Git全面教程

简介

Git分布式版本管理系统。

Linus在1991年创建了开源的Linux，但是一直没有一个合适的版本管理工具，在2002年以前，世界各地的志愿者都是通过把源代码文件通过diff的方式给Linus，然后他本人通过手工的方式进行合并代码。后来在2002年BitMover公司同意BitKeeper免费给Linux社区使用，但是2005年，社区里的同学们试图破解BitKeeper的协议，被发现后，该公司撤销了他们免费试用的权利，然后Linus用两周时间，自己用c写了一个分布式版本控制工具，从此git就诞生了。。。

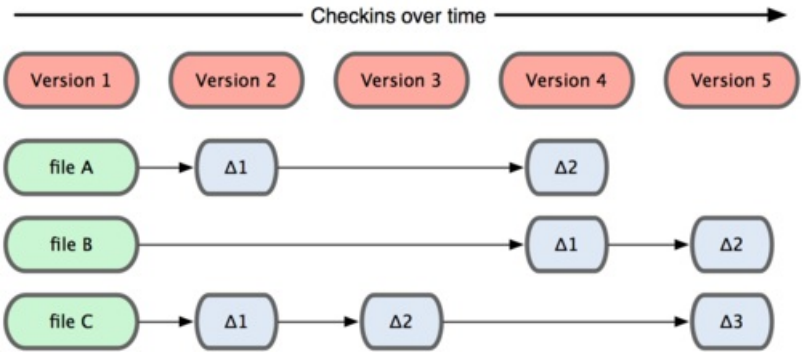
Git设计之初的目标： - 速度 - 简单的设计 - 对非线性开发模式的强力支持（允许上千个并行开发的分支） - 完全分布式 - 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

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，可以应付各种复杂的项目开发需求。

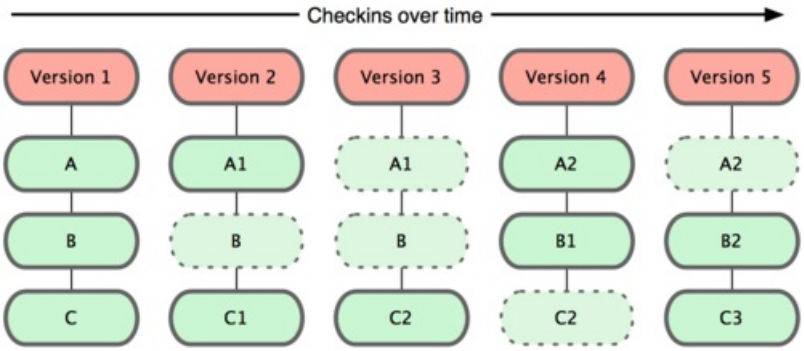
Git和其它版本控制系统的区别

直接记录快找，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，如下图：



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 的工作方式入下图所示：



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。

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

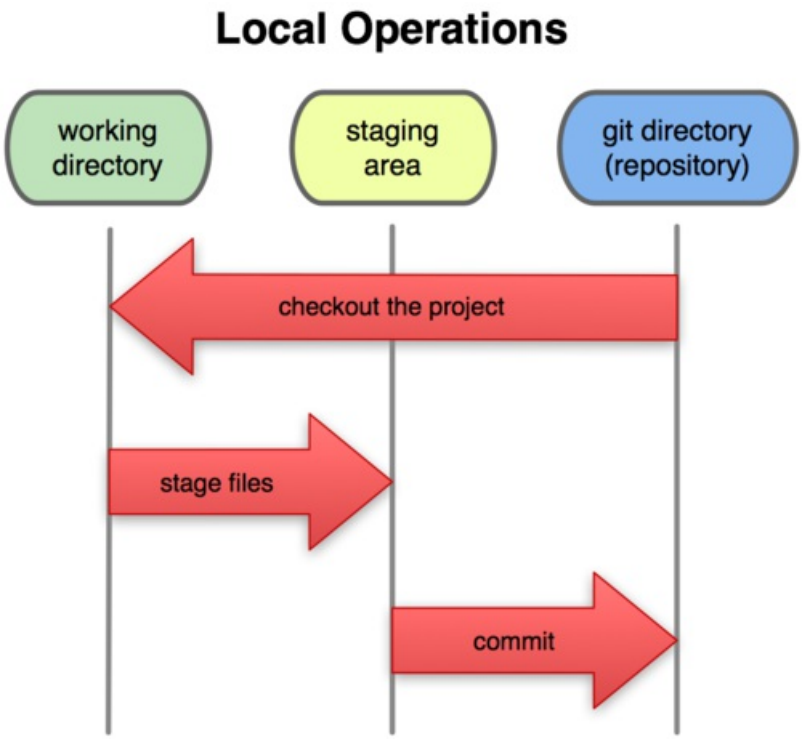
常用的 Git 操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，都会使回退或重现历史版本变得困难重重。在别的 VCS 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 Git 里，一旦提交快照之后就完全不用担心丢失数据，特别是养成定期推送到其他仓库的习惯的话。

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。

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了；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。



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。

Git的配置

一般在新的系统上，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。

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
```

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 <verb>
$ git <verb> --help
$ man git-<verb>
```

比如，要学习 config 命令可以怎么用，运行：

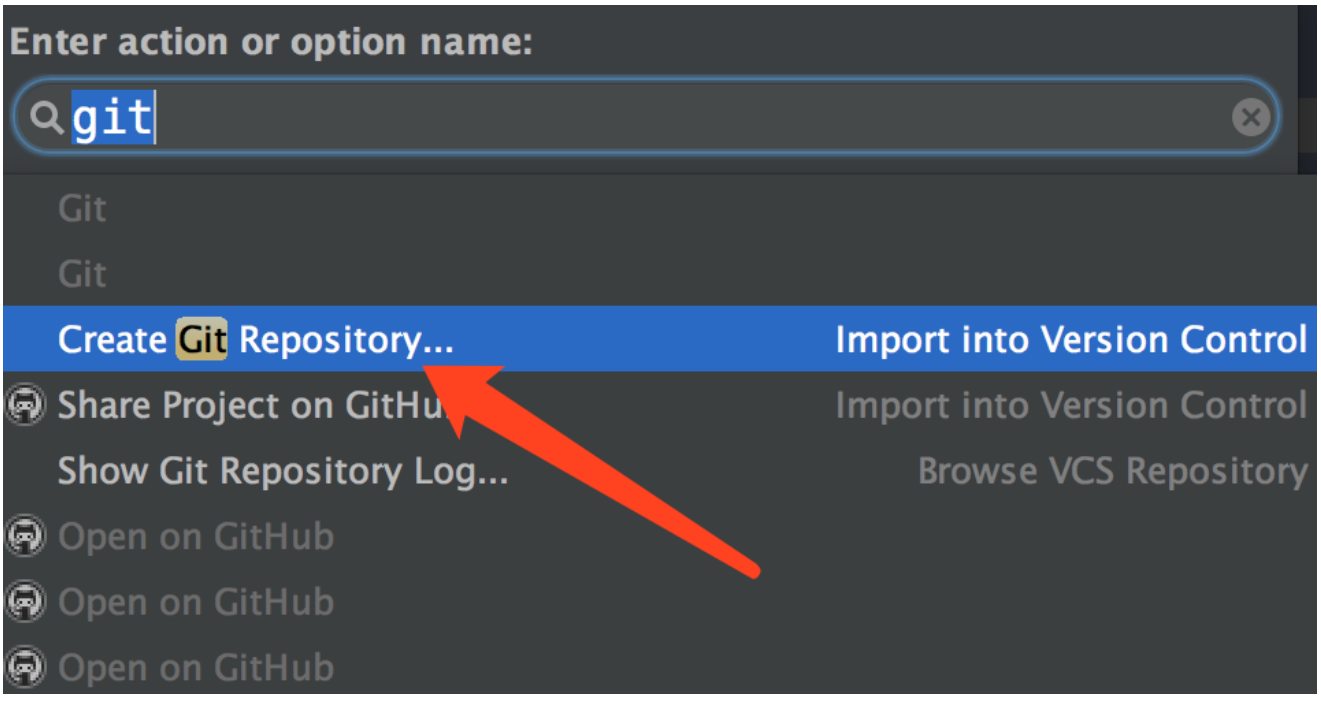
```
$ git help config
```

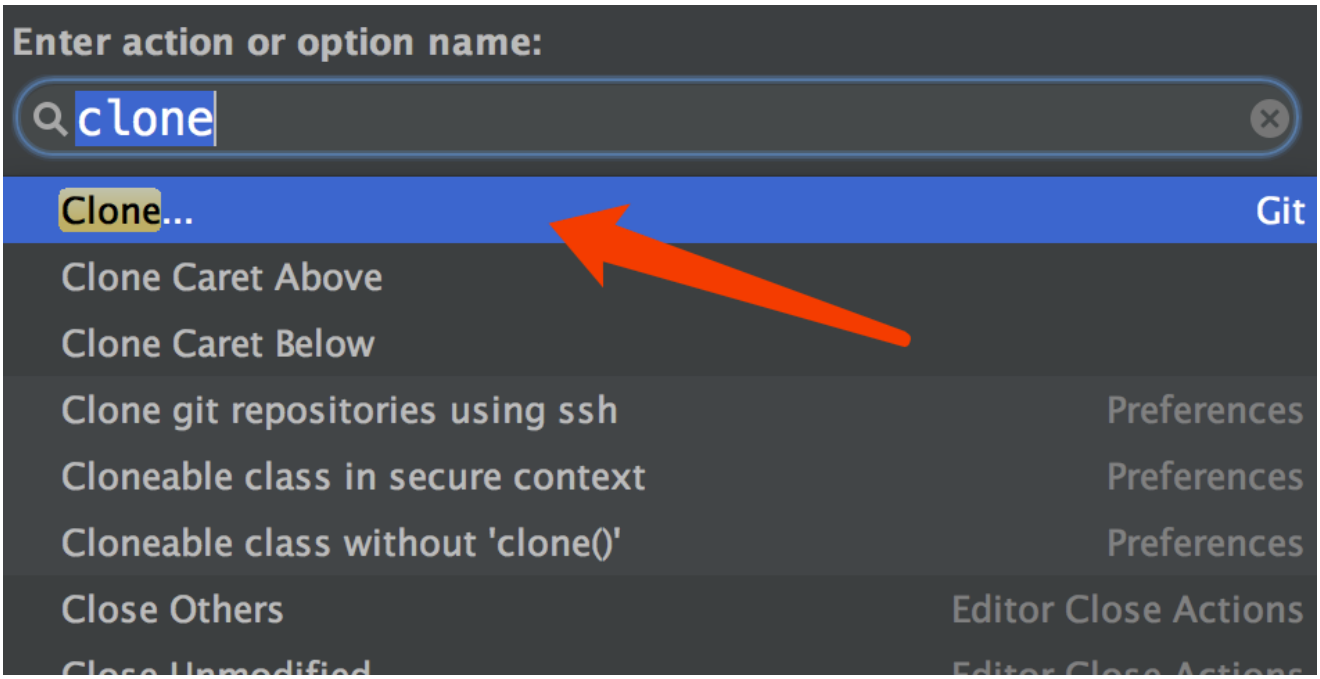
我们随时都可以浏览这些帮助信息而无需连网。 不过，要是你觉得还不够，可以到 Freenode IRC 服务器（irc.freenode.net）上的 `#git` 或 `#github` 频道寻求他人帮助。这两个频道上总有着上百号人，大多都有着丰富的 Git 知识，并且乐于助人。

Git基础

创建版本库

- `git init`
- `git clone`





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。

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：

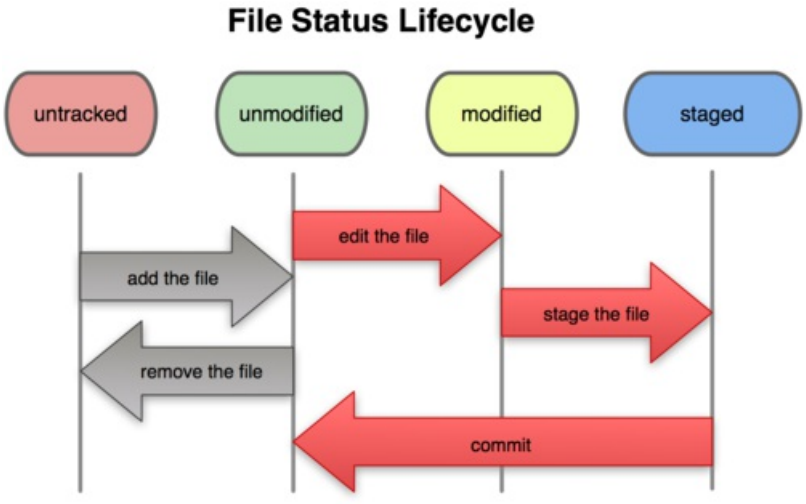
```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

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下图所示：



检查当前文件状态

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
```

这说明你现在的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。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。下一章我们就会详细讨论分支和引用。

现在让我们用 vim 创建一个新文件 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 <file>..." to include in what will be committed)

        README

nothing added to commit but untracked files present (use "git add" to track)
```

在状态报告中可以看到新建的 `README` 文件出现在“Untracked files”下面。未跟踪的文件意味着Git在之前的快照（提交）中没有这些文件；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 <file>..." 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 <file>..." to unstage)

        new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

文件 `benchmarks.rb` 出现在 “Changes not staged for commit” 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 <file>..." 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 <file>..." to unstage)

        new file:   README
        modified:   benchmarks.rb

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   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 <file>..." 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
*.[oa]
*~
```

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
# 忽略所有 .a 结尾的文件
*.a
# 但 lib.a 除外
!lib.a
# 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO
/TODO
# 忽略 build/ 目录下的所有文件
build/
# 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.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` 命令将会看到：

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

        new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   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:
  (use "git reset HEAD <file>..." to unstage)

        modified:   benchmarks.rb

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   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
```

GUI:

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:
#       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 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 3 insertions(+)
 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。

GUI:

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
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+)
```

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 “Changes not staged for commit” 部分（也就是*未暂存*清单）看到：

```
$ rm grit.gemspec
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    grit.gemspec

no changes added to commit (use "git add" and/or "git commit -a")
```

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 <file>..." 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
Changes to be committed:
  (use "git reset HEAD <file>..." 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。

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。

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
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 <schacon@gee-mail.com>
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
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
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
```

该选项除了显示基本信息之外，还在附带了每次 commit 的变化。当进行代码审查，或者快速浏览某个搭档提交的 commit 的变化的时候，这个参数就非常有用了。

某些时候，单词层面的对比，比行层面的对比，更加容易观察。Git 提供了 `--word-diff` 选项。我们可以将其添加到 `git log -p` 命令的后面，从而获取单词层面上的对比。在程序代码中进行单词层面的对比常常是没什么用的。不过当你需要在书籍、论文这种很大的文本文件上进行对比的时候，这个功能就显出用武之地了。下面是一个简单的例子：

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

如你所见，这里并没有平常看到的添加行或者删除行的信息。这里的对比显示在行间。新增加的单词被 `{+ +}` 括起来，被删除的单词被 `[- -]` 括起来。在进行单词层面的对比的时候，你可能希望上下文（ context ）行数从默认的 3 行，减为 1 行，那么可以使用 `-U1` 选项。上面的例子中，我们就使用了这个选项。

另外， `git log` 还提供了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 <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |    6 ++++++
 Rakefile         |   23 +++++++++++++++++++++++
 lib/simplegit.rb |   25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

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）的完整哈希字串
%h  提交对象的简短哈希字串
%T  树对象（tree）的完整哈希字串
%t  树对象的简短哈希字串
%P  父对象（parent）的完整哈希字串
%p  父对象的简短哈希字串
%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 -date= 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。
--word-diff 按 word diff 格式显示差异。
--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）。
--oneline  `--pretty=oneline --abbrev-commit` 的简化用法。
```

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。否则，满足任意一个条件的提交都会被匹配出来）

另一个真正实用的 `git log` 选项是路径(path)，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 detac
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

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，有些撤销操作是不可逆的，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 <file>..." 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 <file>...` 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
Unstaged changes after reset:
M       benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

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：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   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 <file>..." 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。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。

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
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... 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 (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

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些用户的仓库中，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，看看有些什么有趣的更新。

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。如果在你推数据前，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。

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
```

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）。轻量级标签就像是个不会变化的分支，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 <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
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 <schacon@gee-mail.com>"
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 <schacon@gee-mail.com>
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
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

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 <schacon@gee-mail.com>
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 <junkio@cox.net> 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 <junkio@cox.net>"
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 <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
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。

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。

自动补全

如果你用的是 Bash shell，可以试试看 Git 提供的自动补全脚本。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补全脚本，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<tab><tab>
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<tab>` 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<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

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 <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

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 的子命令，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'
```

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。

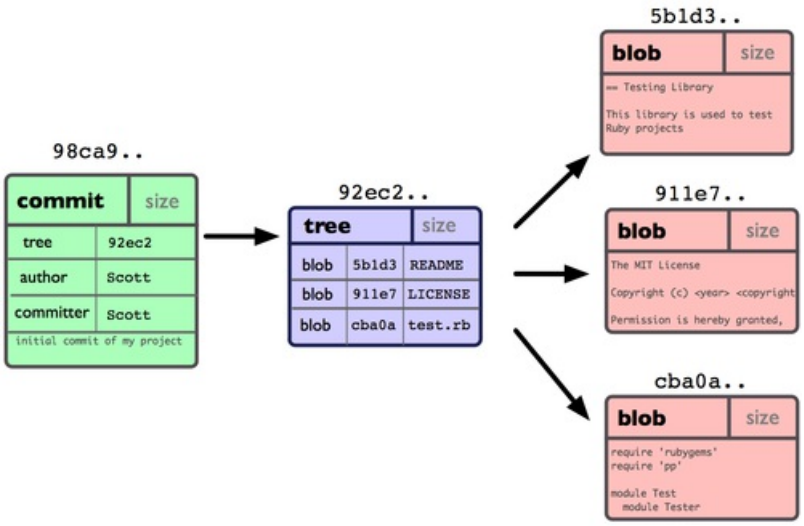
在 Git 中提交时，会保存一个提交（commit）对象，该对象包含一个指向暂存内容快照的指针，包含本次提交的作者等相关附属信息，包含零个或多个指向该提交对象的父对象指针：首次提交是没有直接祖先的，普通提交有一个祖先，由两个或多个分支合并产生的提交则有多个祖先。

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：

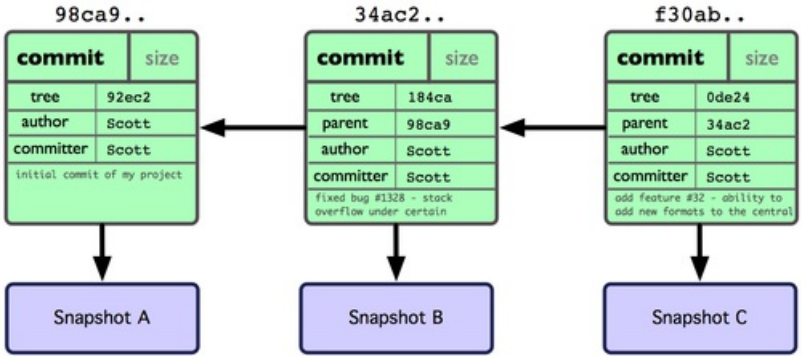
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

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 仓库中将这些目录保存为树（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。

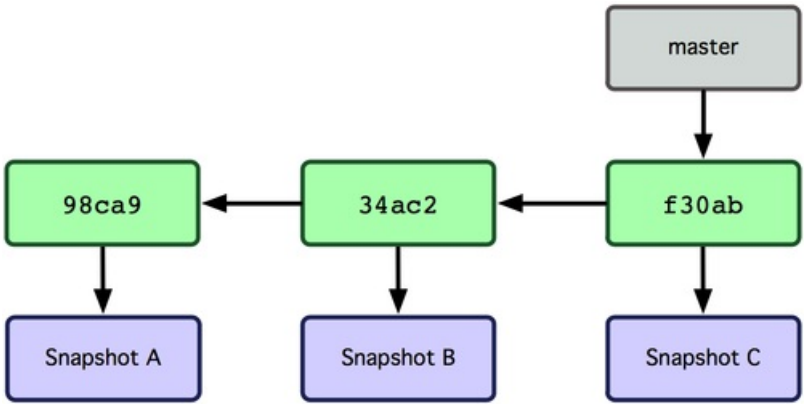
现在，Git 仓库中有五个对象：三个表示文件快照内容的 blob 对象；一个记录着目录树内容及其中各个文件对应 blob 对象索引的 tree 对象；以及一个包含指向 tree 对象（根目录）的索引和其他提交信息元数据的 commit 对象。概念上来说，仓库中的各个对象保存的数据和相互关系看起来如下图所示：



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，仓库历史会变成下图的样子：



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。

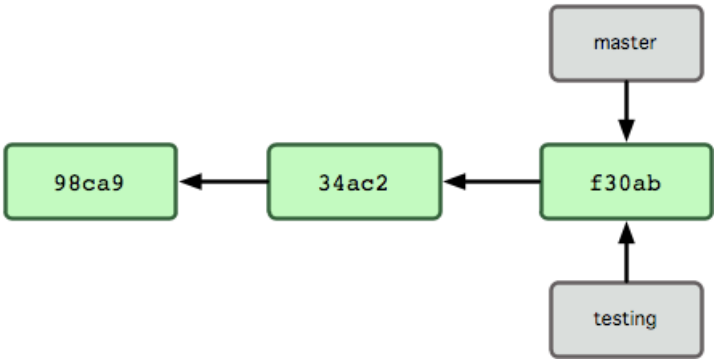


分支其实就是从某个提交对象往回看的历史

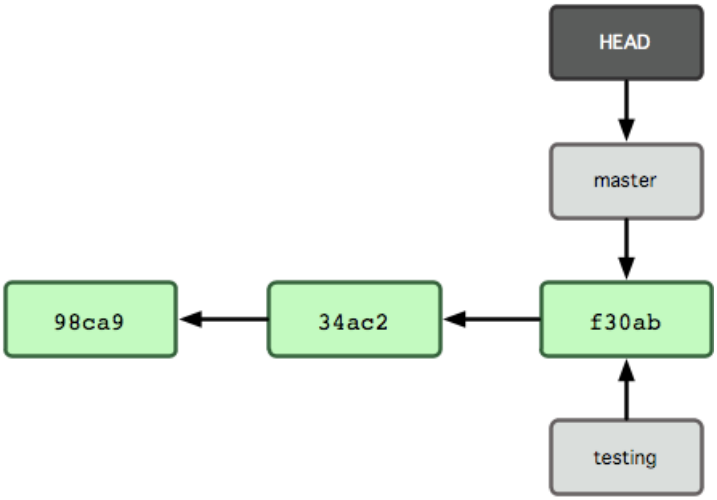
那么，Git 又是如何创建一个新的分支的呢？答案很简单，创建一个新的分支指针。比如新建一个 testing 分支，可以使用 `git branch` 命令：

```
$ git branch testing
```

这会在当前 commit 对象上新建一个分支指针（见下图）。



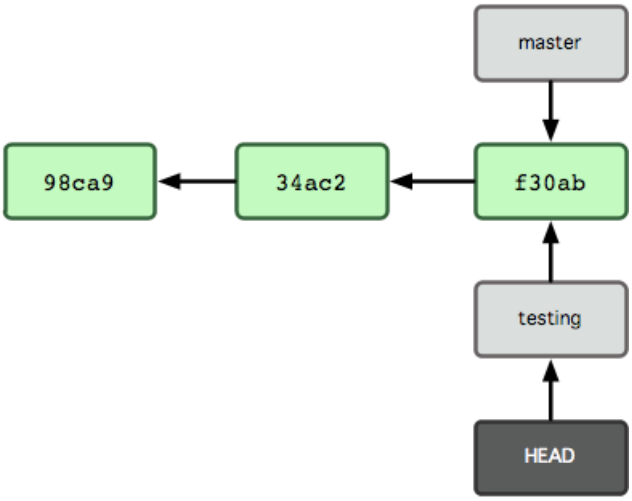
那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HEAD 的特别指针。请注意它和你熟知的许多其他版本控制系统（比如 Subversion 或 CVS）里的 HEAD 概念大不相同。在 Git 中，它是一个指向你正在工作中的本地分支的指针（译注：将 HEAD 想象为当前分支的别名。）。运行 `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 master 分支里工作（见下图）



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
```

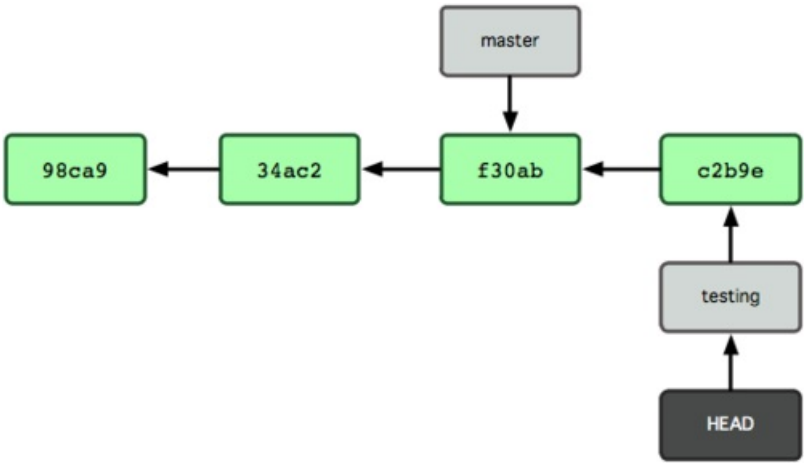
这样 HEAD 就指向了 testing 分支（见下图）。



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：

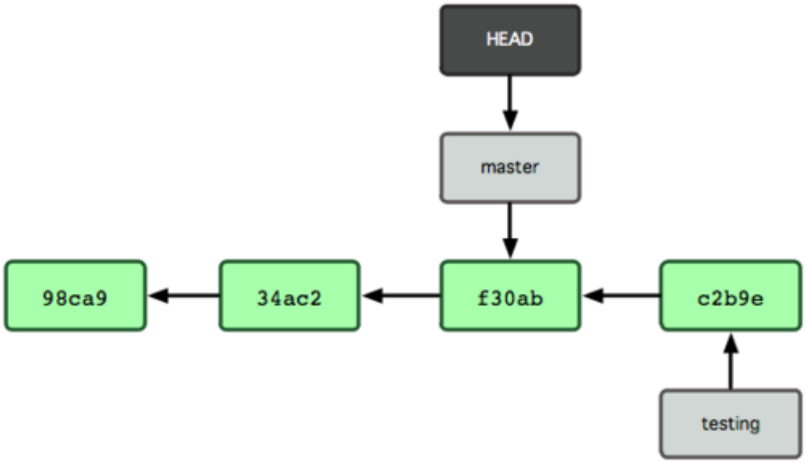
```
$ vim test.rb
$ git commit -a -m 'made a change'
```

下图展示了提交后的结果。



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
```

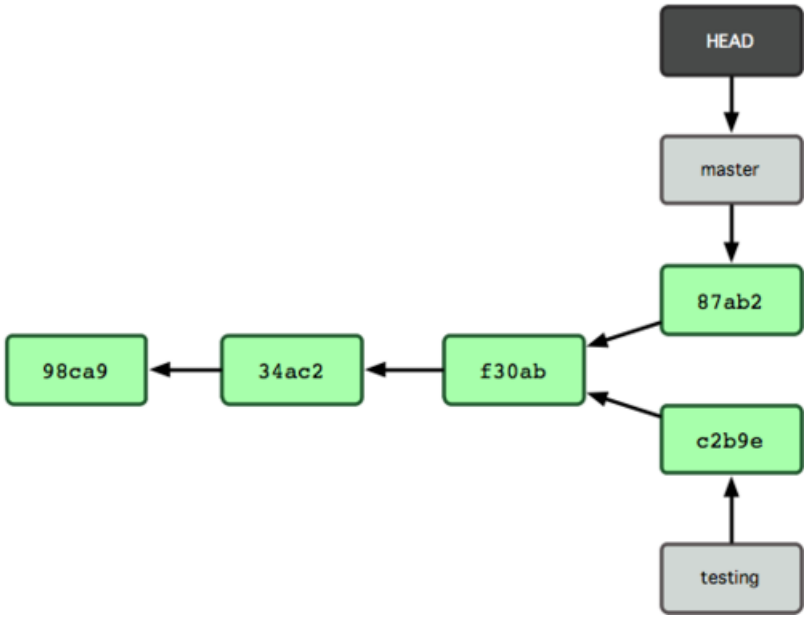


这条命令做了两件事。它把 HEAD 指针移回到 master 分支，并把工作目录中的文件换成了 master 分支所指向的快照内容。也就是说，现在开始所做的改动，将始于本项目中一个较老的版本。它的主要作用是将 testing 分支里作出的修改暂时取消，这样你就可以向另一个方向进行开发。

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。



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。

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。

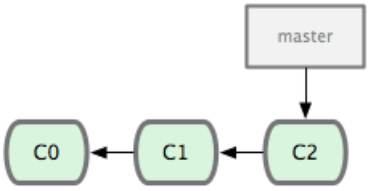
2. 为实现某个新的需求，创建一个分支。
3. 在这个分支上开展工作。

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（见下图）。

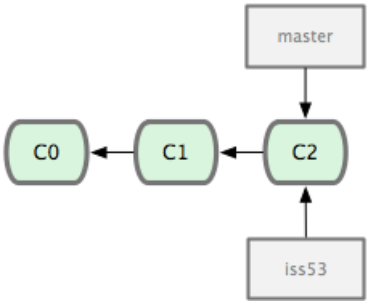


现在，你决定要修补问题追踪系统上的 #53 问题。顺带说明下，Git 并不同任何特定的问题追踪系统打交道。这里为了说明要解决的问题，才把新建的分支取名为 iss53。要新建并切换到该分支，运行 `git checkout` 并加上 `-b` 参数：

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

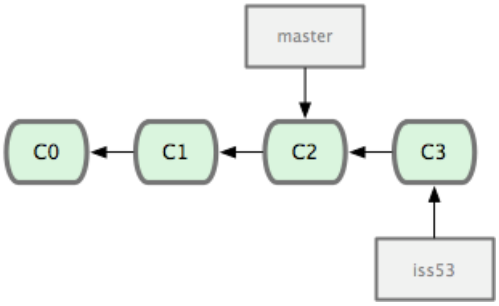
这相当于执行下面这两条命令：

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



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]'
```



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，也不需要在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

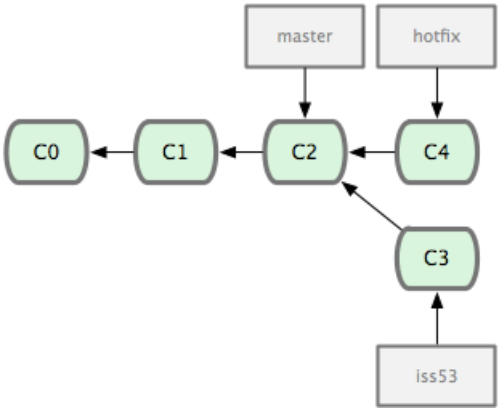
分支。切换分支的时候最好保持一个清洁的工作区域。稍后会介绍几个绕过这种问题的办法（分别叫做 stashing 和 commit amending）。目前已经提交了所有的修改，所以接下来可以正常转换到 `master` 分支：

```
$ 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（见下图）：

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

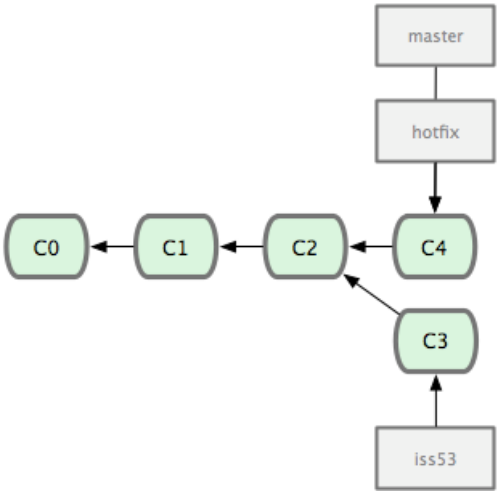


有必要作些测试，确保修补是成功的，然后回到 `master` 分支并把它合并进来，然后发布到生产服务器。用 `git merge` 命令来进行合并：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

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，如果顺着一个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（见下图）。

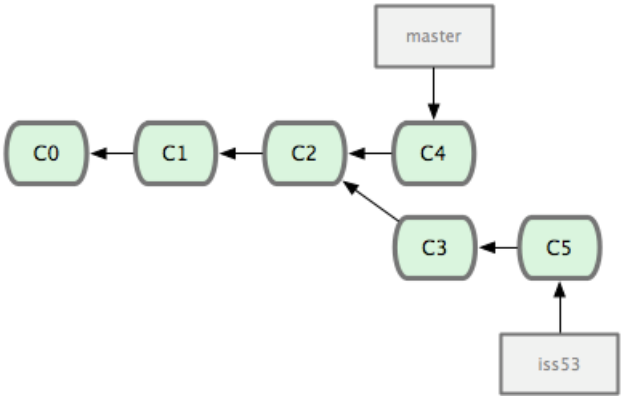


在那个超级重要的修补发布以后，你想要回到被打扰之前的工作。由于当前 `hotfix` 分支和 `master` 都指向相同的提交对象，所以 `hotfix` 已经完成了历史使命，可以删掉了。使用 `git branch` 的 `-d` 选项执行删除操作：

```
$ git branch -d hotfix
Deleted branch hotfix (was 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 ad82d7a] finished the new footer [issue 53]
 1 file changed, 1 insertion(+)
```



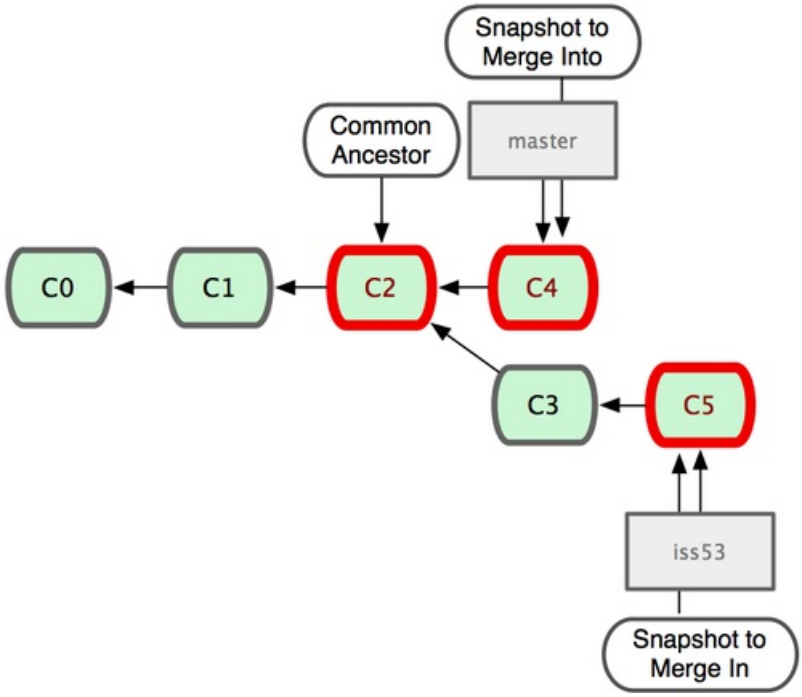
值得注意的是之前 `hotfix` 分支的修改内容尚未包含到 `iss53` 中来。如果需要纳入此次修补，可以用 `git merge master` 把 master 分支合并到 `iss53`；或者等 `iss53` 完成之后，再将 `iss53` 分支中的更新并入 `master`。

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：

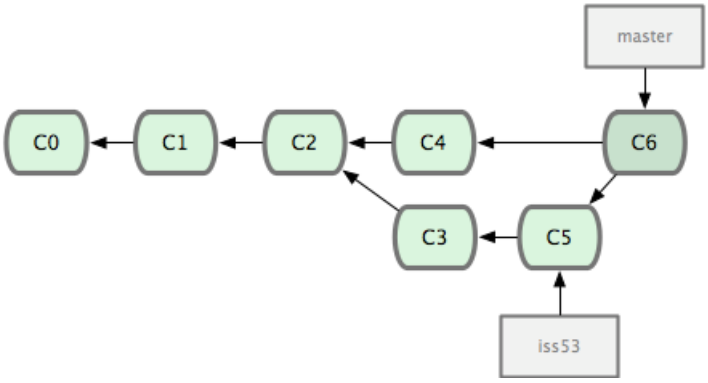
```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

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：



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。



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：

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

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
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

可以看到 `=======` 隔开的上半部分，是 `HEAD`（即 `master` 分支，在运行 `merge` 命令时所切换到的分支）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：

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

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

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
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 <file>..." 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。

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 `--merged` 和 `--no-merged` 选项（Git 1.5.6 以上版本）。比如用 `git branch --merged` 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 fully merged.
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。

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）。

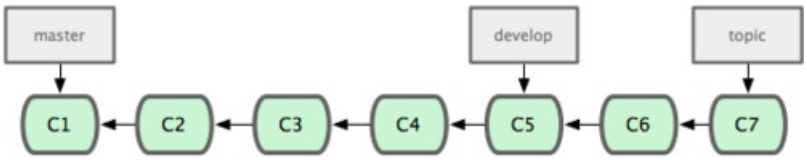


图 3-18. 稳定分支总是比较老旧。

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）。

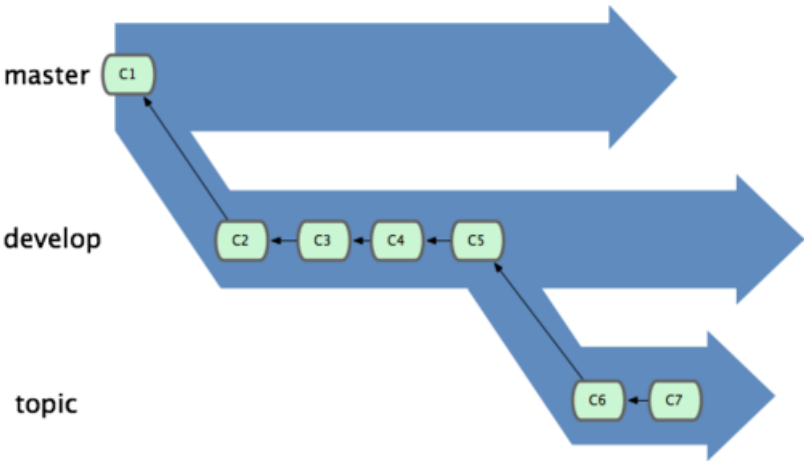


图 3-19. 想象成流水线可能会容易点。

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。

现在我们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。

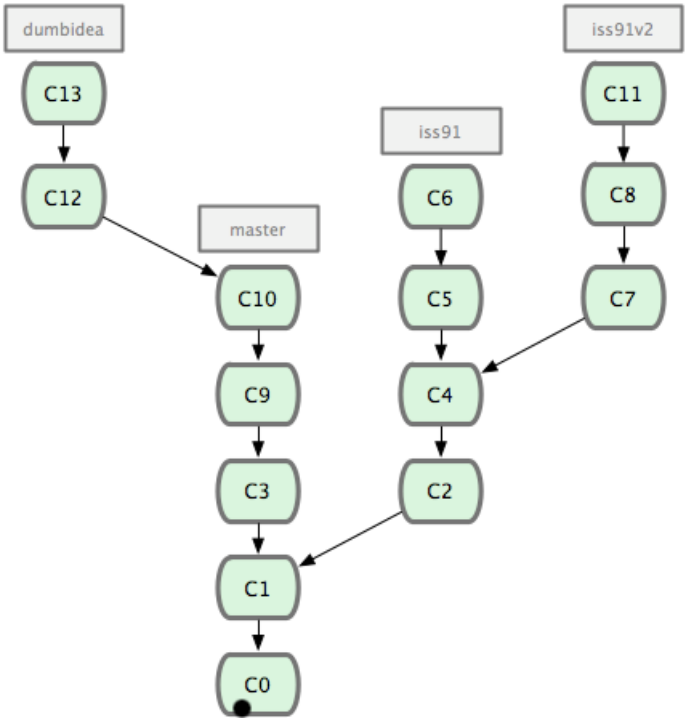


图 3-20. 拥有多个特性分支的提交历史。

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：

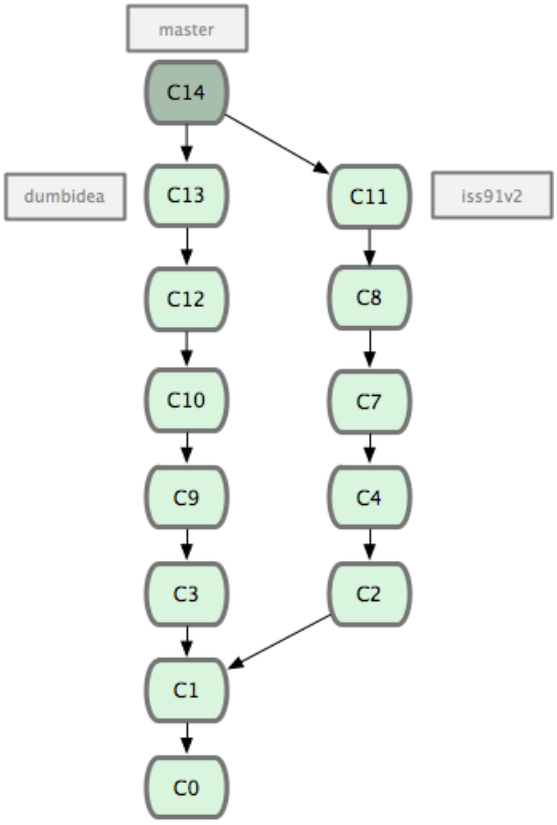


图 3-21. 合并了 dumbidea 和 iss91v2 后的分支历史。

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。

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` 分支的样子，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却应该是 `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）：

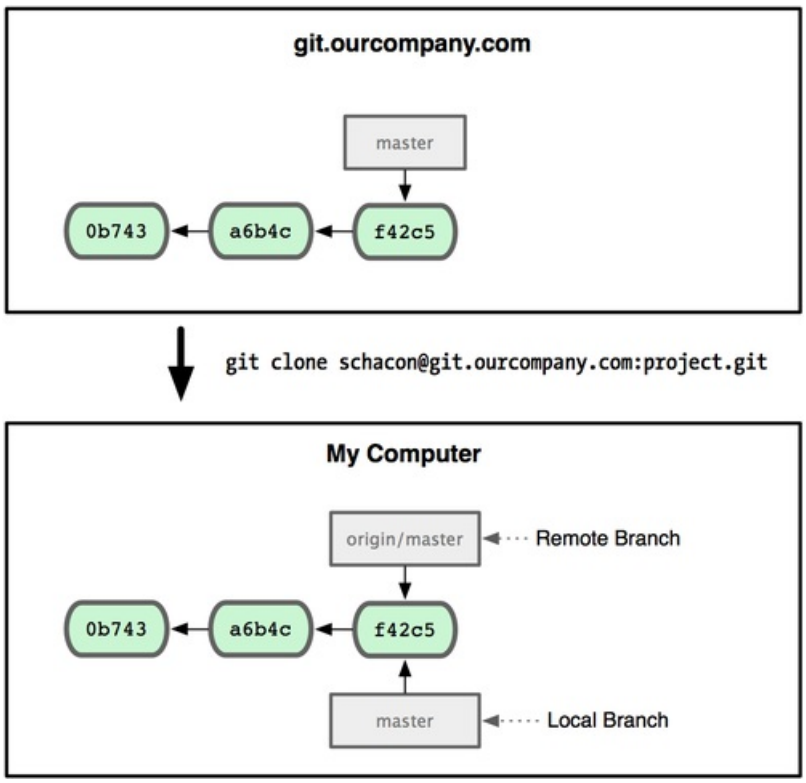


图 3-22. 一次 Git 克隆会建立你自己的本地分支 master 和远程分支 origin/master，并且将它们都指向 `origin` 上的 `master` 分支。

如果你在本地 `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，而与此同时，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）。

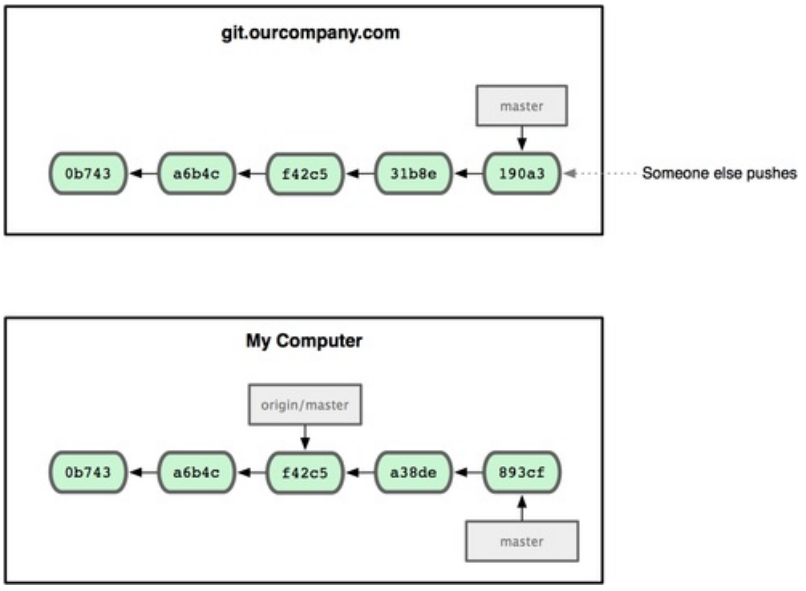


图 3-23. 在本地工作的同时有人向远程仓库推送内容会让提交历史开始分流。

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）。

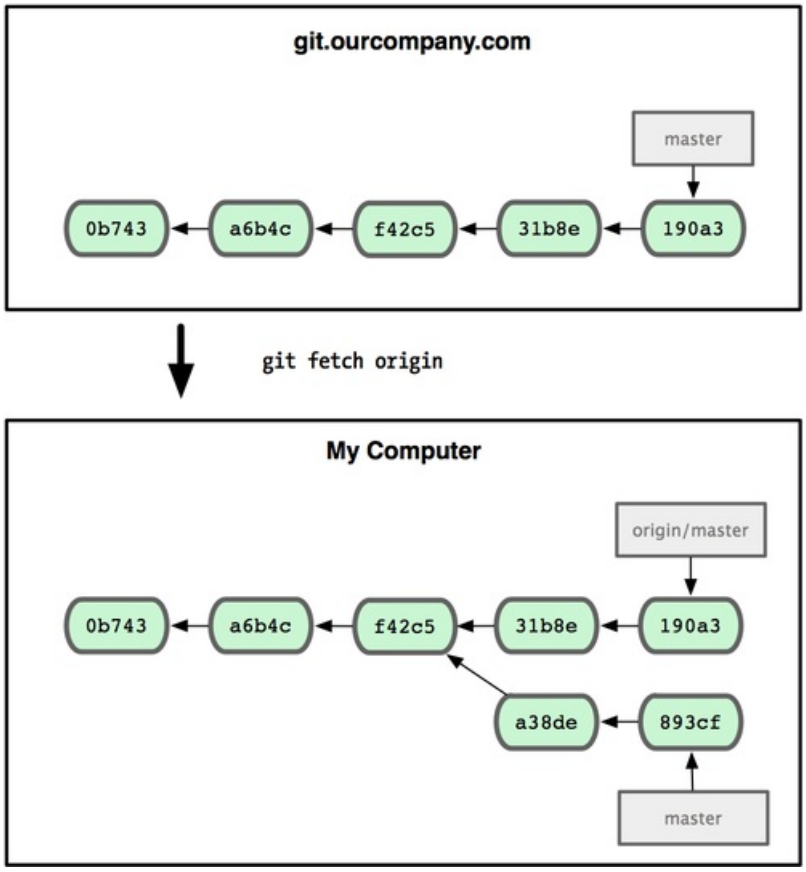


图 3-24. `git fetch` 命令会更新 remote 索引。

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`，以便代替完整的 Git URL 以方便使用（见图 3-25）。

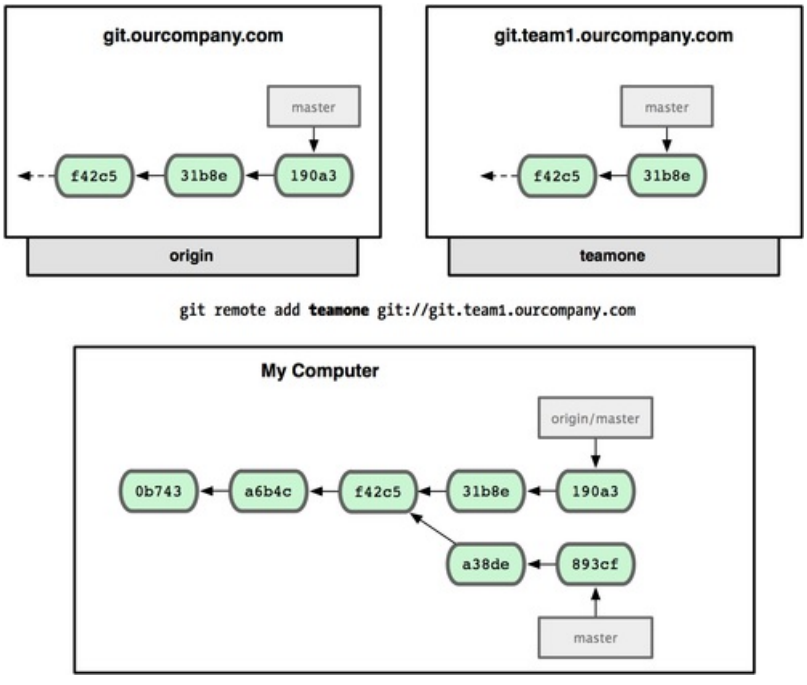


图 3-25. 把另一个服务器加为远程仓库

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` 的远程分支，指向 `teamone` 服务器上 `master` 分支所在的提交对象 `31b8e`（见图 3-26）。

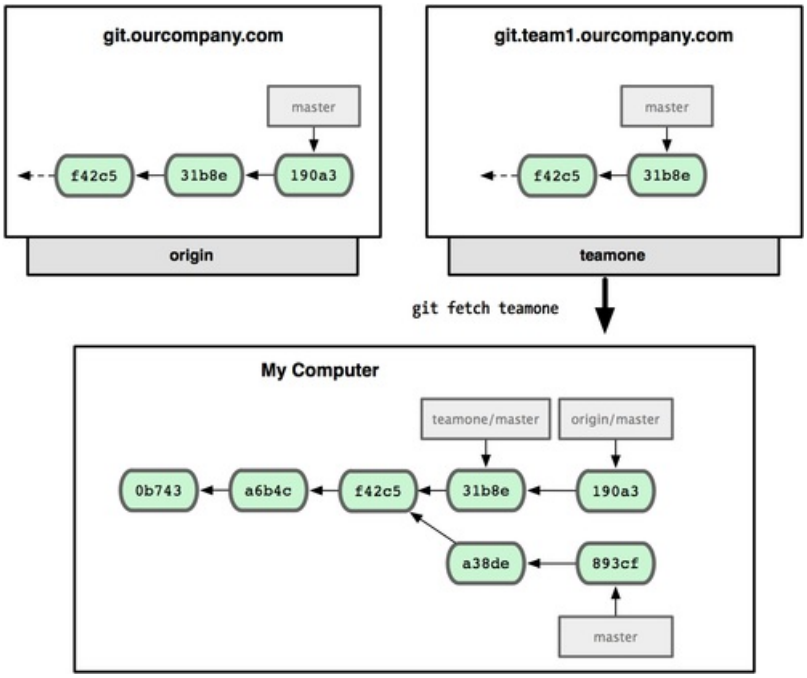


图 3-26. 你在本地有了一个指向 teamone 服务器上 master 分支的索引。

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。你创建的本地分支不会因为你的写入操作而被自动同步到你引入的远程服务器上，你需要明确地执行推送分支的操作。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
```

这里其实走了一点捷径。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`，并指向服务器上 `serverfix` 所指向的版本：

```
$ 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。

如果要把该远程分支的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 serverfix from origin.
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)。跟踪分支是一种和某个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。同样，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 通常会自动创建一个名为 `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一个例子：`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 serverfix from origin.
Switched to a new branch 'serverfix'
```

要为本地分支设定不同于远程分支的名字，只需在第一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 serverfix from origin.
Switched to a new branch 'sf'
```

现在你的本地分支 `sf` 会自动将推送和抓取数据的位置定位到 `origin/serverfix` 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分支），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，而且你很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]`”。

分支的衍合

把一个分支中的修改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。

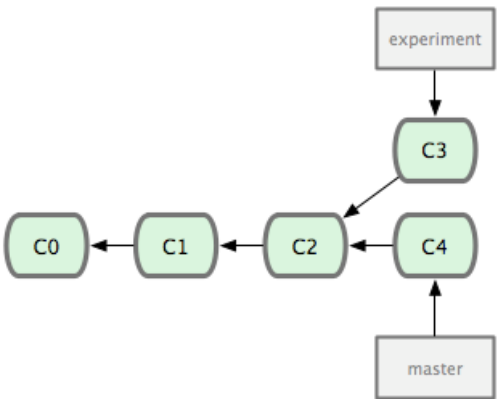


图 3-27. 最初分叉的提交历史。

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：

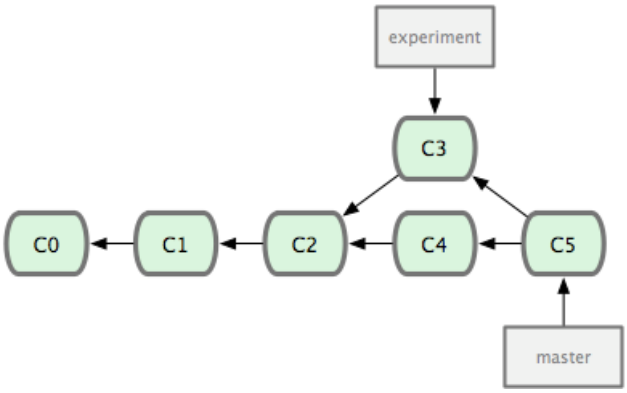


图 3-28. 通过合并一个分支来整合分叉了的历史。

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：

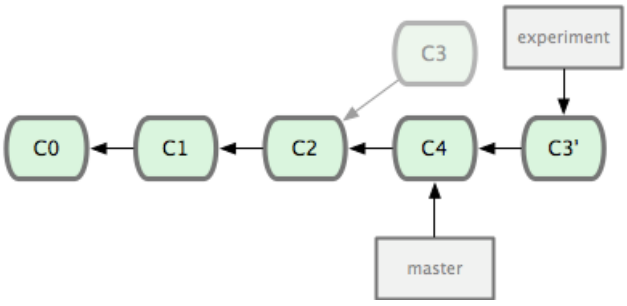


图 3-29. 把 C3 里产生的改变到 C4 上重演一遍。

现在回到 `master` 分支，进行一次快进合并（见图 3-30）：

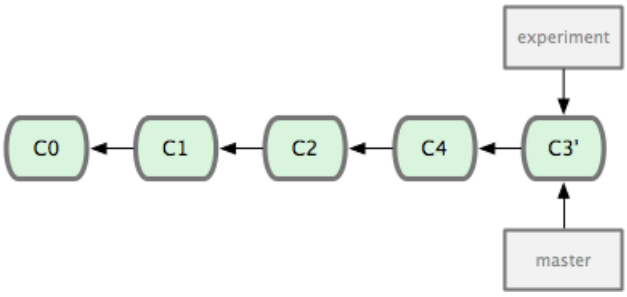


图 3-30. master 分支的快进。

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进合并，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。

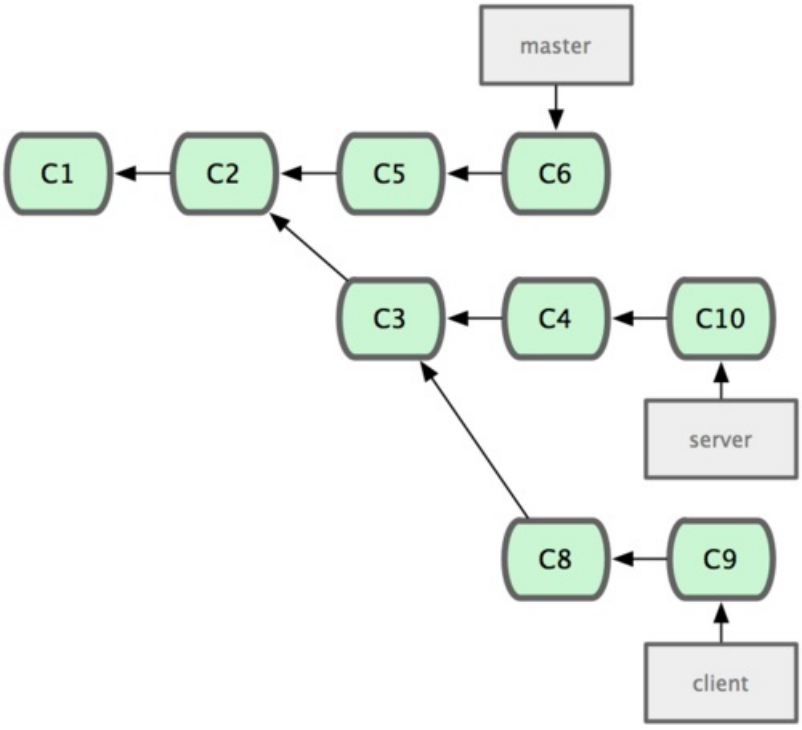


图 3-31. 从一个特性分支里再分出一个特性分支的历史。

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 `client` 分支而非 `server` 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。）：

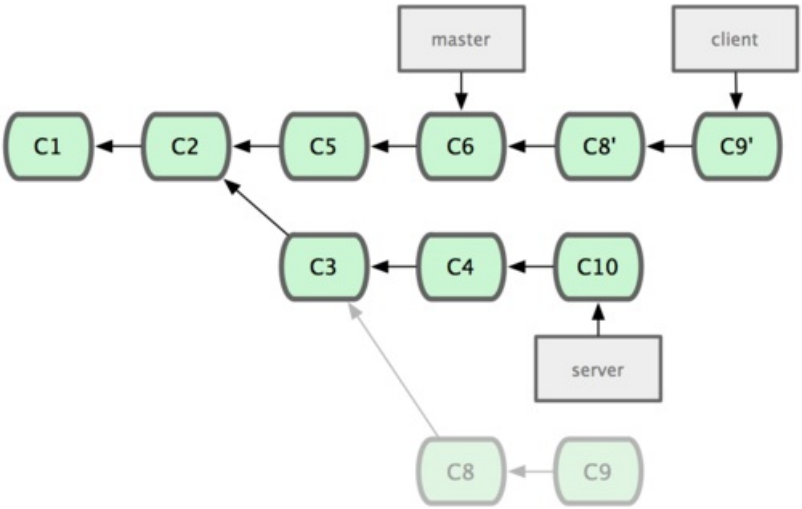


图 3-32. 将特性分支上的另一个特性分支衍合到其他分支。

现在可以快进 `master` 分支了（见图 3-33）：

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

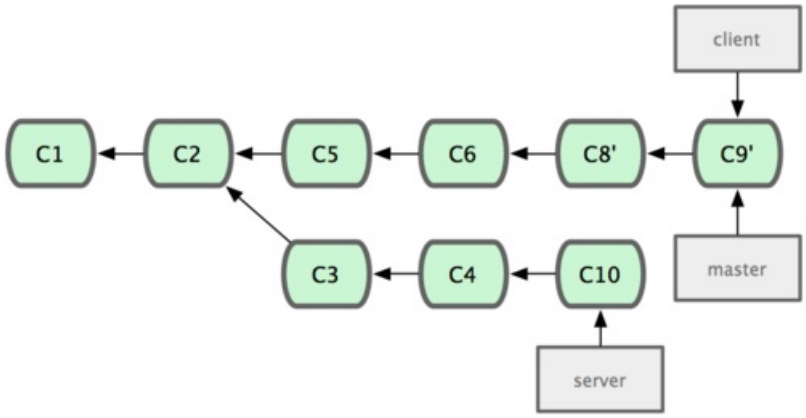


图 3-33. 快进 master 分支，使之包含 client 分支的变化。

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：

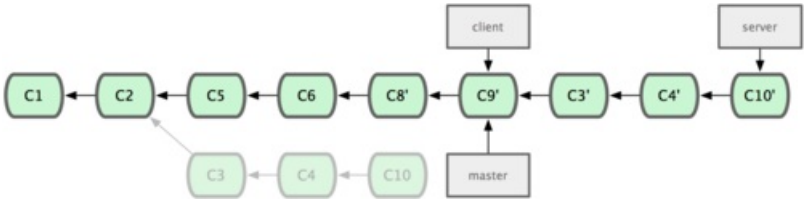


图 3-34. 在 master 分支上衍合 server 分支。

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
```

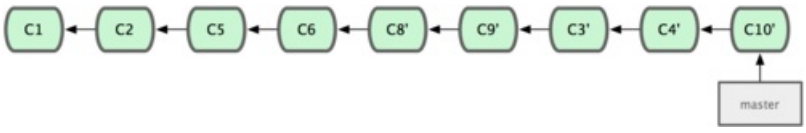


图 3-34. 在 master 分支上衍合 server 分支。

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
```

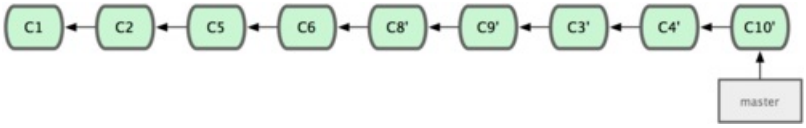


图 3-35. 最终的提交历史

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。

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）。

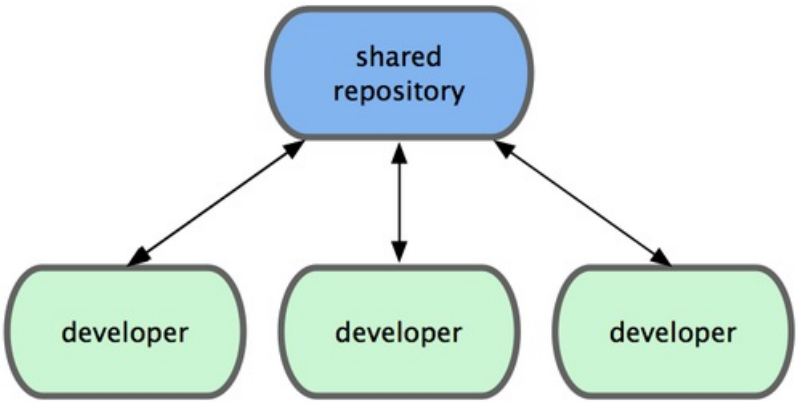


图 5-1. 集中式工作流

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，而同时又可以从别人的仓库中提取他们的更新过来。这种情形通常都会有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. 贡献者推送数据到自己的公共仓库 developer public。
4. 贡献者给维护者发送邮件，请求拉取自己的最新修订。
5. 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。
6. 维护者将合并后的更新推送到主仓库 blessed repository。

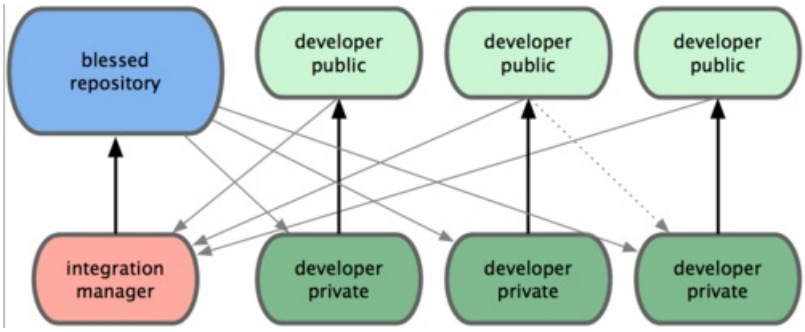


图 5-2. 集成管理员工作流

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：

1. 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。
2. 副官（lieutenant）将普通开发者的特性分支合并到自己的 master 分支中。
3. 司令官（dictator）将所有副官的 master 分支并入自己的 master 分支。
4. 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。

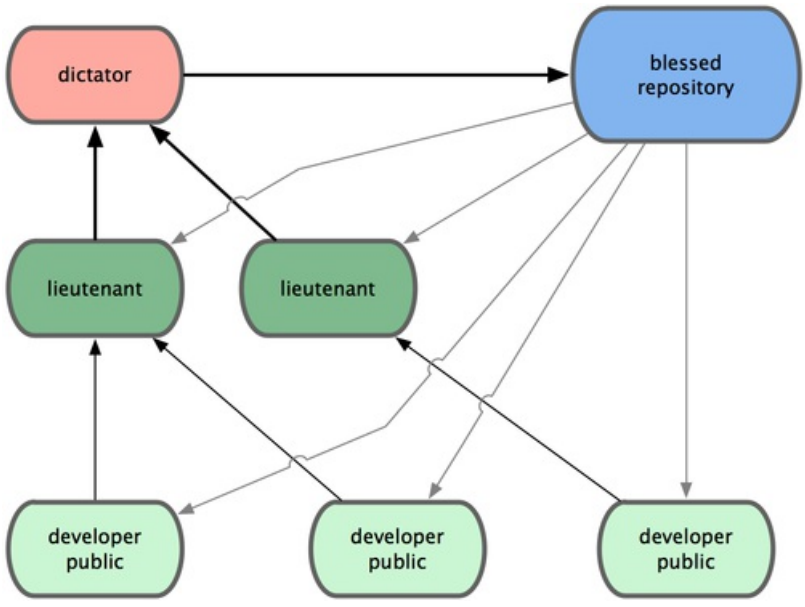


图 5-3. 司令官与副官工作流

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，看看各式工作流

中的每个角色具体应该如何操作。

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。可要是在大公司里，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工作流。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工作流。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工作流。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者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.
+ XXXXXXXXXXX
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@githost: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@githost: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@githost: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@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost: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@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

此刻，John 的本地仓库如图 5-4 所示：

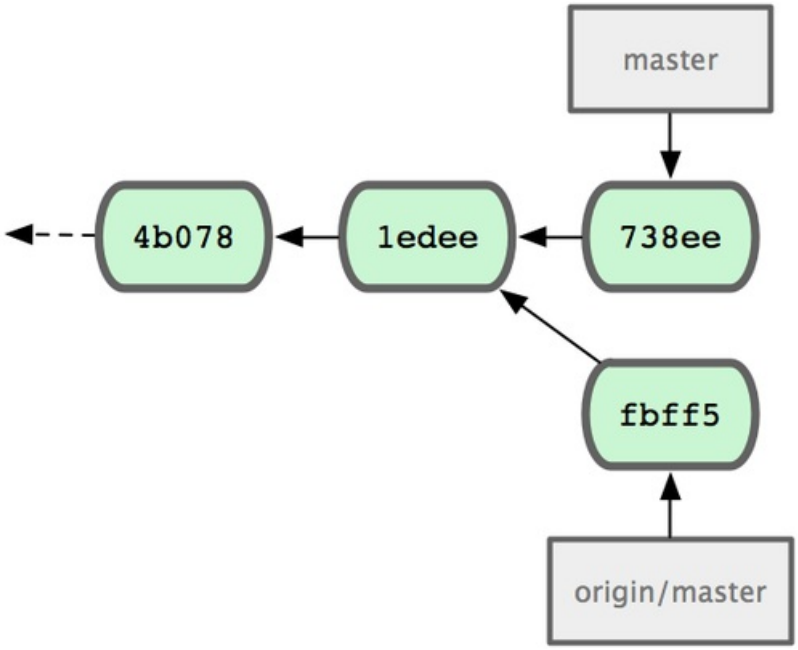


图 5-4. John 的仓库历史

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：

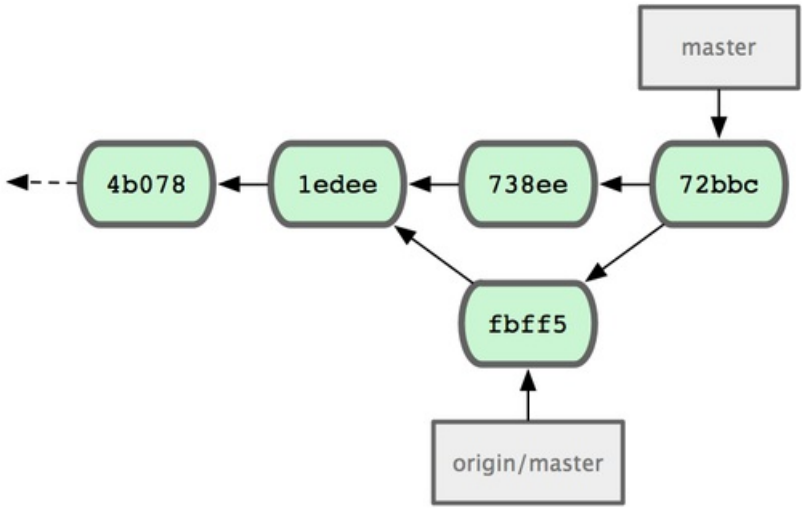


图 5-5. 合并 origin/master 后 John 的仓库历史

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@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

最终，John 的提交历史变为图 5-6 所示：

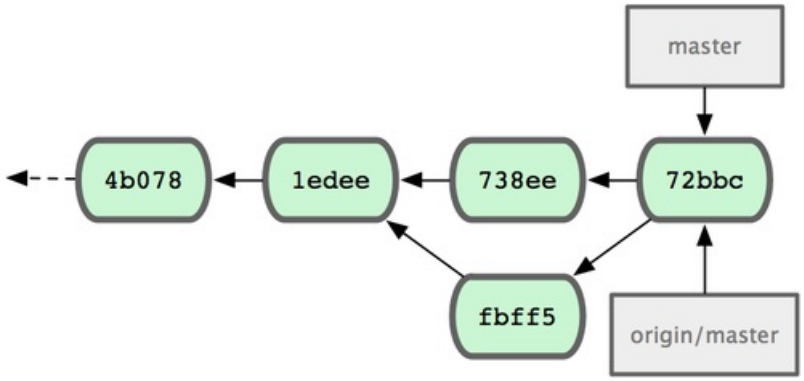


图 5-6. 推送后 John 的仓库历史

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` 并提交了三次更新。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：

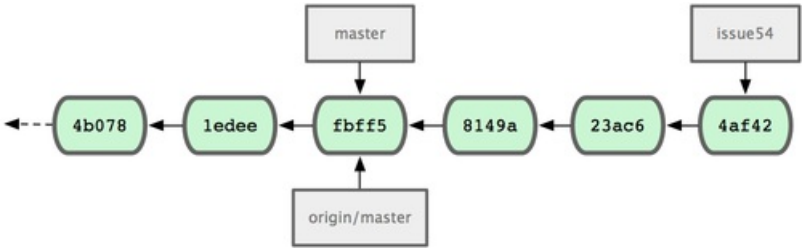


图 5-7. Jessica 的提交历史

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@githost: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：

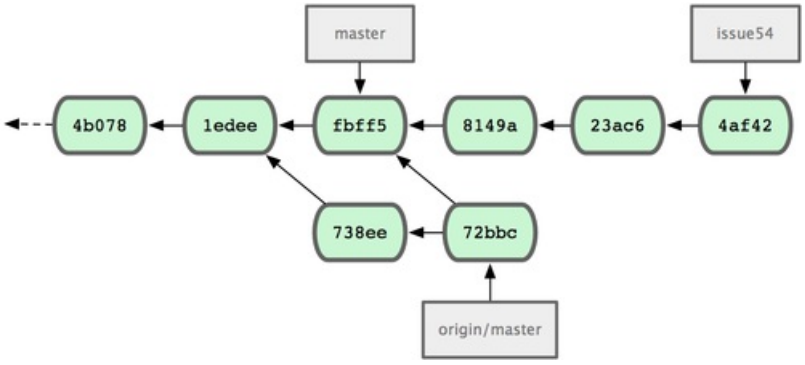


图 5-8. 获取 John 的更新之后 Jessica 的提交历史

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 <jsmith@example.com>
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(-)
```

正如所见，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：

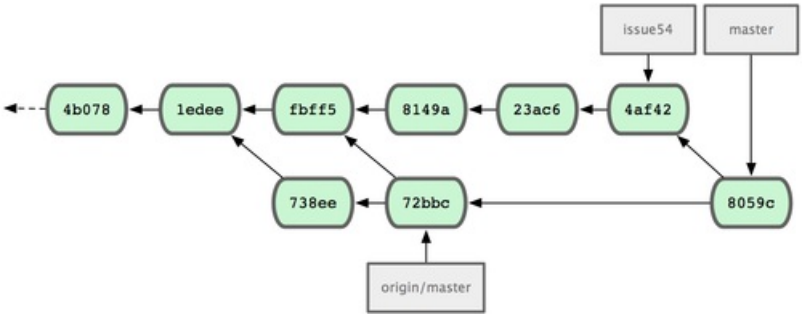


图 5-9. 合并 John 的更新后 Jessica 的提交历史

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@githost: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：

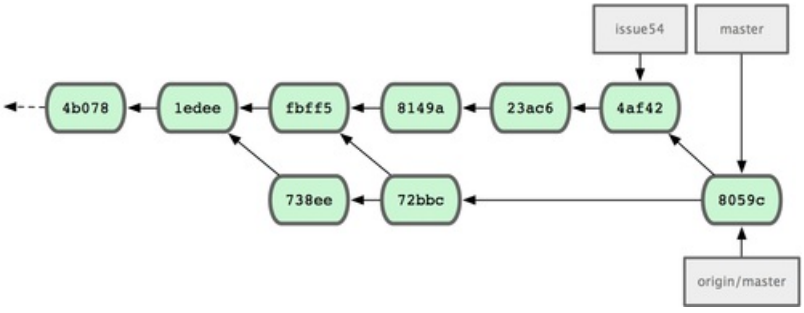


图 5-10. Jessica 推送数据后的提交历史

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：

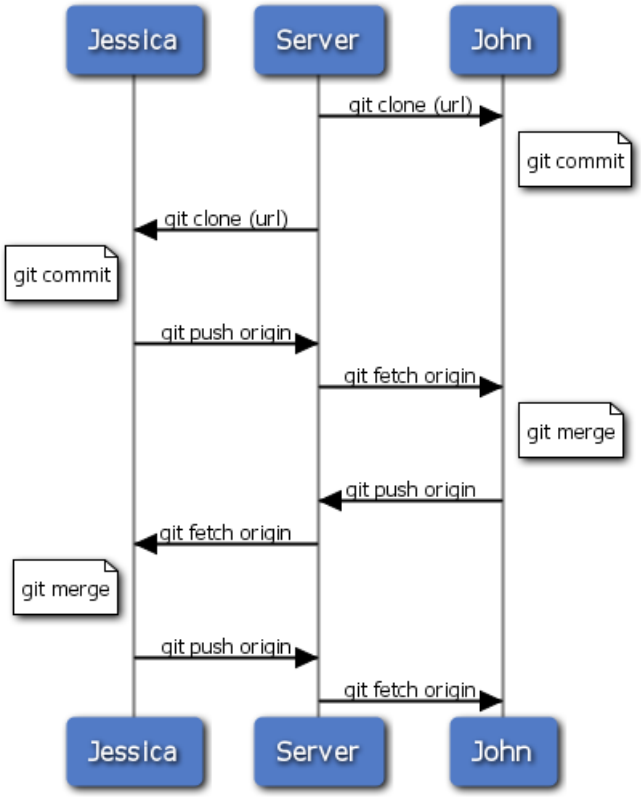


图 5-11. 多用户共享仓库协作方式的一般工作流程时序

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@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica 发邮件给 John 让他上来看看 `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：

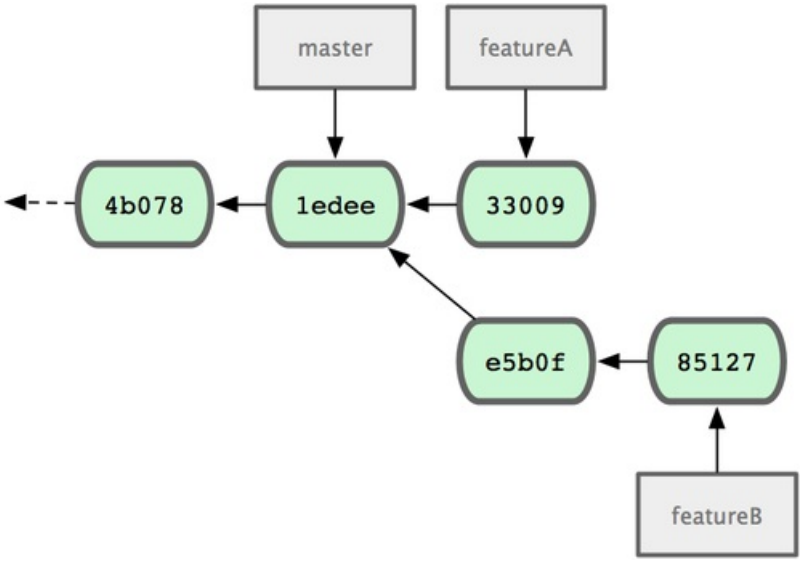


图 5-12. Jessica 的更新历史

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@githost: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@githost: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@githost: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 <jsmith@example.com>
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 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

现在的 Jessica 提交历史如图 5-13 所示：

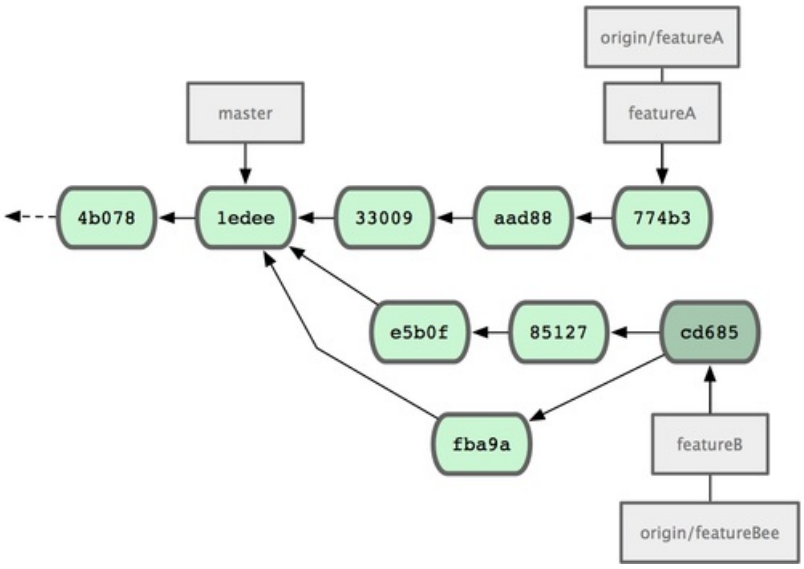


图 5-13. 在特性分支中提交更新后的提交历史

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：

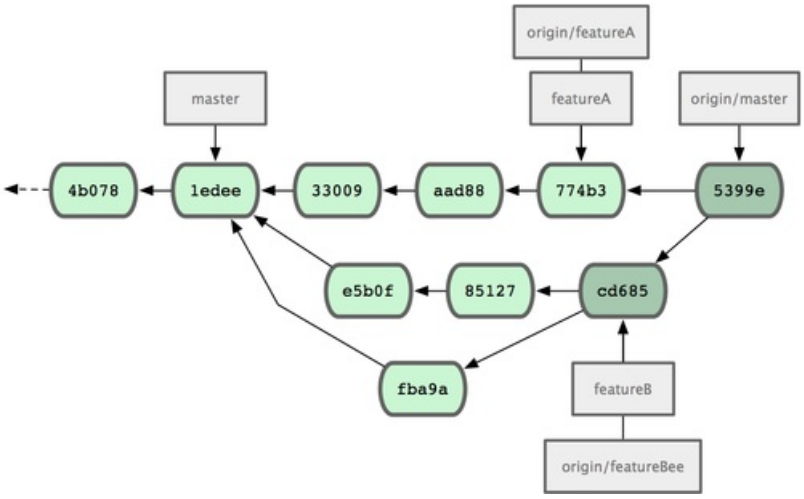


图 5-14. 合并特性分支后的 Jessica 提交历史

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：

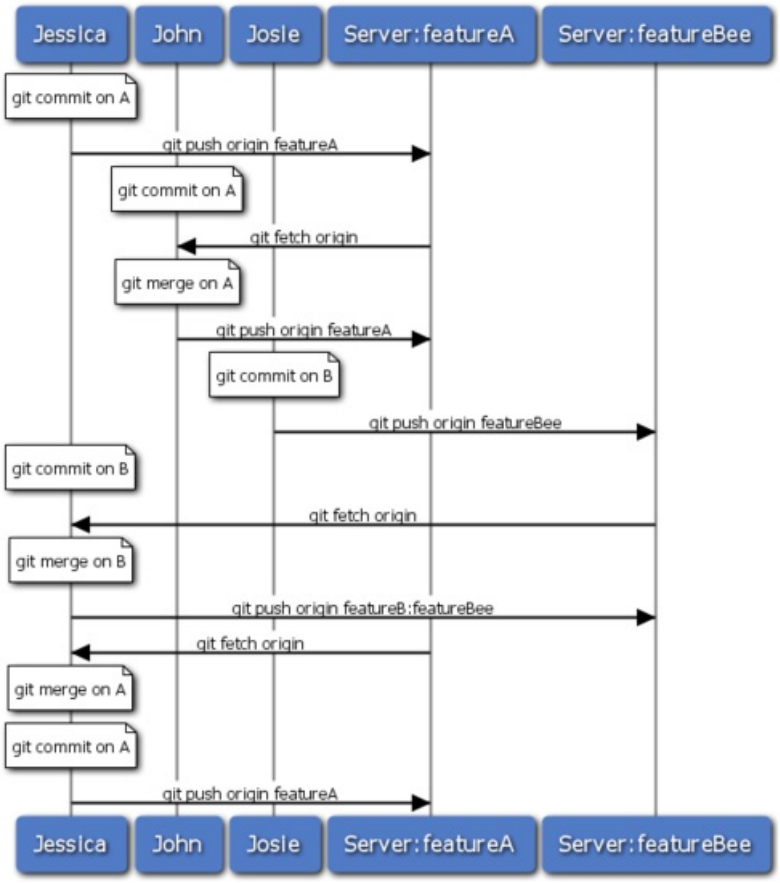


图 5-15. 团队间协作工作流程基本时序

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 requst，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：

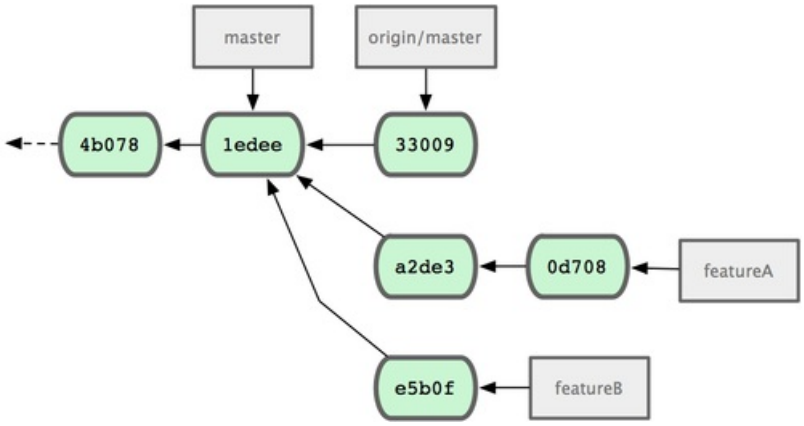


图 5-16. featureB 以后的提交历史

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：

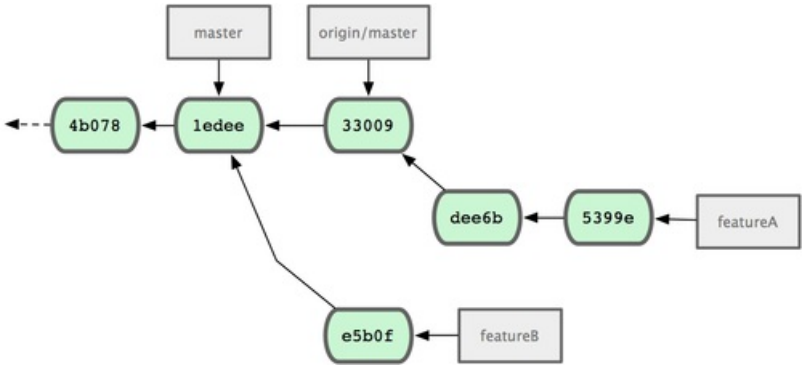


图 5-17. featureA 重新衍合后的提交历史

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：

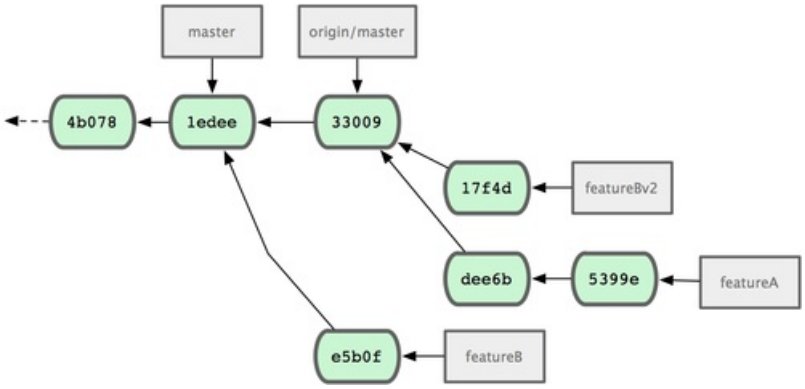


图 5-18. featureBv2 之后的提交历史

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 <jessica@example.com>
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，可以仔细读读，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。）：

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

然后，你应该去你到草稿箱去更改你要发送的补丁的收件人信息，以及需要抄送的人，然后发送它。

您也可以通过SMTP服务器发送补丁。和上面一样，你可以通过 `git config` 命令单独设置每个参数，也可以在你的 `~/.gitconfig` 文件中的sendemail节点手动添加它们。

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

配置完成后，您可以使用 `git send-email` 来发送你的补丁：

```
$ 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 <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
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 <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
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: <y>
References: <y>

Result: OK
```

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，最后再决定是不是真的要并入主干。

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 <jessica@example.com>
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 <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
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，而我们也需要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` 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 <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
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。

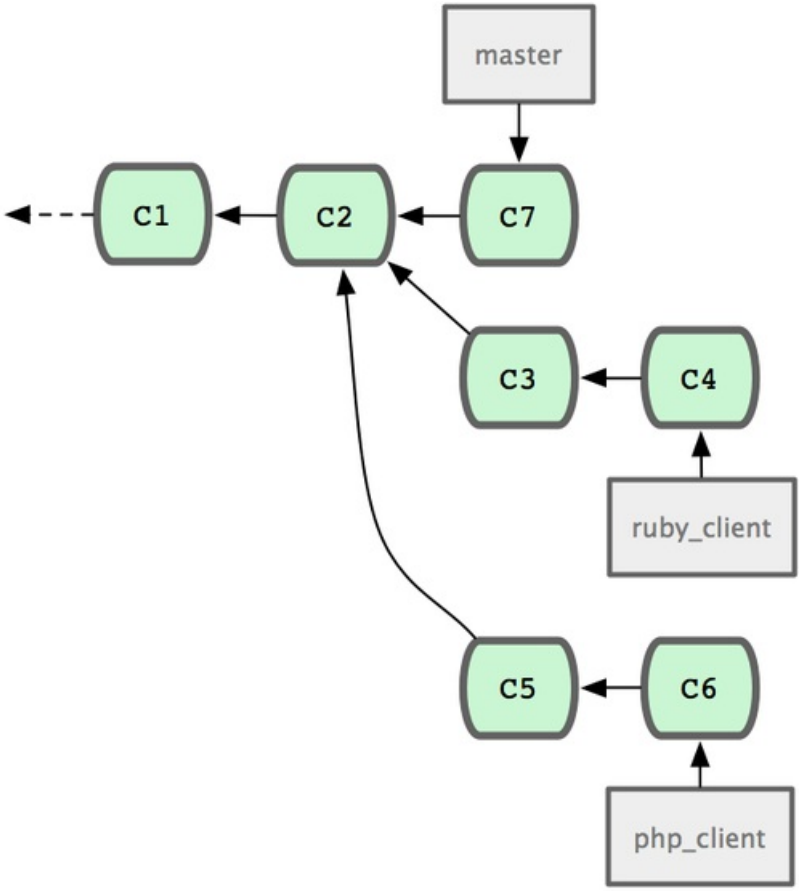


图 5-19. 多个特性分支

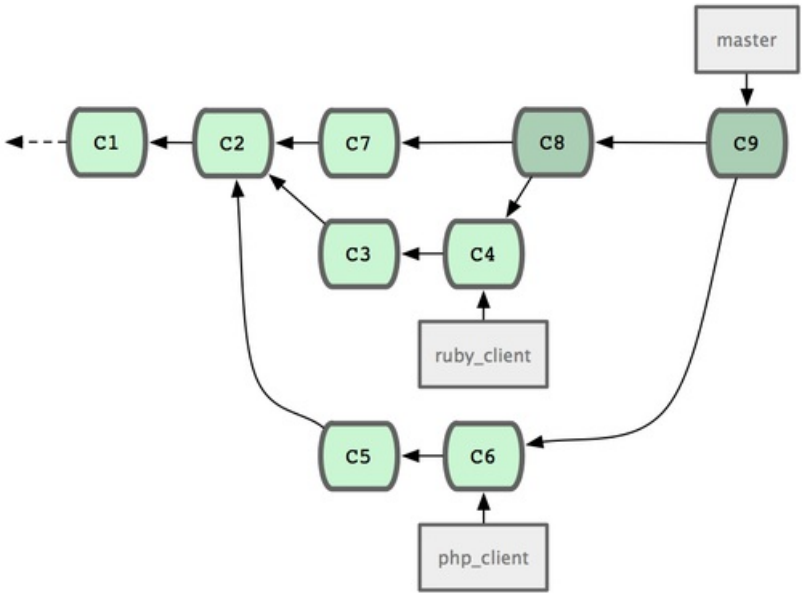


图 5-20. 合并特性分支之后

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。

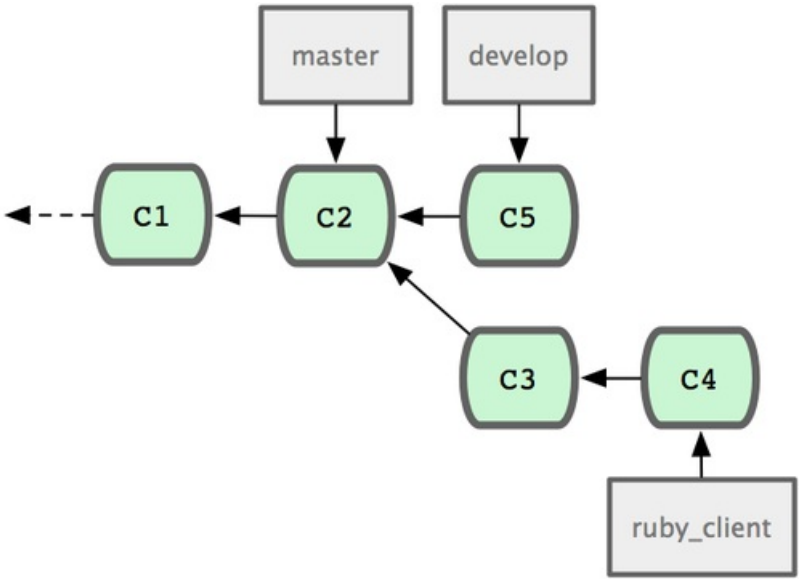


图 5-21. 特性分支合并前

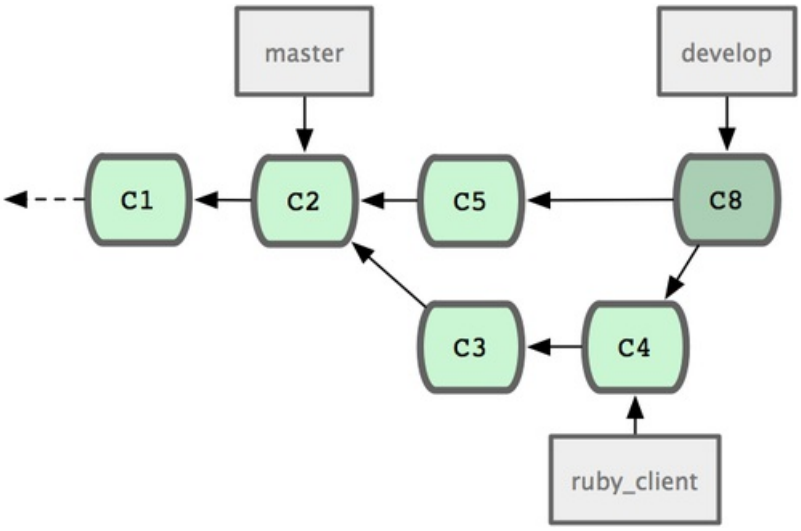


图 5-22. 特性分支合并后

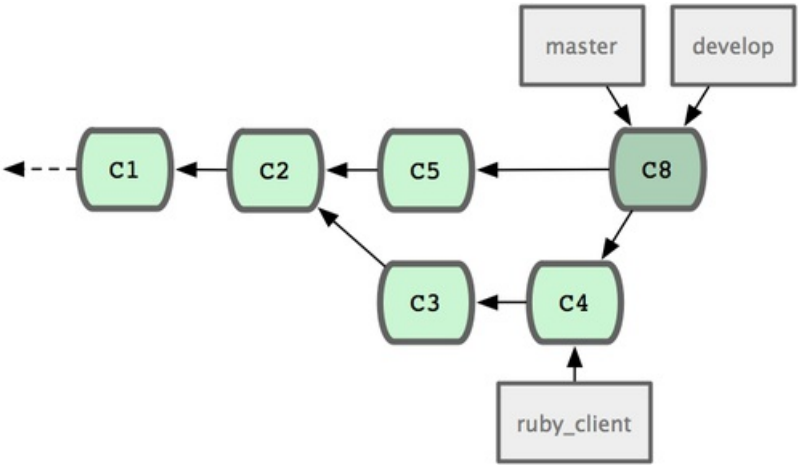


图 5-23. 特性分支发布后

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。

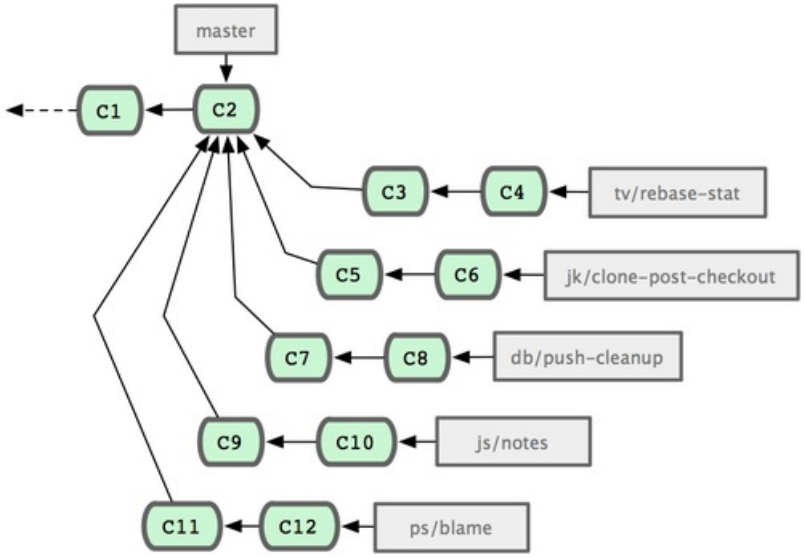


图 5-24. 管理复杂的并行贡献

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：

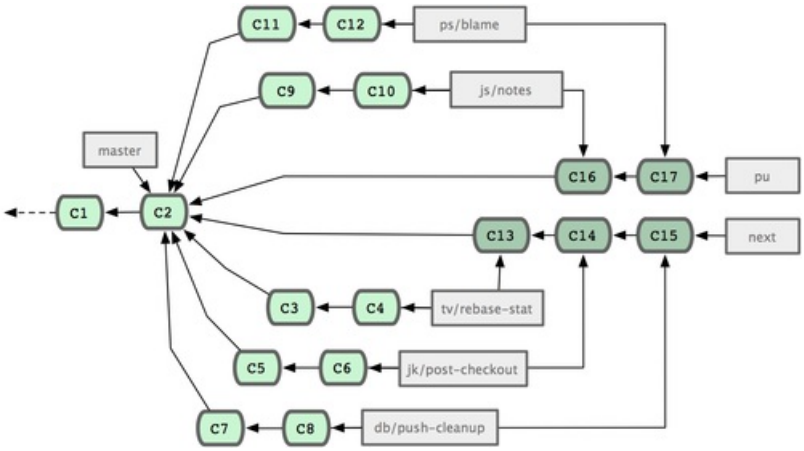


图 5-25. 将特性并入长期分支

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，好在你的主干分支上（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，但你只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。

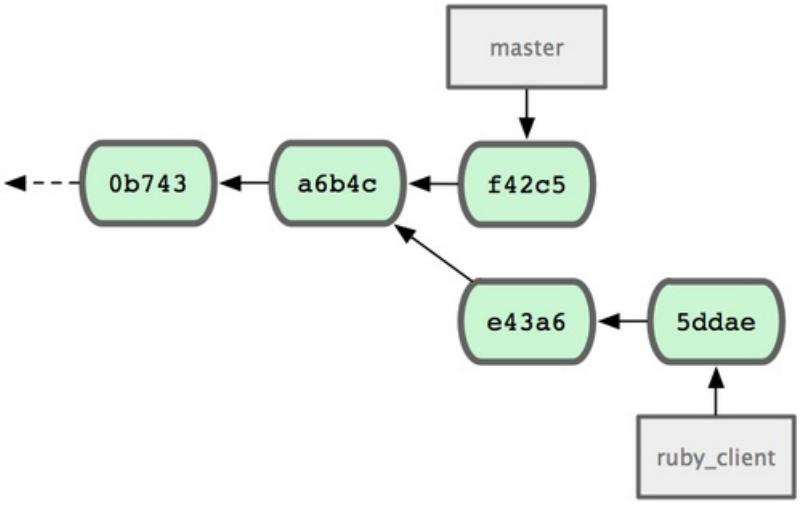


图 5-26. 挑拣（cherry-pick）之前的历史

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.

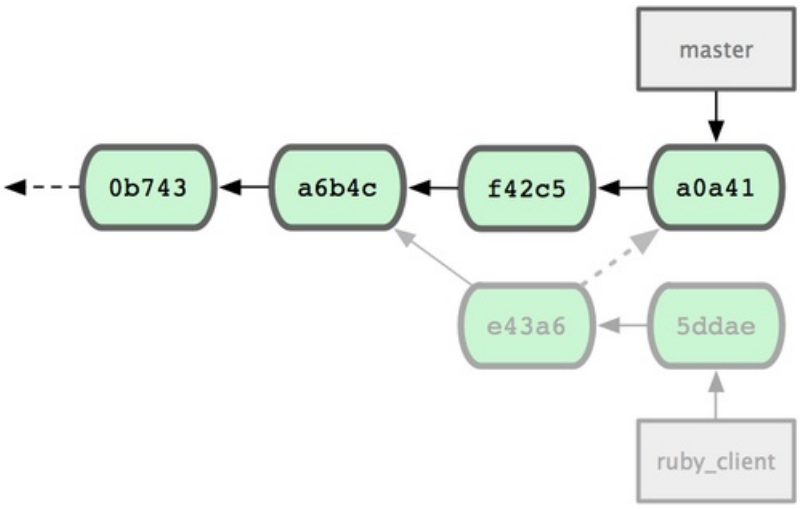


图 5-27. 挑拣（cherry-pick）之后的历史

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 <schacon@gmail.com>"
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（即公钥）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 <schacon@gmail.com>
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-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

在运行 `git push --tags` 命令之后， `maintainer-pgp-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-pgp-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 <tag>` 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，再加上一段所描述的提交的SHA-1值：

```
$ 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

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 x 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执行 `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某个分支在一定时间前的位置。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，引用日志信息只存在于本地——这是一个记录你在你自己的仓库里做过什么的日志。其他人拷贝的仓库里的引用日志不会和你的相同；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。`git show HEAD@{2.months.ago}` 这条命令只有在你克隆了一个项目至少两个月时才会有用——如果你是五分钟前克隆的仓库，那么它将不会有结果返回。

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。

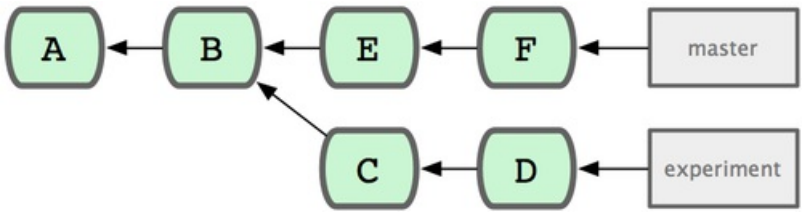


图 6-1. 范围选择的提交历史实例

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` 并且的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；但是你也许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。

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
#
# Changes not staged for commit:
#   (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
# Changes not staged for commit:
#   (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。如果你是这么做的，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
#
# Changes not staged for commit:
#   (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。

取消储藏(Un-applying a Stash)

在某些情况下，你可能想应用储藏的修改，在进行了一些其他的修改后，又要取消之前所应用储藏的修改。Git没有提供类似于

`stash unapply` 的命令，但是可以通过取消该储藏的补丁达到同样的效果：

```
$ git stash show -p stash@{0} | git apply -R
```

同样的，如果你没有指定具体的某个储藏，Git 会选择最近的储藏：

```
$ git stash show -p | git apply -R
```

你可能会想要新建一个别名，在你的 Git 里增加一个 `stash-unapply` 命令，这样更有效率。例如：

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
$ #... 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
#
# Changes not staged for commit:
#   (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。

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了你该干什么。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，借此对它的能力有所认识。

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 "find * -type f -name '*~' -delete" 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。