的提交信息，然后用正则表达式来测试该信息包含的格式话的内容。

下面要搞定如何从所有的提交内容中提取出提交信息。使用另一个叫做 `git cat-file` 的 `Git plumbing` 工具可以获得原始的提交数据。我们将在第九章了解到这些 `plumbing` 工具的细节；现在暂时先看一下这条命令的输出：

```
$ git cat-file commit ca82a6
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

```
changed the version number
```

通过 **SHA-1** 值获得提交内容中的提交信息的一个简单办法是找到提交的第一行，然后取从它往后的所有内容。可以使用 `Unix` 系统的 `sed` 命令来实现该效果：

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

这条咒语从每一个待提交内容里提取提交信息，并且会在提取信息不符合要求的情况下退

出。为了退出脚本和拒绝此次推送，返回一个非零值。整个脚本大致如下：

```
$regex = /\[ref: (\d+)\]/

# 指定提交信息格式

def check_message_format

  missed_revs = `git rev-list #{oldrev}..#{newrev}`.split("\n")

  missed_revs.each do |rev|

    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`

    if !$regex.match(message)

      puts "[POLICY] Your message is not formatted correctly"

      exit 1

    end

  end

end

check_message_format
```

把这一段放在 **update** 脚本里，所有包含不符合指定规则的提交都会遭到拒绝。

实现基于用户的访问权限控制列表（ACL）系统

假设你需要添加一个使用访问权限控制列表的机制来指定哪些用户对项目的哪些部分有推送权限。某些用户具有全部的访问权，其他人只对某些子目录或者特定的文件具有推送权限。要搞定这一点，所有的规则将被写入一个位于服务器的原始 **Git** 仓库的 **acl** 文件。我们让 **update** 挂钩检阅这些规则，审视推送的提交内容中需要修改的所有文件，然后决定执行推送的用户是否对所有这些文件都有权限。

我们首先要创建这个列表。这里使用的格式和 **CVS** 的 **ACL** 机制十分类似：它由若干行构成，第一项内容是 **avail** 或者 **unavail**，接着是逗号分隔的规则生效用户列表，最后一项是规则生效的目录（空白表示开放访问）。这些项目由 **|** 字符隔开。

下例中，我们指定几个管理员，几个对 **doc** 目录具有权限的文档作者，以及一个对 **lib** 和 **tests** 目录具有权限的开发人员，相应的 **ACL** 文件如下：

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

首先把这些数据读入你编写的数据结构。本例中，为保持简洁，我们暂时只实现 **avail** 的规则（译注：也就是省略了 **unavail** 部分）。下面这个方法生成一个关联数组，它的主键是用户名，值是一个该用户有写权限的所有目录组成的数组：

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

针对之前给出的 ACL 规则文件，这个 `get_acl_access_data` 方法返回的数据结构如下：

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

搞定了用户权限的数据，下面需要找出哪些位置将要被提交的内容修改，从而确保试图推送的用户对这些位置有全部的权限。

使用 `git log` 的 `--name-only` 选项（在第二章里简单的提过）我们可以轻而易举的找出一提交里修改的文件：

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README
lib/test.rb
```

使用 `get_acl_access_data` 返回的 ACL 结构来一一核对每一次提交修改的文件列表，就能找出该用户是否有权限推送所有的提交内容：

```
# 仅允许特定用户修改项目中的特定子目录
```

```

def check_directory_perms

  access = get_acl_access_data('acl')

  # 检查是否有人在向他没有权限的地方推送内容

  new_commits = `git rev-list #{oldrev}..#{newrev}`.split("\n")

  new_commits.each do |rev|

    files_modified = `git log -1 --name-only --pretty=format:''
#{rev}`.split("\n")

    files_modified.each do |path|

      next if path.size == 0

      has_file_access = false

      access[$user].each do |access_path|

        if !access_path # 用户拥有完全访问权限

          || (path.index(access_path) == 0) # 或者对此位置有访问权限

          has_file_access = true

        end

      end

      if !has_file_access

        puts "[POLICY] You do not have access to push to #{path}"

        exit 1

      end

    end

  end

end

```

check_directory_perms

以上的大部分内容应该都比较容易理解。通过 `git rev-list` 获取推送到服务器内容的提交列表。然后，针对其中每一项，找出它试图修改的文件然后确保执行推送的用户对这些文件具有权限。一个不太容易理解的 Ruby 技巧是 `path.index(access_path) == 0` 这句，它的返回真值如果路径以 `access_path` 开头——这是为了确保 `access_path` 并不是只在允许的路径之一，而是所有准许全选的目录都在该目录之下。

现在你的用户没法推送带有不正确的提交信息的内容，也不能在准许他们访问范围之外的位置做出修改。

只允许 **Fast-Forward** 类型的推送

剩下的最后一项任务是指定只接受 `fast-forward` 的推送。在 Git 1.6 或者更新版本里，只

需要设定 `receive.denyDeletes` 和 `receive.denyNonFastForwards` 选项就可以了。但是通过挂钩的实现可以在旧版本的 **Git** 上工作，并且通过一定的修改它它可以做到只针对某些用户执行，或者更多以后可能用的到的规则。

检查这一项的逻辑是看看提交里是否包含从旧版本里能找到但在新版本里却找不到的内容。如果没有，那这是一次纯 **fast-forward** 的推送；如果有，那我们拒绝此次推送：

```
# 只允许纯 fast-forward 推送

def check_fast_forward

  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end
```

```
check_fast_forward
```

一切都设定好了。如果现在运行 `chmod u+x .git/hooks/update` —— 修改包含以上内容文件的权限，然后尝试推送一个包含非 **fast-forward** 类型的索引，会得到一下提示：

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non-fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

这里有几个有趣的信息。首先，我们可以看到挂钩运行的起点：

```
Enforcing Policies...
```

```
(refs/heads/master) (fb8c72) (c56860)
```

注意这是从 **update** 脚本开头输出到标准你输出的。所有从脚本输出的提示都会发送到客户端，这点很重要。

下一个值得注意的部分是错误信息。

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

第一行是我们的脚本输出的，在往下是 **Git** 在告诉我们 **update** 脚本退出时返回了非零值因而推送遭到了拒绝。最后一点：

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

我们将为每一个被挂钩拒之门外的索引受到一条远程信息，解释它被拒绝是因为一个挂钩的原因。

而且，如果那个 **ref** 字符串没有包含在任何的提交里，我们将看到前面脚本里输出的错误信息：

```
[POLICY] Your message is not formatted correctly
```

又或者某人想修改一个自己不具备权限的文件然后推送了一个包含它的提交，他将看到类似的提示。比如，一个文档作者尝试推送一个修改到 **lib** 目录的提交，他会看到

```
[POLICY] You do not have access to push to lib/test.rb
```

全在这了。从这里开始，只要 **update** 脚本存在并且可执行，我们的仓库永远都不会遭到回转或者包含不符合要求信息的提交内容，并且用户都被锁在了沙箱里面。

客户端挂钩

这种手段的缺点在于用户推送内容遭到拒绝后几乎无法避免的抱怨。辛辛苦苦写成的代码在最后时刻惨遭拒绝是十分悲剧切具迷惑性的；更可怜的是他们不得不修改提交历史来解决问题，这怎么也算不上王道。

逃离这种两难境地的法宝是给用户一些客户端的挂钩，在他们作出可能悲剧的事情的时候给以警告。然后呢，用户们就能在提交-问题变得更难修正之前解除隐患。由于挂钩本身不跟随克隆的项目副本分发，所以必须通过其他途径把这些挂钩分发到用户的 **.git/hooks** 目录并设为可执行文件。虽然可以在相同或单独的项目内容里加入并分发它们，全自动的解决方案是不存在的。

首先，你应该在每次提交前核查你的提交注释信息，这样你才能确保服务器不会因为不合条件的提交注释信息而拒绝你的更改。为了达到这个目的，你可以增加'commit-msg'挂钩。如果你使用该挂钩来阅读作为第一个参数传递给 **git** 的提交注释信息，并且与规定的模式作对比，你就可以使 **git** 在提交注释信息不符合条件的情况下，拒绝执行提交。

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

如果这个脚本放在这个位置 (**.git/hooks/commit-msg**) 并且是可执行的，并且你的提交注释信息不是符合要求的，你会看到：

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

在这个实例中，提交没有成功。然而如果你的提交注释信息是符合要求的，**git** 会允许你提交：

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
1 files changed, 1 insertions(+), 0 deletions(-)
```

接下来我们要保证没有修改到 **ACL** 允许范围之外的文件。加入你的 **.git** 目录里有前面使用过的 **ACL** 文件，那么以下的 **pre-commit** 脚本将把里面的规定执行起来：

```
#!/usr/bin/env ruby

$user = ENV['USER']

# [ insert acl_access_data method from above ]

# 只允许特定用户修改项目重特定子目录的内容
def check_directory_perms
  access = get_acl_access_data('.git/acl')
```

```

files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
files_modified.each do |path|
  next if path.size == 0
  has_file_access = false
  access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
  end
  if !has_file_access
    puts "[POLICY] You do not have access to push to #{path}"
    exit 1
  end
end
end
end

```

check_directory_perms

这和服务端的脚本几乎一样，除了两个重要区别。第一，**ACL** 文件的位置不同，因为这个脚本在当前工作目录运行，而非 **Git** 目录。**ACL** 文件的目录必须从

```
access = get_acl_access_data('acl')
```

修改成：

```
access = get_acl_access_data('.git/acl')
```

另一个重要区别是获取被修改文件列表的方式。在服务端的时候使用了查看提交纪录的方式，可是目前的提交都还没被记录下来呢，所以这个列表只能从暂存区域获取。和原来的

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

不同，现在要用

```
files_modified = `git diff-index --cached --name-only HEAD`
```

不同的就只有这两点——除此之外，该脚本完全相同。一个小陷阱在于它假设在本地运行的账户和推送到远程服务端的相同。如果这二者不一样，则需要手动设置一下 **\$user** 变量。

最后一项任务是检查确认推送内容中不包含非 **fast-forward** 类型的索引，不过这个需求比较少见。要找出一个非 **fast-forward** 类型的索引，要么符合超过某个已经推送过的提交，要么从本地不同分支推送到远程相同的分支上。

既然服务器将给出无法推送非 **fast-forward** 内容的提示，而且上面的挂钩也能阻止强制的推送，唯一剩下的潜在问题就是符合一次已经推送过的提交内容。

下面是一个检查这个问题的 **pre-rabase** 脚本的例子。它获取一个所有即将重写的提交内容的列表，然后检查它们是否在远程的索引里已经存在。一旦发现某个提交可以从远程索引里衍变过来，它就放弃衍合操作：

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
end
```

这个脚本利用了一个第六章“修订版本选择”一节中不曾提到的语法。通过这一句可以获得一个所有已经完成推送的提交的列表：

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

SHA^@ 语法解析该次提交的所有祖先。这里我们从检查远程最后一次提交能够衍变获得但从所有我们尝试推送的提交的 **SHA** 值祖先无法衍变获得的提交内容——也就是 **fast-forward** 的内容。

这个解决方案的硬伤在于它有可能很慢而且常常没有必要——只要不用 **-f** 来强制推送，服务器会自动给出警告并且拒绝推送内容。然而，这是个不错的练习而且理论上能帮助用户避

免一次将来不得不折回来修改的衍合操作。

7.5 总结

你已经见识过绝大多数通过自定义 Git 客户端和服务端来来适应自己工作流程和项目内容的方式了。无论你创造出了什么样的工作流程，Git 都能用的顺手。

Git 与其他系统

世界不是完美的。大多数时候，将所有接触到的项目全部转向 Git 是不可能的。有时我们不得不为某个项目使用其他的版本控制系统（VCS, Version Control System），其中比较常见的是 Subversion。你将在本章的第一部分学习使用 git svn，Git 为 Subversion 附带的双向桥接工具。

或许现在你已经在考虑将先前的项目转向 Git。本章的第二部分将介绍如何将项目迁移到 Git：先介绍从 Subversion 的迁移，然后是 Perforce，最后介绍如何使用自定义的脚本进行非标准的导入。

8.1 Git 与 Subversion

当前，大多数开发中的开源项目以及大量的商业项目都使用 Subversion 来管理源码。作为最流行的开源版本控制系统，Subversion 已经存在了接近十年的时间。它在许多方面与 CVS 十分类似，后者是前者出现之前代码控制世界的霸主。

Git 最为重要的特性之一是名为 git svn 的 Subversion 双向桥接工具。该工具把 Git 变成了 Subversion 服务的客户端，从而让你在本地享受到 Git 所有的功能，而后直接向 Subversion 服务器推送内容，仿佛在本地使用了 Subversion 客户端。也就是说，在其他人的忍受古董的同时，你可以在本地享受分支合并，使暂存区域，符合以及 单项挑拣等等。这是个让 Git 偷偷潜入合作开发环境的好东西，在帮助你的开发同伴们提高效率的同时，它还能帮你劝说团队让整个项目框架转向对 Git 的支持。这个 Subversion 之桥是通向分布式版本控制系统（DVCS, Distributed VCS）世界的神奇隧道。

git svn

Git 中所有 Subversion 桥接命令的基础是 git svn。所有的命令都从它开始。相关的命令数目不少，你将通过几个简单的工作流程了解到其中常见的一些。

值得警戒的是，在使用 git svn 的时候，你实际是在与 Subversion 交互，Git 比它要高级复杂的多。尽管可以在本地随意的进行分支和合并，最好还是通过符合保持线性的提交历史，尽量避免类似与远程 Git 仓库动态交互这样的操作。

避免修改历史再重新推送的做法，也不要同时推送到并行的 Git 仓库来试图与其他 Git 用户合作。Subversion 只能保存单一的线性提交历史，一不小心就会被搞糊涂。合作团队中同时有人用 SVN 和 Git，一定要确保所有人都使用 SVN 服务来协作——这会让生活轻松很多。

初始设定

为了展示功能，先要一个具有写权限的 SVN 仓库。如果想尝试这个范例，你必须复制一

份其中的测试仓库。比较简单的做法是使用一个名为 **svnsync** 的工具。较新的 **Subversion** 版本中都带有该工具，它将数据编码为用于网络传输的格式。

要尝试本例，先在本地新建一个 **Subversion** 仓库：

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

然后，允许所有用户修改 **revprop** —— 简单的做法是添加一个总是以 **0** 作为返回值的 **pre-revprop-change** 脚本：

```
$ cat /tmp/test-svn/hooks/pre-revprop-change

#!/bin/sh

exit 0;

$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

现在可以调用 **svnsync init** 加目标仓库，再加源仓库的格式来把该项目同步到本地了：

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

这将建立进行同步所需的属性。可以通过运行以下命令来克隆代码：

```
$ svnsync sync file:///tmp/test-svn

Committed revision 1.

Copied properties for revision 1.

Committed revision 2.

Copied properties for revision 2.

Committed revision 3.

...
```

别看这个操作只花掉几分钟，要是你想把源仓库复制到另一个远程仓库，而不是本地仓库，那将花掉接近一个小时，尽管项目中只有不到 **100** 次的提交。**Subversion** 每次只复制一次修改，把它推送到另一个仓库里，然后周而复始——惊人的低效，但是我们别无选择。

入门

有了可以写入的 **Subversion** 仓库以后，就可以尝试一下典型的工作流程了。我们从 **git svn clone** 命令开始，它会把整个 **Subversion** 仓库导入到一个本地的 **Git** 仓库中。提醒一下，这里导入的是一个货真价实的 **Subversion** 仓库，所以应该把下面的 **file:///tmp/test-svn** 换成你所用的 **Subversion** 仓库的 **URL**：

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags

Initialized empty Git repository in
/Users/schacon/projects/testsvnsync/svn/.git/
```

```

r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
    A    m4/acx_pthread.m4
    A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn/branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
file:///tmp/test-svn/branches/my-calc-branch r76

```

这相当于针对所提供的 URL 运行了两条命令——`git svn init` 加上 `gitsvn fetch`。可能会花上一段时间。我们所用的测试项目仅仅包含 75 次提交并且它的代码量不算大，所以只有几分钟而已。不过，Git 仍然需要提取每一个版本，每次一个，再逐个提交。对于一个包含成百上千次提交的项目，花掉的时间则可能是几小时甚至数天。

`-T trunk -b branches -t tags` 告诉 Git 该 Subversion 仓库遵循了基本的分支和标签命名法则。如果你的主干(译注: **trunk**，相当于非分布式版本控制里的 **master** 分支，代表开发的主线)，分支或者标签以不同的方式命名，则应做出相应改变。由于该法则的常见性，可以使用 `-s` 来代替整条命令，它意味着标准布局 (`s` 是 **Standard layout** 的首字母)，也就是前面选项的内容。下面的命令有相同的效果：

```
$ git svn clone file:///tmp/test-svn -s
```

现在，你有了一个有效的 Git 仓库，包含着导入的分支和标签：

```

$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk

```

值得注意的是，该工具分配命名空间时和远程引用的方式不尽相同。克隆普通的 Git 仓库时，可以以 `origin/[branch]` 的形式获取远程服务器上所有可用的分支——分配到远程服务

的名称下。然而 **git svn** 假定不存在多个远程服务器，所以把所有指向远程服务的引用不加区分的保存下来。可以用 **Git** 探测命令 **show-ref** 来查看所有引用的全名。

```
$ git show-ref
1cbd4904d9982f386d87f88fcec1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fcec1c24ad7c0f0471 refs/remotes/trunk
```

而普通的 **Git** 仓库应该是这个模样：

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

这里有两个远程服务器：一个名为 **gitserver**，具有一个 **master** 分支；另一个叫 **origin**，具有 **master** 和 **testing** 两个分支。

注意本例中通过 **git svn** 导入的远程引用，（**Subversion** 的）标签是当作远程分支添加的，而不是真正的 **Git** 标签。导入的 **Subversion** 仓库仿佛是有一个带有不同分支的 **tags** 远程服务器。

提交到 **Subversion**

有了可以开展工作的（本地）仓库以后，你可以开始对该项目做出贡献并向上游仓库提交内容了，**Git** 这时相当于一个 **SVN** 客户端。假如编辑了一个文件并进行提交，那么这次提交仅存在于本地的 **Git** 而非 **Subversion** 服务器上。

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

接下来，可以将作出的修改推送到上游。值得注意的是，**Subversion** 的使用流程也因此改变了——你可以在离线状态下进行多次提交然后一次性的推送到 **Subversion** 的服务器上。向 **Subversion** 服务器推送的命令是 **git svn dcommit**：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
```

```
M      README.txt
Committed r79

M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

所有在原 **Subversion** 数据基础上提交的 **commit** 会一一提交到 **Subversion**，然后你本地 **Git** 的 **commit** 将被重写，加入一个特别标识。这一步很重要，因为它意味着所有 **commit** 的 **SHA-1** 指都会发生变化。这也是同时使用 **Git** 和 **Subversion** 两种服务作为远程服务不是个好主意的原因之一。检视以下最后一个 **commit**，你会找到新添加的 **git-svn-id**（译注：即本段开头所说的特别标识）：

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000
```

```
Adding git-svn instructions to the README
```

```
git-svn-id: file:///tmp/test-svn/trunk@79
4c93b258-373f-11de-be05-5f7a86268029
```

注意看，原本以 **97031e5** 开头的 **SHA-1** 校验值在提交完成以后变成了 **938b1a5**。如果既要向 **Git** 远程服务器推送内容，又要推送到 **Subversion** 远程服务器，则必须先向 **Subversion** 推送（**dcommit**），因为该操作会改变所提交的数据内容。

拉取最新进展

如果要与其他开发者协作，总有那么一天你推送完毕之后，其他人发现他们推送自己修改的时候（与你推送的内容）产生冲突。这些修改在你合并之前将一直被拒绝。在 **git svn** 里这种情况形似：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably
\
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

为了解决该问题，可以运行 **git svn rebase**，它会拉取服务器上所有最新的改变，再次基

础上衍合你的修改:

```
$ git svn rebase
      M       README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

现在, 你做出的修改都发生在服务器内容之后, 所以可以顺利的运行 **dcommit** :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M       README.txt
Committed r81
      M       README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

需要牢记的一点是, **Git** 要求我们在推送之前先合并上游仓库中最新的内容, 而 **git svn** 只要求存在冲突的时候才这样做。假如有人向一个文件推送了一些修改, 这时你要向另一个文件推送一些修改, 那么 **dcommit** 将正常工作:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M       configure.ac
Committed r84
      M       autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
      M       configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
    using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
    015e4c98c482f0fa71e4d5434338014530b37fa6 M    autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

这一点需要牢记, 因为它的结果是推送之后项目处于一个不完整存在与任何主机上的状态。如果做出的修改无法兼容但没有产生冲突, 则可能造成一些很难确诊的难题。这和使用 **Git**

服务器是不同的——在 **Git** 世界里，发布之前，你可以在客户端系统里完整的测试项目的状态，而在 **SVN** 永远都没法确保提交前后项目的状态完全一样。

及时还没打算进行提交，你也应该用这个命令从 **Subversion** 服务器拉取最新修改。**git svn fetch** 能获取最新的数据，不过 **git svn rebase** 才会在获取之后在本地进行更新。

```
$ git svn rebase

M      generate_descriptor_proto.sh

r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)

First, rewinding head to replay your work on top of it...

Fast-forwarded master to refs/remotes/trunk.
```

不时地运行一下 **git svn rebase** 可以确保你的代码没有过时。不过，运行该命令时需要确保工作目录的整洁。如果在本地做了修改，则必须在运行 **git svn rebase** 之前或暂存工作，或暂时提交内容——否则，该命令会发现衍合的结果包含着冲突因而终止。

Git 分支问题

习惯了 **Git** 的工作流程以后，你可能会创建一些特性分支，完成相关的开发工作，然后合并他们。如果要用 **git svn** 向 **Subversion** 推送内容，那么最好是每次用衍合来并入一个单一分支，而不是直接合并。使用衍合的原因是 **Subversion** 只有一个线性的历史而不像 **Git** 那样处理合并，所以 **Git svn** 在把快照转换为 **Subversion** 的 **commit** 时只能包含第一个祖先。

假设分支历史如下：创建一个 **experiment** 分支，进行两次提交，然后合并到 **master**。在 **dcommit** 的时候会得到如下输出：

```
$ git svn dcommit

Committing to file:///tmp/test-svn/trunk ...

M      CHANGES.txt

Committed r85

M      CHANGES.txt

r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)

No changes between current HEAD and refs/remotes/trunk

Resetting to the latest refs/remotes/trunk

COPYING.txt: locally modified

INSTALL.txt: locally modified

M      COPYING.txt

M      INSTALL.txt

Committed r86

M      INSTALL.txt
```

```
M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

在一个包含了合并历史的分支上使用 `dcommit` 可以成功运行，不过在 `Git` 项目的历史中，它没有重写你在 `experiment` 分支中的两个 `commit` ——另一方面，这些改变却出现在了 `SVN` 版本中同一个合并 `commit` 中。

在别人克隆该项目的时候，只能看到这个合并 `commit` 包含了所有发生过的修改；他们无法获知修改的作者和时间等提交信息。

Subversion 分支

`Subversion` 的分支和 `Git` 中的不尽相同；避免过多的使用可能是最好方案。不过，用 `git svn` 创建和提交不同的 `Subversion` 分支仍是可行的。

创建新的 SVN 分支

要在 `Subversion` 中建立一个新分支，需要运行 `git svn branch [分支名]` To create a new branch in Subversion, you run `git svn branch [branchname]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to
file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

相当于在 `Subversion` 中的 `svn copy trunk branches/opera` 命令并且对 `Subversion` 服务器进行了相关操作。值得提醒的是它没有检出和转换到那个分支；如果现在进行提交，将提交到服务器上的 `trunk`，而非 `opera`。

切换当前分支

`Git` 通过搜寻提交历史中 `Subversion` 分支的头部来决定 `dcommit` 的目的地——而它应该只有一个，那就是当前分支历史中最近一次包含 `git-svn-id` 的提交。

如果需要同时多个分支上提交，可以通过导入 `Subversion` 上某个其他分支的 `commit` 来建立以该分支为 `dcommit` 目的地的本地分支。比如你想拥有一个并行维护的 `opera` 分

支，可以运行

```
$ git branch opera remotes/opera
```

然后，如果要把 **opera** 分支并入 **trunk**（本地的 **master** 分支），可以使用普通的 **git merge**。不过最好提供一条描述提交的信息（通过 **-m**），否则这次合并的记录是 **Merge branch opera**，而不是任何有用的东西。

记住，虽然使用了 **git merge** 来进行这次操作，并且合并过程可能比使用 **Subversion** 简单一些（因为 **Git** 会自动找到适合的合并基础），这并不是一次普通的 **Git** 合并提交。最终它将被推送回 **commit** 无法包含多个祖先的 **Subversion** 服务器上；因而在推送之后，它将变成一个包含了所有在其他分支上做出的改变的单一 **commit**。把一个分支合并到另一个分支以后，你没法像在 **Git** 中那样轻易的回到那个分支上继续工作。提交时运行的 **dcommit** 命令擦除了全部有关哪个分支被并入的信息，因而以后的合并基础计算将是不正确的——**dcommit** 让 **git merge** 的结果变得类似于 **git merge --squash**。不幸的是，我们没有什么好办法来避免该情况——**Subversion** 无法储存这个信息，所以在使用它作为服务器的时候你将永远为这个缺陷所困。为了不出现这种问题，在把本地分支（本例中的 **opera**）并入 **trunk** 以后应该立即将其删除。

对应 Subversion 的命令

git svn 工具集合了若干个与 **Subversion** 类似的功能，对应的命令可以简化向 **Git** 的转化过程。下面这些命令能实现 **Subversion** 的这些功能。

SVN 风格的历史

习惯了 **Subversion** 的人可能想以 **SVN** 的风格显示历史，运行 **git svn log** 可以让提交历史显示为 **SVN** 格式：

```
$ git svn log
```

```
-----  
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
```

```
autogen change
```

```
-----  
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
```

```
Merge branch 'experiment'
```

```
-----  
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
```

updated the changelog

关于 `git svn log`，有两点需要注意。首先，它可以离线工作，不像 `svn log` 命令，需要向 Subversion 服务器索取数据。其次，它仅仅显示已经提交到 Subversion 服务器上的 `commit`。在本地尚未 `dcommit` 的 Git 数据不会出现在这里；其他人向 Subversion 服务器新提交的数据也不会显示。等于说是显示了最近已知 Subversion 服务器上的状态。

SVN 日志

类似 `git svn log` 对 `git log` 的模拟，`svn annotate` 的等效命令是 `git svn blame [文件名]`。其输出如下：

```
$ git svn blame README.txt

2   temporal Protocol Buffers - Google's data interchange format
2   temporal Copyright 2008 Google Inc.
2   temporal http://code.google.com/apis/protocolbuffers/
2   temporal
22  temporal C++ Installation - Unix
22  temporal =====
2   temporal
79  schacon Committing in git-svn.
78  schacon
2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2   temporal Buffer compiler (protoc) execute the following:
2   temporal
```

同样，它不显示本地的 Git 提交以及 Subversion 上后来更新的内容。

SVN 服务器信息

还可以使用 `git svn info` 来获取与运行 `svn info` 类似的信息：

```
$ git svn info

Path: .

URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
```

Last Changed Rev: 87

Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)

它与 `blame` 和 `log` 的相同点在于离线运行以及只更新到最后一次与 Subversion 服务器通信的状态。

略 Subversion 之所略

假如克隆了一个包含了 `svn:ignore` 属性的 Subversion 仓库，就有必要建立对应的 `.gitignore` 文件来防止意外提交一些不应该提交的文件。`git svn` 有两个有益于改善该问题的命令。第一个是 `git svn create-ignore`，它自动建立对应的 `.gitignore` 文件，以便下次提交的时候可以包含它。

第二个命令是 `git svn show-ignore`，它把需要放进 `.gitignore` 文件中的内容打印到标准输出，方便我们把输出重定向到项目的黑名单文件：

```
$ git svn show-ignore > .git/info/exclude
```

这样一来，避免了 `.gitignore` 对项目的干扰。如果你是一个 Subversion 团队里唯一的 Git 用户，而其他队友不喜欢项目包含 `.gitignore`，该方法是你的不二之选。

Git-Svn 总结

`git svn` 工具集在当前不得不使用 Subversion 服务器或者开发环境要求使用 Subversion 服务器的时候格外有用。不妨把它看成一个跛脚的 Git，然而，你还是有可能在转换过程中碰到一些困惑你和合作者们的谜题。为了避免麻烦，试着遵守如下守则：

- 保持一个不包含由 `git merge` 生成的 `commit` 的线性提交历史。将在主线分支外进行的开发通衍合回主线；避免直接合并。
- 不要单独建立和使用一个 Git 服务来搞合作。可以为了加速新开发者的克隆进程建立一个，但是不要向它提供任何不包含 `git-svn-id` 条目的内容。甚至可以添加一个 `pre-receive` 挂钩来在每一个提交信息中查找 `git-svn-id` 并拒绝提交那些不包含它的 `commit`。

如果遵循这些守则，在 Subversion 上工作还可以接受。然而，如果能迁徙到真正的 Git 服务器，则能为团队带来更多好处。

8.2 迁移到 Git

如果在其他版本控制系统中保存了某项目的代码而后决定转而使用 Git，那么该项目必须经历某种形式的迁移。本节将介绍 Git 中包含的一些针对常见系统的导入脚本，并将展示编写自定义的导入脚本的方法。

导入

你将学习到如何从专业重量级的版本控制系统中导入数据—— Subversion 和 Perforce —— 因为据我所知这二者的用户是（向 Git）转换的主要群体，而且 Git 为此二者附带了高质量的转换工具。

Subversion

读过前一节有关 `git svn` 的内容以后，你应该能轻而易举的根据其中的指导来 `git svn clone` 一个仓库了；然后，停止 Subversion 的使用，向一个新 Git server 推送，并开始使用它。想保留历史记录，所花的时间应该不过就是从 Subversion 服务器拉取数据的时间（可能要等上好一会就是了）。

然而，这样的导入并不完美；而且还要花那么多时间，不如干脆一次把它做对！首当其冲的任务是作者信息。在 Subversion，每个提交者在主机上有一个用户名，记录在提交信息中。上节例子中多处显示了 `schacon`，比如 `blame` 的输出以及 `git svn log`。如果想让这条信息更好的映射到 Git 作者数据里，则需要从 Subversion 用户名到 Git 作者的一个映射关系。建立一个叫做 `user.txt` 的文件，用如下格式表示映射关系：

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

通过该命令可以获得 SVN 作者的列表：

```
$ svn log --xml | grep author | sort -u | perl -pe 's/.>(.*?)<./$1 = /'
```

它将输出 XML 格式的日志——你可以找到作者，建立一个单独的列表，然后从 XML 中抽取需要的信息。（显而易见，本方法要求主机上安装了 `grep`，`sort` 和 `perl`。）然后把输出重定向到 `user.txt` 文件，然后就可以在每一项的后面添加相应的 Git 用户数据。

为 `git svn` 提供该文件可以然它更精确的映射作者数据。你还可以在 `clone` 或者 `init` 后面添加 `--no-metadata` 来阻止 `git svn` 包含那些 Subversion 的附加信息。这样 `import` 命令就变成了：

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
    --authors-file=users.txt --no-metadata -s my_project
```

现在 `my_project` 目录下导入的 Subversion 应该比原来整洁多了。原来的 `commit` 看上去是这样：

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

```
git-svn-id: https://my-project.googlecode.com/svn/trunk@94
4c93b258-373f-11de-
be05-5f7a86268029
```

现在是这样：

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

不仅作者一项干净了不少，**git-svn-id** 也就此消失了。

你还需要一点 **post-import**（导入后）清理工作。最起码的，应该清理一下 **git svn** 创建的那些怪异的索引结构。首先要移动标签，把它们从奇怪的远程分支变成实际的标签，然后把剩下的分支移动到本地。

要把标签变成合适的 **Git** 标签，运行

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

该命令将原本以 **tag/** 开头的远程分支的索引变成真正的（轻巧的）标签。

接下来，把 **refs/remotes** 下面剩下的索引变成本地分支：

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

现在所有的旧分支都变成真正的 **Git** 分支，所有的旧标签也变成真正的 **Git** 标签。最后一项工作就是把新建的 **Git** 服务器添加为远程服务器并且向它推送。下面是新增远程服务器的例子：

```
$ git remote add origin git@my-git-server:myrepository.git
```

为了让所有的分支和标签都得到上传，我们使用这条命令：

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

所有的分支和标签现在都应该整齐干净的躺在新的 **Git** 服务器里了。

Perforce

你将了解到的下一个被导入的系统是 **Perforce**. **Git** 发行的时候同时也附带了一个 **Perforce** 导入脚本，不过它是包含在源码的 **contrib** 部分——而不像 **git svn** 那样默认可用。运行它之前必须获取 **Git** 的源码，可以在 git.kernel.org 下载：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

在这个 **fast-import** 目录下，应该有一个叫做 **git-p4** 的 **Python** 可执行脚本。主机上必须装有 **Python** 和 **p4** 工具该导入才能正常进行。例如，你要从 **Perforce** 公共代码仓库（译注：**Perforce Public Depot**, **Perforce** 官方提供的代码寄存服务）导入 **Jam** 工程。为了设定客户端，我们要把 **P4PORT** 环境变量 **export** 到 **Perforce** 仓库：

```
$ export P4PORT=public.perforce.com:1666
```

运行 **git-p4 clone** 命令将从 **Perforce** 服务器导入 **Jam** 项目，我们需要给出仓库和项目的路径以及导入的目标路径：

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

现在去 **/opt/p4import** 目录运行一下 **git log**，就能看到导入的成果：

```
$ git log -2

commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

[git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```



```
Update derived jamgram.c
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

每一个 **commit** 里都有一个 **git-p4** 标识符。这个标识符可以保留，以防以后需要引用 **Perforce** 的修改版本号。然而，如果想删除这些标识符，现在正是时候——在开启新仓库之前。可以通过 **git filter-branch** 来批量删除这些标识符：

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

现在运行一下 **git log**，你会发现这些 **commit** 的 **SHA-1** 校验值都发生了改变，而那些 **git-p4** 字符串则从提交信息里消失了：

```
$ git log -2

commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

```
Update derived jamgram.c
```

至此导入已经完成，可以开始向新的 **Git** 服务器推送了。

自定义导入脚本

如果先前的系统不是 **Subversion** 或 **Perforce** 之一，先上网找一下有没有与之对应的导入脚本——导入 **CVS**，**Clear Case**，**Visual Source Safe**，甚至存档目录的导入脚本已经存在。假如这些工具都不适用，或者使用的工具很少见，抑或你需要导入过程具有更多可定制性，则应该使用 **git fast-import**。该命令从标准输入读取简单的指令来写入具体的 **Git** 数据。这

样创建 **Git** 对象比运行纯 **Git** 命令或者手动写对象要简单的多（更多相关内容见第九章）。通过它，你可以编写一个导入脚本来从导入源读取必要的信息，同时在标准输出直接输出相关指示。你可以运行该脚本并把它的输出管道连接到 **git fast-import**。

下面演示一下如何编写一个简单的导入脚本。假设你在进行一项工作，并且按时通过把工作目录复制为以时间戳 **back_YY_MM_DD** 命名的目录来进行备份，现在你需要把它们导入 **Git**。目录结构如下：

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

为了导入到一个 **Git** 目录，我们首先回顾一下 **Git** 储存数据的方式。你可能还记得，**Git** 本质上是一个 **commit** 对象的链表，每一个对象指向一个内容的快照。而这里需要做的工作就是告诉 **fast-import** 内容快照的位置，什么样的 **commit** 数据指向它们，以及它们的顺序。我们采取一次处理一个快照的策略，为每一个内容目录建立对应的 **commit**，每一个 **commit** 与之前的建立链接。

正如在第七章“**Git** 执行策略一例”一节中一样，我们将使用 **Ruby** 来编写这个脚本，因为它是我日常使用的语言而且阅读起来简单一些。你可以用任何其他熟悉的语言来重写这个例子——它仅需要把必要的信息打印到标准输出而已。同时，如果你在使用 **Windows**，这意味着你要特别留意不要在换行的时候引入回车符（译注：**carriage returns**，**Windows** 换行时加入的符号，通常说的 **\r**）——**Git** 的 **fast-import** 对仅使用换行符（**LF**）而非 **Windows** 的回车符（**CRLF**）要求非常严格。

首先，进入目标目录并且找到所有子目录，每一个子目录将作为一个快照被导入为一个 **commit**。我们将依次进入每一个子目录并打印所需的命令来导出它们。脚本的主循环大致是这样：

```
last_mark = nil

# 循环遍历所有目录
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # 进入目标目录
```

```

    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
end

```

我们在每一个目录里运行 `print_export`，它会取出上一个快照的索引和标记并返回本次快照的索引和标记；由此我们就可以正确的把二者连接起来。”标记（mark）”是 `fast-import` 中对 `commit` 标识符的叫法；在创建 `commit` 的同时，我们逐一赋予一个标记以便以后在把它连接到其他 `commit` 时使用。因此，在 `print_export` 方法中要做的第一件事就是根据目录名生成一个标记：

```
mark = convert_dir_to_mark(dir)
```

实现该函数的方法是建立一个目录的数组序列并使用数组的索引值作为标记，因为标记必须是一个整数。这个方法大致是这样的：

```

$marks = []

def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end

  ($marks.index(dir) + 1).to_s
end

```

有了整数来代表每个 `commit`，我们现在需要提交附加信息中的日期。由于日期是用目录名表示的，我们就从中解析出来。`print_export` 文件的下一行将是：

```
date = convert_dir_to_date(dir)
```

而 `convert_dir_to_date` 则定义为

```

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end

```

它为每个目录返回一个整型值。提交附加信息里最后一项所需的是提交者数据，我们在一个全局变量中直接定义之：

```
$author = 'Scott Chacon <schacon@example.com>'
```

我们差不多可以开始为导入脚本输出提交数据了。第一项信息指明我们定义的是一个 **commit** 对象以及它所在的分支，随后是我们生成的标记，提交者信息以及提交备注，然后是前一个 **commit** 的索引，如果有的话。代码大致这样：

```
# 打印导入所需的信息

puts 'commit refs/heads/master'

puts 'mark :' + mark

puts "committer #{ $author } #{date} -0700"

export_data('imported from ' + dir)

puts 'from :' + last_mark if last_mark
```

时区（-0700）处于简化目的使用硬编码。如果是从其他版本控制系统导入，则必须以变量的形式指明时区。提交备注必须以特定格式给出：

```
data (size)\n(contents)
```

该格式包含了单词 **data**，所读取数据的大小，一个换行符，最后是数据本身。由于随后指明文件内容的时候要用到相同的格式，我们写一个辅助方法，**export_data**：

```
def export_data(string)

  print "data #{string.size}\n#{string}"

end
```

唯一剩下的就是每一个快照的内容了。这简单的很，因为它们分别处于一个目录——你可以输出 **deleall** 命令，随后是目录中每个文件的内容。**Git** 会正确的记录每一个快照：

```
puts 'deleteall'

Dir.glob("**/*").each do |file| next if !File.file?(file)

  inline_data(file)

end
```

注意：由于很多系统把每次修订看作一个 **commit** 到另一个 **commit** 的变化量，**fast-import** 也可以依据每次提交获取一个命令来指出哪些文件被添加，删除或者修改过，以及修改的内容。我们将需要计算快照之间的差别并且仅仅给出这项数据，不过该做法要复杂很多——还不如不直接把所有数据丢给 **Git** 然它自己搞清楚。假如前面这个方法更适用于你的数据，参考 **fast-import** 的 **man** 帮助页面来了解如何以这种方式提供数据。

列举新文件内容或者指明带有新内容的已修改文件的格式如下：

```
M 644 inline path/to/file
data (size)
(file contents)
```

这里，**644** 是权限模式（加入有可执行文件，则需要探测之并设定为 **755**），而 **inline** 说明我们在本行结束之后立即列出文件的内容。我们的 **inline_data** 方法大致是：

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

我们重用了前面定义过的 **export_data**，因为这里和指明提交注释的格式如出一辙。

最后一项工作是返回当前的标记以便下次循环的使用。

```
return mark
```

注意：如果你在用 **Windows**，一定记得添加一项额外的步骤。前面提过，**Windows** 使用 **CRLF** 作为换行字符而 **Git fast-import** 只接受 **LF**。为了绕开这个问题来满足 **git fast-import**，你需要让 **ruby** 用 **LF** 取代 **CRLF**：

```
$stdout.binmode
```

搞定了。现在运行该脚本，你将得到如下内容：

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
```

```

M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)

```

要运行导入脚本，在需要导入的目录把该内容用管道定向到 **git fast-import**。你可以建立一个空目录然后运行 **git init** 作为开头，然后运行该脚本：

```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:       18 (      1 duplicates      )
  blobs   :          7 (      1 duplicates      0 deltas)
  trees   :          6 (      0 duplicates      1 deltas)
  commits:          5 (      0 duplicates      0 deltas)
  tags    :          0 (      0 duplicates      0 deltas)
Total branches:       1 (      1 loads      )
  marks:      1024 (      5 unique      )
  atoms:           3
Memory total:        2255 KiB
  pools:        2098 KiB
  objects:       156 KiB
-----
pack_report: getpagesize()      =      4096
pack_report: core.packedGitWindowSize =  33554432
pack_report: core.packedGitLimit  =  268435456
pack_report: pack_used_ctr       =           9
pack_report: pack_mmap_calls     =           5
pack_report: pack_open_windows  =           1 /           1
pack_report: pack_mapped        =      1356 /      1356
-----

```

你会发现，在它成功执行完毕以后，会给出一堆有关已完成工作的数据。上例在一个分支导

入了5次提交数据，包含了18个对象。现在可以运行 `git log` 来检视新的历史：

```
$ git log -2

commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700
```

```
    imported from current
```

```
commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700
```

```
    imported from back_2009_02_03
```

就它了——一个干净整洁的 **Git** 仓库。需要注意的是此时没有任何内容被检出——刚开始当前目录里没有任何文件。要获取它们，你得转到 **master** 分支的所在：

```
$ ls

$ git reset --hard master
HEAD is now at 10bfe7d imported from current

$ ls
file.rb lib
```

fast-import 还可以做更多——处理不同的文件模式，二进制文件，多重分支与合并，标签，进展标识等等。一些更加复杂的实例可以在 **Git** 源码的 `contrib/fast-import` 目录里找到；其中较为出众的是前面提过的 **git-p4** 脚本。

8.3 总结

现在的你应该掌握了在 **Subversion** 上使用 **Git** 以及把几乎任何先存仓库无损失的导入为 **Git** 仓库。下一章将介绍 **Git** 内部的原始数据格式，从而是使你能亲手锻造其中的每一个字节，如果必要的话。

Git 内部原理

不管你是从前面的章节直接跳到了本章，还是读完了其余各章一直到这，你都将在本章见识 Git 的内部工作原理和实现方式。我个人发现学习这些内容对于理解 Git 的用处和强大是非常重要的，不过也有人认为这些内容对于初学者来说可能难以理解且过于复杂。正因如此我把这部分内容放在最后一章，你在学习过程中可以先阅读这部分，也可以晚点阅读这部分，这完全取决于你自己。

既然已经读到这了，就让我们开始吧。首先要弄明白一点，从根本上来讲 Git 是一套内容寻址 (content-addressable) 文件系统，在此之上提供了一个 VCS 用户界面。马上你就会学到这意味着什么。

早期的 Git (主要是 1.5 之前版本) 的用户界面要比现在复杂得多，这是因为它更侧重于成为文件系统而不是一套更精致的 VCS。最近几年改进了 UI 从而使它跟其他任何系统一样清晰易用。即便如此，还是经常会有一些陈腔滥调提到早期 Git 的 UI 复杂又难学。

内容寻址文件系统这一层相当酷，在本章中我会先讲解这部分。随后你会学到传输机制和最终要使用的各种库管理任务。

9.1 底层命令 (Plumbing) 和高层命令 (Porcelain)

本书讲解了使用 checkout, branch, remote 等共约 30 个 Git 命令。然而由于 Git 一开始被设计成供 VCS 使用的工具集而不是一整套用户友好的 VCS，它还包含了许多底层命令，这些命令用于以 UNIX 风格使用或由脚本调用。这些命令一般被称为“plumbing”命令（底层命令），其他的更友好的命令则被称为“porcelain”命令（高层命令）。

本书前八章主要专门讨论高层命令。本章将主要讨论底层命令以理解 Git 的内部工作机制、演示 Git 如何及为何要以这种方式工作。这些命令主要不是用来从命令行手工使用的，更多的是用来为其他工具和自定义脚本服务的。

当你在一个新目录或已有目录内执行 git init 时，Git 会创建一个 .git 目录，几乎所有 Git 存储和操作的内容都位于该目录下。如果你要备份或复制一个库，基本上将这一目录拷贝至其他地方就可以了。本章基本上都讨论该目录下的内容。该目录结构如下：

```
$ ls
HEAD
branches/
config
description
hooks/
index
```



```
info/  
objects/  
refs/
```

该目录下有可能还有其他文件，但这是一个全新的 `git init` 生成的库，所以默认情况下这些就是你能看到的结构。新版本的 **Git** 不再使用 `branches` 目录，`description` 文件仅供 **GitWeb** 程序使用，所以不用关心这些内容。`config` 文件包含了项目特有的配置选项，`info` 目录保存了一份不希望在 `.gitignore` 文件中管理的忽略模式 (`ignored patterns`) 的全局可执行文件。`hooks` 目录包住了第六章详细介绍了的客户端或服务端钩子脚本。

另外还有四个重要的文件或目录：`HEAD` 及 `index` 文件，`objects` 及 `refs` 目录。这些是 **Git** 的核心部分。`objects` 目录存储所有数据内容，`refs` 目录存储指向数据 (分支) 的提交对象的指针，`HEAD` 文件指向当前分支，`index` 文件保存了暂存区域信息。马上你将详细了解 **Git** 是如何操纵这些内容的。

9.2 Git 对象

Git 是一套内容寻址文件系统。很不错。不过这是什么意思呢？这种说法的意思是，从内部来看，**Git** 是简单的 **key-value** 数据存储。它允许插入任意类型的内容，并会返回一个键值，通过该键值可以在任何时候再取出该内容。可以通过底层命令 `hash-object` 来示范这点，传一些数据给该命令，它会将数据保存在 `.git` 目录并返回表示这些数据的键值。首先初始化一个 **Git** 仓库并确认 `objects` 目录是空的：

```
$ mkdir test  
$ cd test  
$ git init  
Initialized empty Git repository in /tmp/test/.git/  
$ find .git/objects  
.git/objects  
.git/objects/info  
.git/objects/pack  
$ find .git/objects -type f  
$
```

Git 初始化了 `objects` 目录，同时在该目录下创建了 `pack` 和 `info` 子目录，但是该目录下没有其他常规文件。我们往这个 **Git** 数据库里存储一些文本：

```
$ echo 'test content' | git hash-object -w --stdin  
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

参数 **-w** 指示 **hash-object** 命令存储 (数据) 对象, 若不指定这个参数该命令仅仅返回键值。**--stdin** 指定从标准输入设备 (**stdin**) 来读取内容, 若不指定这个参数则需指定一个要存储的文件的路径。该命令输出长度为 **40** 个字符的校验和。这是个 **SHA-1** 哈希值——其值为要存储的数据加上你马上会了解到的一种头信息的校验和。现在可以查看到 **Git** 已经存储了数据:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

可以在 **objects** 目录下看到一个文件。这便是 **Git** 存储数据内容的方式——为每份内容生成一个文件, 取得该内容与头信息的 **SHA-1** 校验和, 创建以该校验和前两个字符为名称的子目录, 并以 (校验和) 剩下 **38** 个字符为文件命名 (保存至子目录下)。

通过 **cat-file** 命令可以将数据内容取回。该命令是查看 **Git** 对象的瑞士军刀。传入 **-p** 参数可以让该命令输出数据内容的类型:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

可以往 **Git** 中添加更多内容并取回了。也可以直接添加文件。比方说可以对一个文件进行简单的版本控制。首先, 创建一个新文件, 并把文件内容存储到数据库中:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

接着往该文件中写入一些新内容并再次保存:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

数据库中已经将文件的两个新版本连同一开始的内容保存下来了:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

再将文件恢复到第一个版本:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

或恢复到第二个版本：

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

需要记住的是几个版本的文件 **SHA-1** 值可能与实际的值不同，其次，存储的并不是文件名而仅仅是文件内容。这种对象类型称为 **blob**。通过传递 **SHA-1** 值给 **cat-file -t** 命令可以让 **Git** 返回任何对象的类型：

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

tree (树) 对象

接下去来看 **tree** 对象，**tree** 对象可以存储文件名，同时也允许存储一组文件。**Git** 以一种类似 **UNIX** 文件系统但更简单的方式来存储内容。所有内容以 **tree** 或 **blob** 对象存储，其中 **tree** 对象对应于 **UNIX** 中的目录，**blob** 对象则大致对应于 **inodes** 或文件内容。一个单独的 **tree** 对象包含一条或多条 **tree** 记录，每一条记录含有一个指向 **blob** 或子 **tree** 对象的 **SHA-1** 指针，并附有该对象的权限模式 (**mode**)、类型和文件名信息。以 **simplegit** 项目为例，最新的 **tree** 可能是这个样子：

```
$ git cat-file -p master^{tree}

100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

master^{tree} 表示 **branch** 分支上最新提交指向的 **tree** 对象。请注意 **lib** 子目录并非一个 **blob** 对象，而是一个指向别一个 **tree** 对象的指针：

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

从概念上来讲，**Git** 保存的数据如图 9-1 所示。

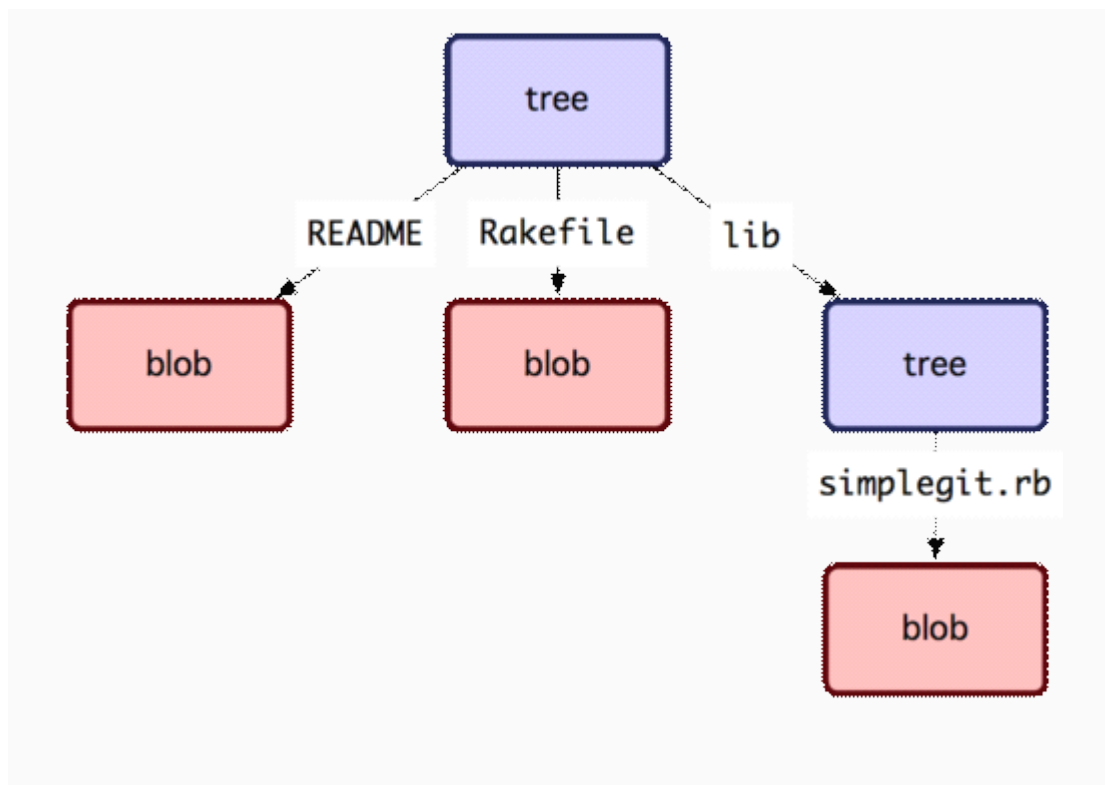


图 9-1. Git 对象模型的简化版你可以自己创建 **tree**。通常 Git 根据你的暂存区域或 **index** 来创建并写入一个 **tree**。因此要创建一个 **tree** 对象的话首先要通过将一些文件暂存从而创建一个 **index**。可以使用 **plumbing** 命令 **update-index** 为一个单独文件 —— **test.txt** 文件的第一个版本 —— 创建一个 **index**。通过该命令人为的将 **test.txt** 文件的首个版本加入到了一个新的暂存区域中。由于该文件原先并不在暂存区域中（甚至连暂存区域也还没被创建出来呢），必须传入 **--add** 参数；由于要添加的文件并不在当前目录下而是在数据库中，必须传入 **--cacheinfo** 参数。同时指定了文件模式，**SHA-1** 值和文件名：

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

在本例中，指定了文件模式为 **100644**，表明这是一个普通文件。其他可用的模式有：**100755** 表示可执行文件，**120000** 表示符号链接。文件模式是从常规的 **UNIX** 文件模式中参考来的，但是没有那么灵活 —— 上述三种模式仅对 Git 中的文件 (**blobs**) 有效（虽然也有其他模式用于目录和子模块）。

现在可以用 **write-tree** 命令将暂存区域的内容写到一个 **tree** 对象了。无需 **-w** 参数 —— 如果目标 **tree** 不存在，调用 **write-tree** 会自动根据 **index** 状态创建一个 **tree** 对象。

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

可以这样验证这确实是一个 **tree** 对象：

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

再根据 **test.txt** 的第二个版本以及一个新文件创建一个新 **tree** 对象：

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

这时暂存区域中包含了 **test.txt** 的新版本及一个新文件 **new.txt**。创建 (写) 该 **tree** 对象 (将暂存区域或 **index** 状态写入到一个 **tree** 对象)，然后瞧瞧它的样子：

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

请注意该 **tree** 对象包含了两个文件记录，且 **test.txt** 的 **SHA** 值是原先值的“第二版” (**1f7a7a**)。来点更有趣的，你将把第一个 **tree** 对象作为一个子目录加进该 **tree** 中。可以用 **read-tree** 命令将 **tree** 对象读到暂存区域中去。在这时，通过传一个 **--prefix** 参数给 **read-tree**，将一个已有的 **tree** 对象作为一个子 **tree** 读到暂存区域中：

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

如果从刚写入的新 **tree** 对象创建一个工作目录，将得到位于工作目录顶级的两个文件和一个名为 **bak** 的子目录，该子目录包含了 **test.txt** 文件的第一个版本。可以将 **Git** 用来包含这些内容的数据想象成如图 9-2 所示的样子。

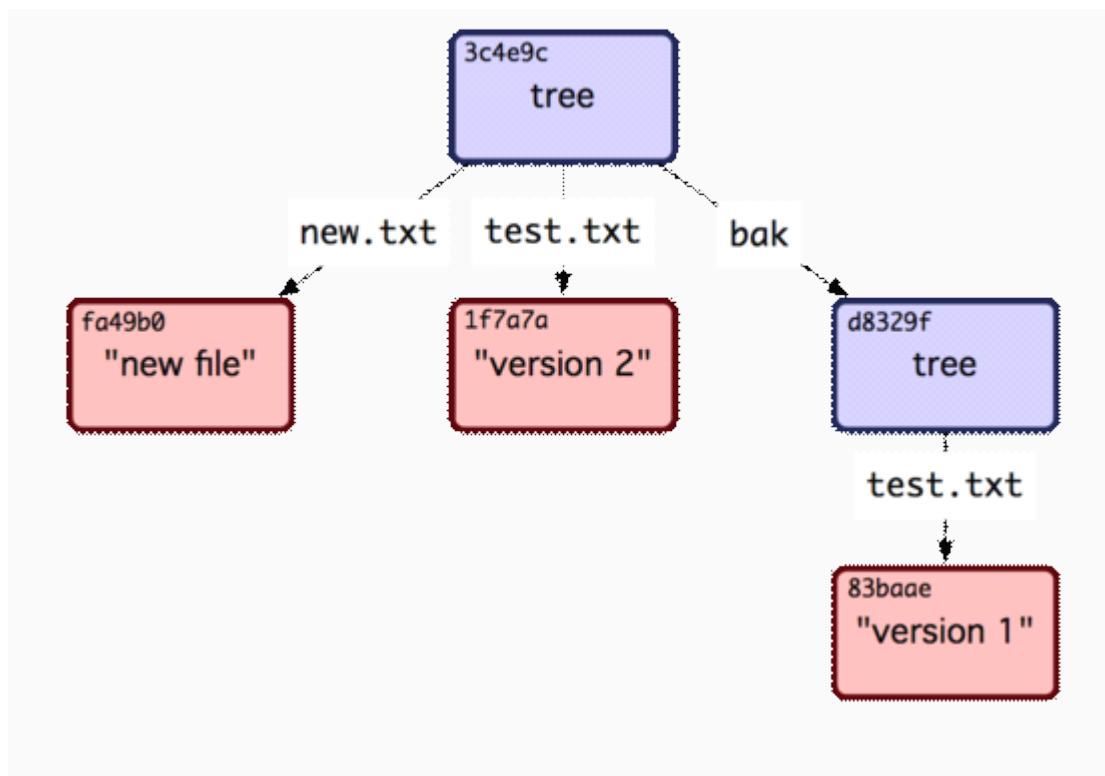


图 9-2. 当前 Git 数据的内容结构 **commit (提交) 对象**

你现在有三个 **tree** 对象，它们指向了你要跟踪的项目的不同快照，可是先前的问题依然存在：必须记住三个 **SHA-1** 值以获得这些快照。你也没有关于谁、何时以及为何保存了这些快照的信息。**commit** 对象为你保存了这些基本信息。

要创建一个 **commit** 对象，使用 **commit-tree** 命令，指定一个 **tree** 的 **SHA-1**，如果有任何前继提交对象，也可以指定。从你写的第一个 **tree** 开始：

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

通过 **cat-file** 查看这个新 **commit** 对象：

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon
```

```
1243040974 -0700 committer Scott Chacon 1243040974 -0700 first commit
```

commit 对象有格式很简单：指明了该时间点项目快照的顶层树对象、作者/提交者信息（从 Git 设理发店的 **user.name** 和 **user.email** 中获得）以及当前时间戳、一个空行，以及提交注释信息。

接着再写入另外两个 **commit** 对象，每一个都指定其之前的那个 **commit** 对象：

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

每一个 **commit** 对象都指向了你创建的树对象快照。出乎意料的是，现在已经有了真实的 **Git** 历史了，所以如果运行 **git log** 命令并指定最后那个 **commit** 对象的 **SHA-1** 便可以查看历史：

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon
```

```

    Date: Fri May 22 18:15:24 2009 -0700 third commit bak/test.txt | 1 + 1 files
changed, 1 insertions(+), 0 deletions(-) commit
cac0cab538b970a37ea1e769cbbde608743bc96d Author: Scott Chacon Date: Fri May 22
18:14:29 2009 -0700 second commit new.txt | 1 + test.txt | 2 +- 2 files changed,
2 insertions(+), 1 deletions(-) commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon Date: Fri May 22 18:09:34 2009 -0700 first commit test.txt
| 1 + 1 files changed, 1 insertions(+), 0 deletions(-)
```

真棒。你刚刚通过使用低级操作而不是那些普通命令创建了一个 **Git** 历史。这基本上就是运行 **git add** 和 **git commit** 命令时 **Git** 进行的工作——保存修改了的文件的 **blob**，更新索引，创建 **tree** 对象，最后创建 **commit** 对象，这些 **commit** 对象指向了顶层 **tree** 对象以及先前的 **commit** 对象。这三类 **Git** 对象——**blob**，**tree** 以及 **tree**——都各自以文件的方式保存在 **.git/objects** 目录下。以下所列是目前为止样例中的所有对象，每个对象后面的注释里标明了它们保存的内容：

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
```

```
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

如果你按照以上描述进行了操作，可以得到如图 9-3 所示的对象图。

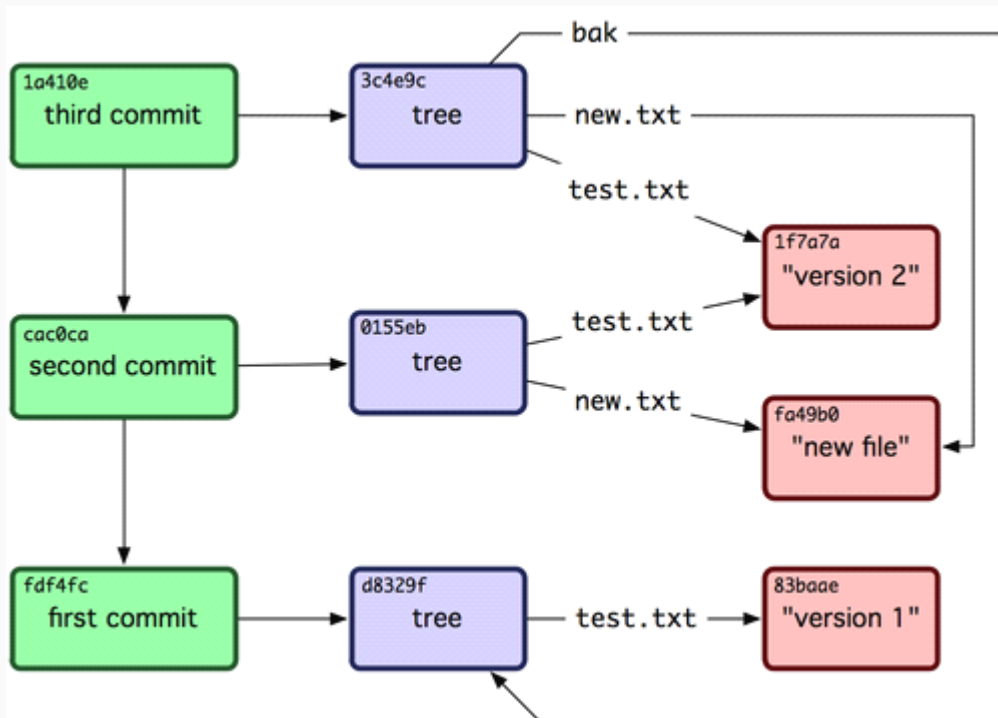


图 9-3. Git 目录下的所有对象对象存储

之前我提到当存储数据内容时，同时会有一个文件头被存储起来。我们花些时间来看看 Git 是如何存储对象的。你将看来如何通过 Ruby 脚本语言存储一个 blob 对象（这里以字符串 “what is up, doc?” 为例）。使用 irb 命令进入 Ruby 交互式模式：

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git 以对象类型为起始内容构造一个文件头，本例中是一个 blob。然后添加一个空格，接着是数据内容的长度，最后是一个空字节 (null byte):

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git 将文件头与原始数据内容拼接起来，并计算拼接后的新内容的 SHA-1 校验和。可以在 Ruby 中使用 require 语句导入 SHA1 digest 库，然后调用 Digest::SHA1.hexdigest() 方法计算字符串的 SHA-1 值：


```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git 用 **zlib** 对数据内容进行压缩，在 Ruby 中可以用 **zlib** 库来实现。首先需要导入该库，然后用 **Zlib::Deflate.deflate()** 对数据进行压缩：

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

最后将用 **zlib** 压缩后的内容写入磁盘。需要指定保存对象的路径 (**SHA-1** 值的头两个字符作为子目录名称，剩余 38 个字符作为文件名保存至该子目录中)。在 Ruby 中，如果子目录不存在可以用 **FileUtils.mkdir_p()** 函数创建它。接着用 **File.open** 方法打开文件，并用 **write()** 方法将之前压缩的内容写入该文件：

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

这就行了 —— 你已经创建了一个正确的 **blob** 对象。所有的 Git 对象都以这种方式存储，惟一的区别是类型不同 —— 除了字符串 **blob**，文件头起始内容还可以是 **commit** 或 **tree**。不过虽然 **blob** 几乎可以是任意内容，**commit** 和 **tree** 的数据却是有固定格式的。

9.3 Git References

你可以执行像 **git log 1a410e** 这样的命令来查看完整的历史，但是这样你就要记得 **1a410e** 是你最后一次提交，这样才能在提交历史中找到这些对象。你需要一个文件来用一个简单的名字来记录这些 **SHA-1** 值，这样你就可以用这些指针而不是原来的 **SHA-1** 值去检索了。

在 **Git** 中，我们称之为“引用”（**references** 或者 **refs**，译者注）。你可以在 `.git/refs` 目录下面找到这些包含 **SHA-1** 值的文件。在这个项目里，这个目录还没不包含任何文件，但是包含这样一个简单的结构：

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

如果想要创建一个新的引用帮助你记住最后一次提交，技术上你可以这样做：

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

现在，你就可以在 **Git** 命令中使用你刚才创建的引用而不是 **SHA-1** 值：

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

当然，我们并不鼓励你直接修改这些引用文件。如果你确实需要更新一个引用，**Git** 提供了一个安全的命令 **update-ref**：

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

基本上 **Git** 中的一个分支其实就是一个指向某个工作版本一条 **HEAD** 记录的指针或引用。你可以用这条命令创建一个指向第二次提交的分支：

```
$ git update-ref refs/heads/test cac0ca
```

这样你的分支将会只包含那次提交以及之前的工作：

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

现在，你的 **Git** 数据库应该看起来像图 9-4 一样。

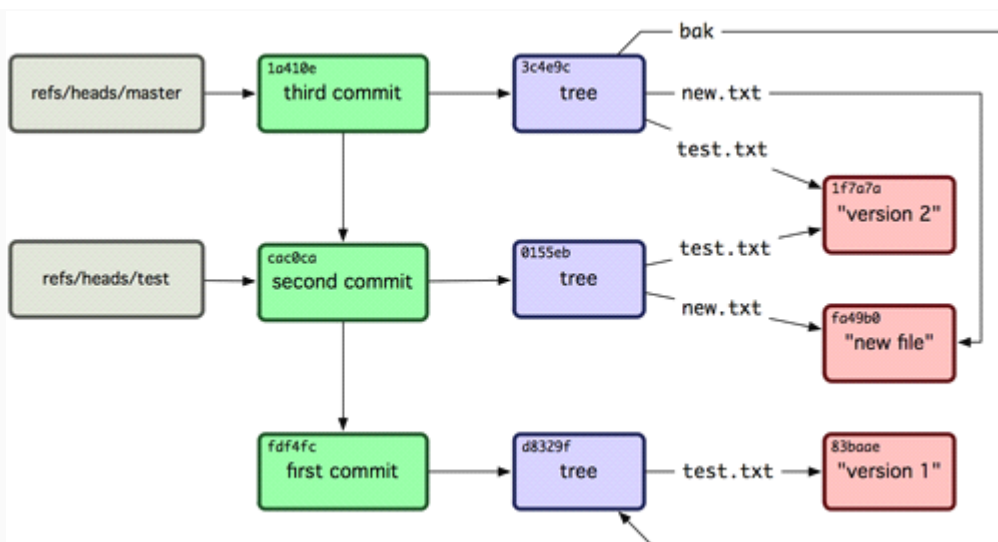


图 9-4. 包含分支引用的 Git 目录对象每当你执行 `git branch (分支名称)` 这样的命令，Git 基本上就是执行 `update-ref` 命令，把你现在所在分支中最后一次提交的 SHA-1 值，添加到你要创建的分支的引用。

HEAD 标记

现在的问题是，当你执行 `git branch (分支名称)` 这条命令的时候，Git 怎么知道最后一次提交的 SHA-1 值呢？答案就是 HEAD 文件。HEAD 文件是一个指向你当前所在分支的引用标识符。这样的引用标识符——它看起来并不像一个普通的引用——其实并不包含 SHA-1 值，而是一个指向另外一个引用的指针。如果你看一下这个文件，通常你将会看到这样的内容：

```
$ cat .git/HEAD
ref: refs/heads/master
```

如果你执行 `git checkout test`，Git 就会更新这个文件，看起来像这样：

```
$ cat .git/HEAD
ref: refs/heads/test
```

当你再执行 `git commit` 命令，它就创建了一个 `commit` 对象，把这个 `commit` 对象的父级设置为 HEAD 指向的引用的 SHA-1 值。

你也可以手动编辑这个文件，但是同样有一个更安全的方法可以这样做：`symbolic-ref`。你可以用下面这条命令读取 HEAD 的值：

```
$ git symbolic-ref HEAD
refs/heads/master
```

你也可以设置 HEAD 的值：

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

但是你不能设置成 **refs** 以外的形式：

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

你刚刚已经重温过了 **Git** 的三个主要对象类型，现在这是第四种。**Tag** 对象非常像一个 **commit** 对象——包含一个标签，一组数据，一个消息和一个指针。最主要的区别就是 **Tag** 对象指向一个 **commit** 而不是一个 **tree**。它就像是一个分支引用，但是不会变化——永远指向同一个 **commit**，仅仅是提供一个更加友好的名字。

正如我们在第二章所讨论的，**Tag** 有两种类型：**annotated** 和 **lightweight**。你可以类似下面这样的命令建立一个 **lightweight tag**：

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

这就是 **lightweight tag** 的全部 —— 一个永远不会发生变化的分支。**annotated tag** 要更复杂一点。如果你创建一个 **annotated tag**，**Git** 会创建一个 **tag** 对象，然后写入一个指向指向它而不是直接指向 **commit** 的 **reference**。你可以这样创建一个 **annotated tag**（**-a** 参数表明这是一个 **annotated tag**）：

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

这是所创建对象的 **SHA-1** 值：

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

现在你可以运行 **cat-file** 命令检查这个 **SHA-1** 值：

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon
```

```
Sat May 23 16:48:58 2009 -0700 test tag
```

值得注意的是这个对象指向你所标记的 **commit** 对象的 **SHA-1** 值。同时需要注意的是它并不是必须要指向一个 **commit** 对象；你可以标记任何 **Git** 对象。例如，在 **Git** 的源代码里，管理者添加了一个 **GPG** 公钥（这是一个 **blob** 对象）对它做了一个标签。你就可以运行：

```
$ git cat-file blob junio-gpg-pub
```

来查看 **Git** 源代码仓库中的公钥。**Linux kernel** 也有一个不是指向 **commit** 对象的 **tag** —— 第一个 **tag** 是在导入源代码的时候创建的，它指向初始 **tree**（**initial tree**，译者注）。

Remotes

你将会看到的第四种 **reference** 是 **remote reference**（远程引用，译者注）。如果你添加了一个 **remote** 然后推送代码过去，**Git** 会把你最后一次推送到这个 **remote** 的每个分支的值都记录在 **refs/remotes** 目录下。例如，你可以添加一个叫做 **origin** 的 **remote** 然后把你的 **master** 分支推送上去：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master

Counting objects: 11, done.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (7/7), 716 bytes, done.

Total 7 (delta 2), reused 4 (delta 1)

To git@github.com:schacon/simplegit-progit
    allbef0..ca82a6d master -> master
```

然后查看 **refs/remotes/origin/master** 这个文件，你就会发现 **origin remote** 中的 **master** 分支就是你最后一次和服务器的通信。

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote 应用和分支主要区别在于他们是不能被 **check out** 的。**Git** 把他们当作是标记这些了这些分支在服务器上最后状态的一种书签。

9.4 Packfiles

我们再来看一下 **test Git** 仓库。目前为止，有 11 个对象 —— 4 个 **blob**，3 个 **tree**，3 个 **commit** 以及一个 **tag**：

```
$ find .git/objects -type f

.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
```

```
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git 用 **zlib** 压缩文件内容，因此这些文件并没有占用太多空间，所有文件加起来总共仅用了 **925** 字节。接下去你会添加一些大文件以演示 Git 的一个很有意思的功能。将你之前用过的 **Grit** 库中的 **repo.rb** 文件加进去 —— 这个源代码文件大小约为 **12K**：

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
3 files changed, 459 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

如果查看一下生成的 **tree**，可以看到 **repo.rb** 文件的 **blob** 对象的 **SHA-1** 值：

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

然后可以用 **git cat-file** 命令查看这个对象有多大：

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

稍微修改一下些文件，看会发生些什么：

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ablafe] modified repo a bit
1 files changed, 1 insertions(+), 0 deletions(-)
```

查看这个 **commit** 生成的 **tree**，可以看到一些有趣的东西：

```
$ git cat-file -p master^{tree}

100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

blob 对象与之前的已经不同了。这说明虽然只是往一个 400 行的文件最后加入了一行内容，**Git** 却用一个全新的对象来保存新的文件内容：

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
12908
```

你的磁盘上有了两个几乎完全相同的 12K 的对象。如果 **Git** 只完整保存其中一个，并保存另一个对象的差异内容，岂不更好？

事实上 **Git** 可以那样做。**Git** 往磁盘保存对象时默认使用的格式叫松散对象 (**loose object**) 格式。**Git** 时不时地将这些对象打包至一个叫 **packfile** 的二进制文件以节省空间并提高效率。当仓库中有太多的松散对象，或是手工调用 **git gc** 命令，或推送至远程服务器时，**Git** 都会这样做。手工调用 **git gc** 命令让 **Git** 将库中对象打包并看会发生些什么：

```
$ git gc

Counting objects: 17, done.

Delta compression using 2 threads.

Compressing objects: 100% (13/13), done.

Writing objects: 100% (17/17), done.

Total 17 (delta 1), reused 10 (delta 0)
```

查看一下 **objects** 目录，会发现大部分对象都不在了，与此同时出现了两个新文件：

```
$ find .git/objects -type f

.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

仍保留着的几个对象是未被任何 **commit** 引用的 **blob** —— 在此例中是你之前创建的“what is up, doc?” 和 “test content” 这两个示例 **blob**。你从没将他们添加至任何 **commit**，所以 **Git** 认为它们是“悬空”的，不会将它们打包进 **packfile**。

剩下的文件是新创建的 **packfile** 以及一个索引。**packfile** 文件包含了刚才从文件系统中移除的所有对象。索引文件包含了 **packfile** 的偏移信息，这样就可以快速定位任意一个指定

对象。有意思的是运行 `gc` 命令前磁盘上的对象大小约为 **12K**，而这个新生成的 `packfile` 仅为 **6K** 大小。通过打包对象减少了一半磁盘使用空间。

Git 是如何做到这点的？**Git** 打包对象时，会查找命名及尺寸相近的文件，并只保存文件不同版本之间的差异内容。可以查看一下 `packfile`，观察它是如何节省空间的。`git verify-pack` 命令用于显示已打包的内容：

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree    71  76 5400
05408d195263d853f09dca71d55116663690c27c blob    12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree    106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit  225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob    10  19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree    101 105 5211
484a59275031909e19aad7c92262719cfcdf19a commit  226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob    10  19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag     136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob     7  18 5193 1
05408d195263d853f09dca71d55116663690c27c \
    ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit  226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree    36  46 5316
e3f094f522629ae358806b17daf78246c27c007b blob    1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree    106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob     9  18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit  177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

如果你还记得的话，**9bc1d** 这个 `blob` 是 `repo.rb` 文件的第一个版本，这个 `blob` 引用了 **05408** 这个 `blob`，即该文件的第二个版本。命令输出内容的第三列显示的是对象大小，可以看到**05408** 占用了 **12K** 空间，而 **9bc1d** 仅为 **7** 字节。非常有趣的是第二个版本才是完整保存文件内容的对象，而第一个版本是以差异方式保存的 —— 这是因为大部分情况下需要快速访问文件的最新版本。

最妙的是可以随时进行重新打包。**Git** 自动定期对仓库进行重新打包以节省空间。当然也可以手工运行 `git gc` 命令来这么做。

9.5 The Refspec

这本书读到这里，你已经使用过一些简单的远程分支到本地引用的映射方式了，这种映射可以更为复杂。假设你像这样添加了一项远程仓库：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

它在你的 `.git/config` 文件中添加了一节，指定了远程的名称 (`origin`)，远程仓库的 URL 地址，和用于获取操作的 Refspec：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec 的格式是一个可选的 `+` 号，接着是 `:` 的格式，这里 是远端上的引用格式， 是将要记录在本地的引用格式。可选的 `+` 号告诉 Git 在即使不能快速演进的情况下，也去强制更新它。

缺省情况下 refspec 会被 `git remote add` 命令所自动生成，Git 会获取远端上 `refs/heads/` 下面的所有引用，并将它写入到本地的 `refs/remotes/origin/`。所以，如果远端上有一个 `master` 分支，你在本地可以通过下面这种方式来访问它的历史记录：

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

它们全是等价的，因为 Git 把它们都扩展成 `refs/remotes/origin/master`。

如果你想让 Git 每次只拉取远程的 `master` 分支，而不是远程的所有分支，你可以把 `fetch` 这一行修改成这样：

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

这是 `git fetch` 操作对这个远端的缺省 refspec 值。而如果你只想做一次该操作，也可以在命令行上指定这个 refspec。如可以这样拉取远程的 `master` 分支到本地的 `origin/mymaster` 分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

你也可以在命令行上指定多个 refspec。像这样可以一次获取远程的多个分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster \
    topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected]        master -> origin/mymaster (non fast forward)
```

```
* [new branch]      topic      -> origin/topic
```

在这个例子中，**master** 分支因为不是一个可以快速演进的引用而拉取操作被拒绝。你可以在 **refspec** 之前使用一个 **+** 号来重载这种行为。

你也可以在配置文件中指定多个 **refspec**。如你想在每次获取时都获取 **master** 和 **experiment** 分支，就添加两行：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

但是这里不能使用部分通配符，像这样就是不合法的：

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

但无论如何，你可以使用命名空间来达到这个目的。如你有一个 **QA** 组，他们推送一系列分支，你想每次获取 **master** 分支和 **QA** 组的所有分支，你可以使用这样的配置段落：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

如果你的工作流很复杂，有 **QA** 组推送的分支、开发人员推送的分支、和集成人员推送的分支，并且他们在远程分支上协作，你可以采用这种方式为他们创建各自的命名空间。

推送 Refspec

采用命名空间的方式确实很棒，但 **QA** 组成员第1次是如何将他们的分支推送到 **qa/** 空间里面的呢？答案是你可以使用 **refspec** 来推送。

如果 **QA** 组成员想把他们的 **master** 分支推送到远程的 **qa/master** 分支上，可以这样运行：

```
$ git push origin master:refs/heads/qa/master
```

如果他们想让 **Git** 每次运行 **git push origin** 时都这样自动推送，他们可以在配置文件中添加 **push** 值：

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

这样，就会让 **git push origin** 缺省就把本地的 **master** 分支推送到远程的 **qa/master** 分支

上。

删除引用

你也可以使用 `refspec` 来删除远程的引用，是通过运行这样的命令：

```
$ git push origin :topic
```

因为 `refspec` 的格式是 `:`，通过把 部分留空的方式，这个意思是把远程的 `topic` 分支变成空，也就是删除它。

9.6 传输协议

Git 可以以两种主要的方式跨越两个仓库传输数据：基于 HTTP 协议之上，和 `file://`, `ssh://`, 和 `git://` 等智能传输协议。这一节带你快速浏览这两种主要的协议操作过程。

哑协议

Git 基于 HTTP 之上传输通常被称为哑协议，这是因为它在服务端不需要有针对 Git 特有的代码。这个获取过程仅仅是一系列 GET 请求，客户端可以假定服务端的 Git 仓库中的布局。让我们以 `simplegit` 库来看看 `http-fetch` 的过程：

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

它做的第1件事情就是获取 `info/refs` 文件。这个文件是在服务端运行了 `update-server-info` 所生成的，这也解释了为什么在服务端要想使用 HTTP 传输，必须要开启 `post-receive` 钩子：

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

现在你有一个远端引用和 SHA 值的列表。下一步是寻找 HEAD 引用，这样你就知道了在完成，什么应该被检出到工作目录：

```
=> GET HEAD
ref: refs/heads/master
```

这说明在完成获取后，需要检出 `master` 分支。这时，已经可以开始漫游操作了。因为你的起点是在 `info/refs` 文件中所提到的 `ca82a6 commit` 对象，你的开始操作就是获取它：

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

然后你取回了这个对象 — 这在服务端是一个松散格式的对象，你使用的是静态的 HTTP GET 请求获取的。可以使用 `zlib` 解压缩它，去除其头部，查看它的 `commit` 内容：

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon
```

```
1205815931 -0700 committer Scott Chacon 1240030591 -0700 changed the version
number
```

这样,就得到了两个需要进一步获取的对象 — **cfda3b** 是这个 **commit** 对象所对应的 **tree** 对象, 和 **085bb3** 是它的父对象;

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

这样就取得了这它的下一步 **commit** 对象, 再抓取 **tree** 对象:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops - 看起来这个 **tree** 对象在服务端并不以松散格式对象存在, 所以得到了**404**响应, 代表在 **HTTP** 服务端没有找到该对象。这有好几个原因 — 这个对象可能在替代仓库里面, 或者在打包文件里面, **Git** 会首先检查任何列出的替代仓库:

```
=> GET objects/info/http-alternates
(empty file)
```

如果这返回了几个替代仓库列表, 那么它会去那些地方检查松散格式对象和文件 — 这是一种在软件分叉之间共享对象以节省磁盘的好方法。然而, 在这个例子中, 没有替代仓库。所以你所需要的对象肯定在某个打包文件中。要检查服务端有哪些打包格式文件, 你需要获取 **objects/info/packs** 文件, 这里面包含有打包文件列表 (是的, 它也是被 **update-server-info** 所生成的);

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

这里服务端只有一个打包文件, 所以你要的对象显然就在里面。但是你可以先检查它的索引文件以确认。这在服务端有多个打包文件时也很有用, 因为这样就可以先检查你所需要的对象空间是在哪一个打包文件里面了:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

现在你有了这个打包文件的索引，你可以看看你要的对象是否在里面 — 因为索引文件列出了这个打包文件所包含的所有对象的 **SHA** 值，和该对象存在于打包文件中的偏移量，所以你只需要简单地获取整个打包文件：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

现在你也有了这个 **tree** 对象，你可以继续在 **commit** 对象上漫游。它们全部都在这个你已经下载到的打包文件里面，所以你不用继续向服务端请求更多下载了。在这完成之后，由于下载开始时已探明 **HEAD** 引用是指向 **master** 分支，**Git** 会将它检出到工作目录。

整个过程看起来就像这样：

```
$ git clone http://github.com/schacon/simplegit-progit.git  
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/  
got ca82a6dff817ec66f44342007202690a93763949  
walk ca82a6dff817ec66f44342007202690a93763949  
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Getting alternates list for http://github.com/schacon/simplegit-progit.git  
Getting pack list for http://github.com/schacon/simplegit-progit.git  
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835  
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835  
    which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
walk allbefe06a3f659402fe7563abf99ad00de2209e6
```

智能协议

这个 **HTTP** 方法是很简单但效率不是很高。使用智能协议是传送数据的更常用的方法。这些协议在远端都有 **Git** 智能型进程在服务 — 它可以读出本地数据并计算出客户端所需要的，并生成合适的数据给它，这有两类传输数据的进程：一对用于上传数据和一对用于下载。

上传数据

为了上传数据至远端，**Git** 使用 **send-pack** 和 **receive-pack** 进程。这个 **send-pack** 进程运行在客户端上，它连接至远端运行的 **receive-pack** 进程。

举例来说，你在你的项目上运行了 **git push origin master**，并且 **origin** 被定义为一个使用 **SSH** 协议的 **URL**。**Git** 会使用 **send-pack** 进程，它会启动一个基于 **SSH** 的连接到服务器。它尝试像这样透过 **SSH** 在服务端运行命令：

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"  
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status
```

```
delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

这里的 `git-receive-pack` 命令会立即对它所拥有的每一个引用响应一行 — 在这个例子中，只有 `master` 分支和它的 `SHA` 值。这里第1行也包含了服务端的能力列表（这里是 `report-status` 和 `delete-refs`）。

每一行以4字节的十六进制开始，用于指定整行的长度。你看到第1行以 `005b` 开始，这在十六进制中表示 `91`，意味着第1行有 `91` 字节长。下一行以 `003e` 起始，表示有 `62` 字节长，所以需要读剩下的 `62` 字节。再下一行是 `0000` 开始，表示服务器已完成了引用列表过程。

现在它知道了服务端的状态，你的 `send-pack` 进程会判断哪些 `commit` 是它所拥有但服务端没有的。针对每个引用，这次推送都会告诉对端的 `receive-pack` 这个信息。举例说，如果你在更新 `master` 分支，并且增加 `experiment` 分支，这个 `send-pack` 将会是像这样：

```
0085ca82a6dff817ec66f44342007202690a93763949
15027957951b64cf874c3557a0f3547bd83b3ff6 refs/heads/master report-status
0067000000000000000000000000000000000000000000000000000000000000
cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/heads/experiment
0000
```

这里的全 `'0'` 的 `SHA-1` 值表示之前没有过这个对象 — 因为你是添加新的 `experiment` 引用。如果你在删除一个引用，你会看到相反的：就是右边是全 `'0'`。

`Git` 针对每个引用发送这样一行信息，就是旧的 `SHA` 值，新的 `SHA` 值，和将要更新的引用的名称。第1行还会包含有客户端的能力。下一步，客户端会发送一个所有那些服务端所没有的对象的一个打包文件。最后，服务端以成功(或者失败)来响应：

```
000Aunpack ok
```

下载数据

当你在下载数据时，`fetch-pack` 和 `upload-pack` 进程就起作用了。客户端启动 `fetch-pack` 进程，连接至远端的 `upload-pack` 进程，以协商后续数据传输过程。

在远端仓库有不同的方式启动 `upload-pack` 进程。你可以使用与 `receive-pack` 相同的透过 `SSH` 管道的方式，也可以通过 `Git` 后台来启动这个进程，它默认监听在 `9418` 号端口上。这里 `fetch-pack` 进程在连接后像这样向后台发送数据：

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

它也是以4字节指定后续字节长度的方式开始，然后是要运行的命令，和一个空字节，然后是服务端的主机名，再跟随一个最后的空字节。`Git` 后台进程会检查这个命令是否可以运

行，以及那个仓库是否存在，以及是否具有公开权限。如果所有检查都通过了，它会启动这个 **upload-pack** 进程并将客户端的请求移交给它。

如果你透过 **SSH** 使用获取功能，**fetch-pack** 会像这样运行：

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

不管哪种方式，在 **fetch-pack** 连接之后，**upload-pack** 都会以这种形式返回：

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

这与 **receive-pack** 响应很类似，但是这里指的能力是不同的。而且它还会指出 **HEAD** 引用，让客户端可以检查是否是一份克隆。

在这里，**fetch-pack** 进程检查它自己所拥有的对象和所有它需要的对象，通过发送 “want” 和所需对象的 **SHA** 值，发送 “have” 和所有它已拥有的对象的 **SHA** 值。在列表完成时，再发送 “done” 通知 **upload-pack** 进程开始发送所需对象的打包文件。这个过程看起来像这样：

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

这是传输协议的一个很基础的例子，在更复杂的例子中，客户端可能会支持 **multi_ack** 或者 **side-band** 能力；但是这个例子中展示了智能协议的基本交互过程。

9.7 维护及数据恢复

你时不时的需要进行一些清理工作 —— 如减小一个仓库的大小，清理导入的库，或是恢复丢失的数据。本节将描述这类使用场景。

维护

Git 会不时地自动运行称为 “auto gc” 的命令。大部分情况下该命令什么都不处理。不过要是存在太多松散对象 (loose object, 不在 **packfile** 中的对象) 或 **packfile**, **Git** 会进行调用 **git gc** 命令。**gc** 指垃圾收集 (garbage collect), 此命令会做很多工作：收集所有松散对象并将它们存入 **packfile**, 合并这些 **packfile** 进一个大的 **packfile**, 然后将不被任何 **commit** 引用并且已存在一段时间 (数月) 的对象删除。

可以手工运行 `auto gc` 命令：

```
$ git gc --auto
```

再次强调，这个命令一般什么都不干。如果有 7,000 个左右的松散对象或是 50 个以上的 `packfile`，Git 才会真正调用 `gc` 命令。可能通过修改配置中的 `gc.auto` 和 `gc.autopacklimit` 来调整这两个阈值。

`gc` 还会将所有引用 (`references`) 并入一个单独文件。假设仓库中包含以下分支和标签：

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

这时如果运行 `git gc`，`refs` 下的所有文件都会消失。Git 会将这些文件挪到 `.git/packed-refs` 文件中去以提高效率，该文件是这个样子的：

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

当更新一个引用时，Git 不会修改这个文件，而是在 `refs/heads` 下写入一个新文件。当查找一个引用的 `SHA` 时，Git 首先在 `refs` 目录下查找，如果未找到则到 `packed-refs` 文件中去查找。因此如果在 `refs` 目录下找不到一个引用，该引用可能存到 `packed-refs` 文件中去了。

请注意文件最后以 `^` 开头的那一行。这表示该行上一行的那个标签是一个 `annotated` 标签，而该行正是那个标签所指向的 `commit`。

数据恢复

在使用 Git 的过程中，有时会不小心丢失 `commit` 信息。这一般出现在以下情况：强制删除了一个分支而后又想重新使用这个分支，`hard-reset` 了一个分支从而丢弃了分支的部分 `commit`。如果这真的发生了，有什么办法把丢失的 `commit` 找回来呢？

下面的示例演示了对 `test` 仓库主分支进行 `hard-reset` 到一个老版本的 `commit` 的操作，然后恢复丢失的 `commit`。首先查看一下当前的仓库状态：


```
$ git log --pretty=oneline
ablafef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

接着将 **master** 分支移回至中间的一个 **commit**：

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

这样就丢弃了最新的两个 **commit** —— 包含这两个 **commit** 的分支不存在了。现在要做的是找出最新的那个 **commit** 的 **SHA**，然后添加一个指它它的分支。关键在于找出最新的 **commit** 的 **SHA** —— 你不大可能记住了这个 **SHA**，是吧？

通常最快捷的办法是使用 **git reflog** 工具。当你 (在一个仓库下) 工作时，**Git** 会在你每次修改了 **HEAD** 时悄悄地将改动记录下来。当你提交或修改分支时，**reflog** 就会更新。**git update-ref** 命令也可以更新 **reflog**，这是在本章前面的“**Git References**”部分我们使用该命令而不是手工将 **SHA** 值写入 **ref** 文件的理由。任何时间运行 **git reflog** 命令可以查看当前的状态：

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ablafef HEAD@{1}: ablafef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

可以看到我们签出的两个 **commit**，但没有更多的相关信息。运行 **git log -g** 会输出 **reflog** 的正常日志，从而显示更多有用信息：

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon
```

```
) Reflog message: updating HEAD Author: Scott Chacon Date: Fri May 22 18:22:37
2009 -0700 third commit commit ablafef80fac8e34258ff41fc1b867c702daa24b Reflog:
```

```
HEAD@{1} (Scott Chacon ) Reflog message: updating HEAD Author: Scott Chacon Date:
Fri May 22 18:15:24 2009 -0700 modified repo a bit
```

看起来弄丢了的 **commit** 是底下那个，这样在那个 **commit** 上创建一个新分支就能把它恢复过来。比方说，可以在那个 **commit (ab1afef)** 上创建一个名为 **recover-branch** 的分支：

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

酷！这样有了一个跟原来 **master** 一样的 **recover-branch** 分支，最新的两个 **commit** 又找回来了。接着，假设引起 **commit** 丢失的原因并没有记录在 **reflog** 中 —— 可以通过删除 **recover-branch** 和 **reflog** 来模拟这种情况。这样最新的两个 **commit** 不会被任何东西引用到：

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

因为 **reflog** 数据是保存在 **.git/logs/** 目录下的，这样就没有 **reflog** 了。现在要怎样恢复 **commit** 呢？办法之一是使用 **git fsck** 工具，该工具会检查仓库的数据完整性。如果指定 **--full** 选项，该命令显示所有未被其他对象引用（指向）的所有对象：

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

本例中，可以从 **dangling commit** 找到丢失了的 **commit**。用相同的方法就可以恢复它，即创建一个指向该 **SHA** 的分支。

移除对象

Git 有许多过人之处，不过有一个功能有时却会带来问题：**git clone** 会将包含每一个文件的所有历史版本的整个项目下载下来。如果项目包含的仅仅是源代码的话这并没有什么坏处，毕竟 **Git** 可以非常高效地压缩此类数据。不过如果有人在某个时刻往项目中添加了一个非常大的文件，那们即便他在后来的提交中将此文件删掉了，所有的签出都会下载这个大文件。因为历史记录中引用了这个文件，它会一直存在着。

当你将 **Subversion** 或 **Perforce** 仓库转换导入至 **Git** 时这会成为一个很严重的问题。在此类系统中, (签出时) 不会下载整个仓库历史, 所以这种情形不大会有不良后果。如果你从其他系统导入了一个仓库, 或是发觉一个仓库的尺寸远超出预计, 可以用下面的方法找到并移除 大 (尺寸) 对象。

警告: 此方法会破坏提交历史。为了移除对一个大文件的引用, 从最早包含该引用的 **tree** 对象开始之后的所有 **commit** 对象都会被重写。如果在刚导入一个仓库并在其他人在此基础上开始工作之前这么做, 那没有什么问题 — 否则你不得不通知所有协作者 (贡献者) 去衍合你新修改的 **commit** 。

为了演示这点, 往 **test** 仓库中加入一个大文件, 然后在下次提交时将它删除, 接着找到并将这个文件从仓库中永久删除。首先, 加一个大文件进去:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

喔, 你并不想往项目中加进一个这么大的 **tar** 包。最后还是去掉它:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

对仓库进行 **gc** 操作, 并查看占用了空间:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

可以运行 **count-objects** 以查看使用了多少空间:

```
$ git count-objects -v
count: 4
```

```
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

size-pack 是以千字节为单位表示的 **packfiles** 的大小，因此已经使用了 **2MB**。而在这次提交之前仅用了 **2K** 左右——显然在这次提交时删除文件并没有真正将其从历史记录中删除。每当有人复制这个仓库去取得这个小项目时，都不得不复制所有 **2MB** 数据，而这仅仅因为你曾经不小心加了个大文件。当我们要解决这个问题。

首先要找出这个文件。在本例中，你知道是哪个文件。假设你并不知道这一点，要如何找出哪个（些）文件占用了这么多的空间？如果运行 **git gc**，所有对象会存入一个 **packfile** 文件；运行另一个底层命令 **git verify-pack** 以识别出大对象，对输出的第三列信息即文件大小进行排序，还可以将输出定向到 **tail** 命令，因为你只关心排在最后的那几个最大的文件：

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail
-3
e3f094f522629ae358806b17daf78246c27c007b blob    1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob    12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob    2056716 2056872 5401
```

最底下那个就是那个大文件：**2MB**。要查看这到底是哪个文件，可以使用第 7 章中已经简单使用过的 **rev-list** 命令。若给 **rev-list** 命令传入 **--objects** 选项，它会列出所有 **commit** **SHA** 值，**blob** **SHA** 值及相应的文件路径。可以这样查看 **blob** 的文件名：

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

接下来要将该文件从历史记录的所有 **tree** 中移除。很容易找出哪些 **commit** 修改了这个文件：

```
$ git log --pretty=oneline -- git.tbz2
da3f30d019005479c99eb4c3406225613985aldb oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

必须重写从 **6df76** 开始的所有 **commit** 才能将文件从 **Git** 历史中完全移除。这么做需要用到第 6 章中用过的 **filter-branch** 命令：

```
$ git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
```

```
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2) rm 'git.tbz2'
```

```
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
```

```
Ref 'refs/heads/master' was rewritten
```

--index-filter 选项类似于第 6 章中使用的 **--tree-filter** 选项，但这里不是传入一个命令去修改磁盘上签出的文件，而是修改暂存区域或索引。不能用 **rm file** 命令来删除一个特定文件，而是必须用 **git rm --cached** 来删除它 —— 即从索引而不是磁盘删除它。这样做是出于速度考虑 —— 由于 **Git** 在运行你的 **filter** 之前无需将所有版本签出到磁盘上，这个操作会快得多。也可以用 **--tree-filter** 来完成相同的操作。**git rm** 的 **--ignore-unmatch** 选项指定当你试图删除的内容并不存在时不显示错误。最后，因为你清楚问题是从哪个 **commit** 开始的，使用 **filter-branch** 重写自 **6df7640** 这个 **commit** 开始的所有历史记录。不这么做的话会重写所有历史记录，花费不必要的更多时间。

现在历史记录中已经不包含对那个文件的引用了。不过 **reflog** 以及运行 **filter-branch** 时 **Git** 往 **.git/refs/original** 添加的一些 **refs** 中仍有对它的引用，因此需要将这些引用删除并对仓库进行 **repack** 操作。在进行 **repack** 前需要将所有对这些 **commits** 的引用去除：

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

看一下节省了多少空间。

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

repack 后仓库的大小减小到了 **7K**，远小于之前的 **2MB**。从 **size** 值可以看出大文件对象还在松散对象中，其实并没有消失，不过这没有关系，重要的是在再进行推送或复制，这个对象不会再传出去。如果真的要完全把这个对象删除，可以运行 **git prune --expire** 命

令。

9.8 总结

现在你应该对 **Git** 可以作什么相当了解了，并且在一定程度上也知道了 **Git** 是如何实现的。本章覆盖了许多 **plumbing** 命令 —— 这些命令比较底层，且比你在本书其他部分学到的 **porcelain** 命令要来得简单。从底层了解 **Git** 的工作原理可以帮助你更好地理解为何 **Git** 实现了目前的这些功能，也使你能够针对你的工作流写出自己的工具和脚本。

Git 作为一套 **content-addressable** 的文件系统，是一个非常强大的工具，而不仅仅只是一个 **VCS** 供人使用。希望借助于你新学到的 **Git** 内部原理的知识，你可以实现自己的有趣的应用，并以更高级便利的方式使用 **Git**。