返回 码云 首页(目录) 章节列表 ▾ Pro Git（中文版）

1. 1.起步
    1. 1.1 关于版本控制
    2. 1.2 Git 简史
    3. 1.3 Git 基础
    4. 1.4 安装 Git
    5. 1.5 初次运行 Git 前的配置
    6. 1.6 获取帮助
    7. 1.7 小结
2. 2.Git 基础
    1. 2.1 取得项目的 Git 仓库
    2. 2.2 记录每次更新到仓库
    3. 2.3 查看提交历史
    4. 2.4 撤消操作
    5. 2.5 远程仓库的使用
    6. 2.6 打标签
    7. 2.7 技巧和窍门
    8. 2.8 小结
3. 3.Git 分支
    1. 3.1 何谓分支
    2. 3.2 分支的新建与合并
    3. 3.3 分支的管理
    4. 3.4 利用分支进行开发的工作流程
    5. 3.5 远程分支
    6. 3.6 分支的衍合
    7. 3.7 小结
1. 4.服务器上的 Git
    1. 4.1 协议
    2. 4.2 在服务器上部署 Git
    3. 4.3 生成 SSH 公钥
    4. 4.4 架设服务器
    5. 4.5 公共访问
    6. 4.6 GitWeb
    7. 4.7 Gitosis
    8. 4.8 Gitolite
    9. 4.9 Git 守护进程
    10. 4.10 Git 托管服务
    11. 4.11 小结

2. 5.分布式 Git

    1. 5.1 分布式工作流程
    2. 5.2 为项目作贡献
    3. 5.3 项目的管理
    4. 5.4 小结

3. 6.Git 工具

    1. 6.1 修订版本（Revision）选择
    2. 6.2 交互式暂存
    3. 6.3 储藏（Stashing）
    4. 6.4 重写历史
    5. 6.5 使用 Git 调试
    6. 6.6 子模块
    7. 6.7 子树合并
    8. 6.8 总结

1. 7.自定义 Git

    1. 7.1 配置 Git
    2. 7.2 Git属性
    3. 7.3 Git挂钩
    4. 7.4 Git 强制策略实例
    5. 7.5 总结

2. 8.Git 与其他系统

    1. 8.1 Git 与 Subversion
    2. 8.2 迁移到 Git
    3. 8.3 总结

3. 9.Git 内部原理

    1. 9.1 底层命令 (Plumbing) 和高层命令 (Porcelain)
    2. 9.2 Git 对象
    3. 9.3 Git References
    4. 9.4 Packfiles
    5. 9.5 The Refspec
    6. 9.6 传输协议
    7. 9.7 维护及数据恢复
    8. 9.8 总结

6 Git 工具

1. 6.1 修订版本（Revision）选择

2. 6.2 交互式暂存
3. 6.3 储藏（Stashing）
4. 6.4 重写历史
5. 6.5 使用 Git 调试
6. 6.6 子模块
7. 6.7 子树合并
8. 6.8 总结

现在，你已经学习了管理或者维护 Git 仓库，实现代码控制所需的大多数日常命令和工作流程。你已经完成了跟踪和提交文件的基本任务，并且发挥了暂存区和轻量级的特性分支及合并的威力。

接下来你将领略到一些 Git 可以实现的非常强大的功能，这些功能你可能并不会在日常操作中使用，但在某些时候你也许会需要。

6.1 修订版本（Revision）选择

Git 允许你通过几种方法来指明特定的或者一定范围内的提交。了解它们并不是必需的，但是了解一下总没坏处。

单个修订版本

显然你可以使用给出的 SHA-1 值来指明一次提交，不过也有更加人性化的方法来做同样的事。本节概述了指明单个提交的诸多方法。

简短的SHA

Git 很聪明，它能够通过你提供的前几个字符来识别你想要的那次提交，只要你提供的那部分 SHA-1 不短于四个字符，并且没有歧义——也就是说，当前仓库中只有一个对象以这段 SHA-1 开头。

例如，想要查看一次指定的提交，假设你运行 `git log` 命令并找到你增加了功能的那次提交：

```
$ git log
    commit 734713bc047d87bf7eac9674765ae793478c50d3
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

    commit d921970aadf03b3cf0e71becdaab3147ba71cdef
    Merge: 1c002dd... 35cfb2b...
    Author: Scott Chacon <schacon@gmail.com>
    Date: Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

    commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
    Author: Scott Chacon <schacon@gmail.com>
    Date: Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

假设是 `1c002dd....` 。如果你想 `git show` 这次提交，下面的命令是等价的（假设简短的版本没有歧义）：

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
    $ git show 1c002dd4b536e7479f
```

```
$ git show 1c002d
```

Git 可以为你的 SHA-1 值生成出简短且唯一的缩写。如果你传递 `--abbrev-commit` 给 `git log` 命令，输出结果里就会使用简短且唯一的值；它默认使用七个字符来表示，不过必要时为了避免 SHA-1 的歧义，会增加字符数：

```
$ git log --abbrev-commit --pretty=oneline
    ca82a6d changed the version number
    085bb3b removed unnecessary test code
    a11bef0 first commit
```

通常在一个项目中，使用八到十个字符来避免 SHA-1 歧义已经足够了。最大的 Git 项目之一，Linux 内核，目前也只需要最长 40 个字符中的 12 个字符来保持唯一性。

关于 SHA-1 的简短说明

许多人可能会担心一个问题：在随机的偶然情况下，在他们的仓库里会出现两个具有相同 SHA-1 值的对象。那会怎么样呢？

如果你真的向仓库里提交了一个跟之前的某个对象具有相同 SHA-1 值的对象，Git 将会发现之前的那个对象已经存在在 Git 数据库中，并认为它已经被写入了。如果什么时候你想再次检出那个对象时，你会总是得到先前的那个对象的数据。

不过，你应该了解到，这种情况发生的概率是多么微小。SHA-1 摘要长度是 20 字节，也就是 160 位。为了保证有 50% 的概率出现一次冲突，需要 2^80 个随机哈希的对象（计算冲突机率的公式是 `p = (n(n-1)/2) * (1/2^160))`。2^80 是 1.2 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。

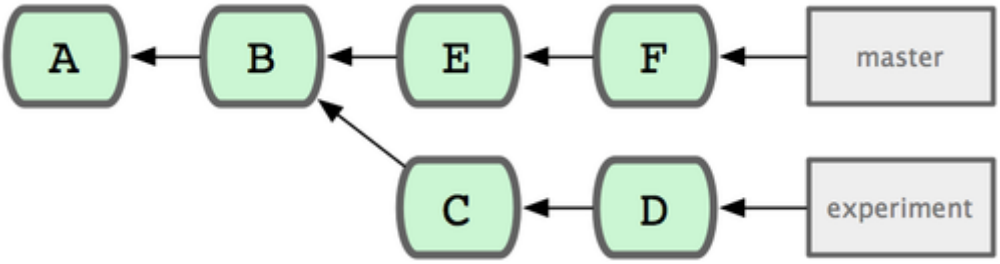

图 6-1. 范围选择的提交历史实例

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。

6.2 交互式暂存

Git提供了很多脚本来辅助某些命令行任务。这里，你将看到一些交互式命令，它们帮助你方便地构建只包含特定组合和部分文件的提交。在你修改了一大批文件然后决定将这些变更分布在几个各有侧重的提交而不是单个又大又乱的提交时，这些工具非常有用。用这种方法，你可以确保你的提交在逻辑上划分为相应的变更集，以便于供和你一起工作的开发者审阅。如果你运行git add时加上-i或者--interactive选项，Git就进入了一个交互式的shell模式，显示一些类似于下面的信息：

```
$ git add -i
    staged unstaged path
    1: unchanged +0/-1 TODO
    2: unchanged +1/-1 index.html
    3: unchanged +5/-1 lib/simplegit.rb

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

你会看到这个命令以一个完全不同的视图显示了你的暂存区——主要是你通过`git status`得到的那些信息但是稍微简洁但信息更加丰富一些。它在左侧列出了你暂存的变更，在右侧列出了未被暂存的变更。

在这之后是一个命令区。这里你可以做很多事情，包括暂存文件，撤回文件，暂存部分文件，加入未被追踪的文件，查看暂存文件的差别。

暂存和撤回文件

如果你在`What now>`的提示后输入`2`或者`u`，这个脚本会提示你那些文件你想要暂存：

```
What now> 2
    staged unstaged path
    1: unchanged +0/-1 TODO
    2: unchanged +1/-1 index.html
    3: unchanged +5/-1 lib/simplegit.rb
    Update>>
```

如果想暂存TODO和index.html，你可以输入相应的编号：

```
Update>> 1,2
    staged unstaged path
    * 1: unchanged +0/-1 TODO
    * 2: unchanged +1/-1 index.html
    3: unchanged +5/-1 lib/simplegit.rb
    Update>>
```

每个文件旁边的`*`表示选中的文件将被暂存。如果你在`update>>`提示后直接敲入回车，Git会替你把所有选中的内容暂存：

```
Update>>
    updated 2 paths

    *** Commands ***
    1: status 2: update 3: revert 4: add untracked
    5: patch 6: diff 7: quit 8: help
    What now> 1
    staged unstaged path
    1: +0/-1 nothing TODO
    2: +1/-1 nothing index.html
    3: unchanged +5/-1 lib/simplegit.rb
```

现在你可以看到TODO和index.html文件被暂存了同时simplegit.rb文件仍然未被暂存。如果这时你想要撤回TODO文件，就使用`3`或者`r`（代表revert，恢复）选项：

```
*** Commands ***
    1: status 2: update 3: revert 4: add untracked
```

```
    5: patch 6: diff 7: quit 8: help
    What now> 3
    staged unstaged path
    1: +0/-1 nothing TODO
    2: +1/-1 nothing index.html
    3: unchanged +5/-1 lib/simplegit.rb
    Revert>> 1
    staged unstaged path
    * 1: +0/-1 nothing TODO
    2: +1/-1 nothing index.html
    3: unchanged +5/-1 lib/simplegit.rb
    Revert>> [enter]
    reverted one path
```

再次查看Git的状态，你会看到你已经撤回了TODO文件

```
*** Commands ***
    1: status 2: update 3: revert 4: add untracked
    5: patch 6: diff 7: quit 8: help
    What now> 1
    staged unstaged path
    1: unchanged +0/-1 TODO
    2: +1/-1 nothing index.html
    3: unchanged +5/-1 lib/simplegit.rb
```

要查看你暂存内容的差异，你可以使用6或者d（表示diff）命令。它会显示你暂存文件的列表，你可以选择其中的几个，显示其被暂存的差异。这跟你在命令行下指定`git diff --cached`非常相似：

```
*** Commands ***
    1: status 2: update 3: revert 4: add untracked
    5: patch 6: diff 7: quit 8: help
    What now> 6
    staged unstaged path
    1: +1/-1 nothing index.html
    Review diff>> 1
    diff --git a/index.html b/index.html
    index 4d07108..4335f49 100644
    --- a/index.html
    +++ b/index.html
    @@ -16,7 +16,7 @@ Date Finder

    <p id="out">...</p>

    -<div id="footer">contact : support@github.com</div>
    +<div id="footer">contact : email.support@github.com</div>

    <script type="text/javascript">
```

通过这些基本命令，你可以使用交互式增加模式更加方便地处理暂存区。

暂存补丁

只让Git暂存文件的某些部分而忽略其他也是有可能的。例如，你对simplegit.rb文件作了两处修改但是只想暂存其中一个而忽略另一个，在Git中实现这一点非常容易。在交互式的提示符下，输入5或者p（表示patch，补丁）。Git会询问哪些文件你希望部分暂存；然后对于被选中文件的每一节，他会逐个显示文件的差异区块并询问你是否希望暂存他们：

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
    index dd5ecc4..57399e0 100644
    --- a/lib/simplegit.rb
    +++ b/lib/simplegit.rb
    @@ -22,7 +22,7 @@ class SimpleGit
    end
```

```
    def log(treeish = 'master')
    - command("git log -n 25 #{treeish}")
    + command("git log -n 30 #{treeish}")
    end

    def blame(path)
    Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

此处你有很多选择。输入?可以显示列表：

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
    y - stage this hunk
    n - do not stage this hunk
    a - stage this and all the remaining hunks in the file
    d - do not stage this hunk nor any of the remaining hunks in the file
    g - select a hunk to go to
    / - search for a hunk matching the given regex
    j - leave this hunk undecided, see next undecided hunk
    J - leave this hunk undecided, see next hunk
    k - leave this hunk undecided, see previous undecided hunk
    K - leave this hunk undecided, see previous hunk
    s - split the current hunk into smaller hunks
    e - manually edit the current hunk
    ? - print help
```

如果你想暂存各个区块，通常你会输入y或者n，但是暂存特定文件里的全部区块或者暂时跳过对一个区块的处理同样也很有用。如果你暂存了文件的一个部分而保留另外一个部分不被暂存，你的状态输出看起来会是这样：

```
What now> 1
    staged unstaged path
    1: unchanged +0/-1 TODO
    2: +1/-1 nothing index.html
    3: +1/-1 +4/-0 lib/simplegit.rb
```

simplegit.rb的状态非常有意思。它显示有几行被暂存了，有几行没有。你部分地暂存了这个文件。在这时，你可以退出交互式脚本然后运行`git commit`来提交部分暂存的文件。

最后你也可以不通过交互式增加的模式来实现部分文件暂存——你可以在命令行下通过`git add -p`或者`git add --patch`来启动同样的脚本。

6.3 储藏（Stashing）

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是`git stash`命令。

“‘储藏”“可以获取你工作目录的中间状态——也就是你修改过的被追踪的文件和暂存的变更——并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

储藏你的工作

为了演示这一功能，你可以进入你的项目，在一些文件上进行工作，有可能还暂存其中一个变更。如果你运行 `git status`，你可以看到你的中间状态：

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

6.4 重写历史

很多时候，在 Git 上工作的时候，你也许会由于某种原因想要修订你的提交历史。Git 的一个卓越之处就是它允许你在最后可能的时刻再作决定。你可以在你即将提交暂存区时决定什么文件归入哪一次提交，你可以使用 stash 命令来决定你暂时搁置的工作，你可以重写已经发生的提交以使它们看起来是另外一种样子。这个包括改变提交的次序、改变说明或者修改提交中包含的文件，将提交归并、拆分或者完全删除——这一切在你尚未开始将你的工作和别人共享前都是可以的。

在这一节中，你会学到如何完成这些很有用的任务以使你的提交历史在你将其共享给别人之前变成你想要的样子。

改变最近一次提交

改变最近一次提交也许是最常见的重写历史的行为。对于你的最近一次提交，你经常想做两件基本事情：改变提交说明，或者改变你刚刚通过增加，改变，删除而记录的快照。

如果你只想修改最近一次提交说明，这非常简单：

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

这会把你带入文本编辑器，里面包含了你最近一次提交说明，供你修改。当你保存并退出编辑器，这个编辑器会写入一个新的提交，里面包含了那个说明，并且让它成为你的新的最近一次提交。

如果你完成提交后又想修改被提交的快照，增加或者修改其中的文件，可能因为你最初提交时，忘了添加一个新建的文件，这个过程基本上一样。你通过修改文件然后对其运行`git add`或对一个已被记录的文件运行`git rm`，随后的`git commit --amend`会获取你当前的暂存区并将它作为新提交对应的快照。

使用这项技术的时候你必须小心，因为修正会改变提交的SHA-1值。这个很像是一次非常小的rebase——不要在你最近一次提交被推送后还去修正它。

修改多个提交说明

要修改历史中更早的提交，你必须采用更复杂的工具。Git没有一个修改历史的工具，但是你可以使用rebase工具来衍合一系列的提交到它们原来所在的HEAD上而不是移到新的上。依靠这个交互式的rebase工具，你就可以停留在每一次提交后，如果你想修改或改变说明、增加文件或任何其他事情。你可以通过给`git rebase`增加`-i`选项来以交互方式地运行rebase。你必须通过告诉命令衍合到哪次提交，来指明你需要重写的提交的回溯深度。

例如，你想修改最近三次的提交说明，或者其中任意一次，你必须给`git rebase -i`提供一个参数，指明你想要修改的提交的父提交，例如`HEAD~2`或者`HEAD~3`。可能记住`~3`更加容易，因为你想修改最近三次提交；但是请记住你事实上所指的是四次提交之前，即你想修改的提交的父提交。

```
$ git rebase -i HEAD~3
```

再次提醒这是一个衍合命令——`HEAD~3..HEAD`范围内的每一次提交都会被重写，无论你是否修改说明。不要涵盖你已经推送到中心服务器的提交——这么做会使其他开发者产生混乱，因为你提供了同样变更的不同版本。

运行这个命令会为你的文本编辑器提供一个提交列表，看起来像下面这样

```
pick f7f3f6d changed my name a bit
    pick 310154e updated README formatting and added blame
    pick a5f4a0d added cat-file

    # Rebase 710f0f8..a5f4a0d onto 710f0f8
    #
    # Commands:
    # p, pick = use commit
    # e, edit = use commit, but stop for amending
    # s, squash = use commit, but meld into previous commit
    #
    # If you remove a line here THAT COMMIT WILL BE LOST.
    # However, if you remove everything, the rebase will be aborted.
    #
```

很重要的一点是你得注意这些提交的顺序与你通常通过`log`命令看到的是相反的。如果你运行`log`，你会看到下面这样的结果：

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
    a5f4a0d added cat-file
    310154e updated README formatting and added blame
    f7f3f6d changed my name a bit
```

请注意这里的倒序。交互式的rebase给了你一个即将运行的脚本。它会从你在命令行上指明的提交开始(`HEAD~3`)然后自上至下重播每次提交里引入的变更。它将最早的列在顶上而不是最近的，因为这是第一个需要重播的。

你需要修改这个脚本来让它停留在你想修改的变更上。要做到这一点，你只要将你想修改的每一次提交前面的pick改为edit。例如，只想修改第三次提交说明的话，你就像下面这样修改文件：

```
edit f7f3f6d changed my name a bit
    pick 310154e updated README formatting and added blame
    pick a5f4a0d added cat-file
```

当你保存并退出编辑器，Git会倒回至列表中的最后一次提交，然后把你送到命令行中，同时显示以下信息：

```
$ git rebase -i HEAD~3
    Stopped at 7482e0d... updated the gemspec to hopefully work better
    You can amend the commit now, with

    git commit --amend

    Once you're satisfied with your changes, run

    git rebase --continue
```

这些指示很明确地告诉了你该干什么。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 'rm -f *~' HEAD`的命令。

你可以观察到 Git 重写目录树并且提交，然后将分支指针移到末尾。一个比较好的办法是在一个测试分支上做这些然后在你确定产物真的是你所要的之后，再 hard-reset 你的主分支。要在你所有的分支上运行`filter-branch`的话，你可以传递一个`--all`给命令。

将一个子目录设置为新的根目录

假设你完成了从另外一个代码控制系统的导入工作，得到了一些没有意义的子目录（trunk, tags等等）。如果你想让`trunk`子目录成为每一次提交的新的项目根目录，`filter-branch`也可以帮你做到：

```
$ git filter-branch --subdirectory-filter trunk HEAD
    Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
    Ref 'refs/heads/master' was rewritten
```

现在你的项目根目录就是`trunk`子目录了。Git 会自动地删除不对这个子目录产生影响的提交。

全局性地更换电子邮件地址

另一个常见的案例是你在开始时忘了运行`git config`来设置你的姓名和电子邮件地址，也许你想开源一个项目，把你所有的工作电子邮件地址修改为个人地址。无论哪种情况你都可以用`filter-branch`来更换多次提交里的电子邮件地址。你必须小心一些，只改变属于你的电子邮件地址，所以你使用`--commit-filter`：

```
$ git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
    GIT_AUTHOR_NAME="Scott Chacon";
```

```
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
    else
    git commit-tree "$@";
    fi' HEAD
```

这个会遍历并重写所有提交使之拥有你的新地址。因为提交里包含了它们的父提交的SHA-1值，这个命令会修改你的历史中的所有提交，而不仅仅是包含了匹配的电子邮件地址的那些。

6.5 使用 Git 调试

Git 同样提供了一些工具来帮助你调试项目中遇到的问题。由于 Git 被设计为可应用于几乎任何类型的项目，这些工具是通用型，但是在遇到问题时可以经常帮助你查找缺陷所在。

文件标注

如果你在追踪代码中的缺陷想知道这是什么时候为什么被引进来的，文件标注会是你的最佳工具。它会显示文件中对每一行进行修改的最近一次提交。因此，如果你发现自己代码中的一个方法存在缺陷，你可以用`git blame`来标注文件，查看那个方法的每一行分别是由谁在哪一天修改的。下面这个例子使用了`-L`选项来限制输出范围在第12至22行：

```
$ git blame -L 12,22 simplegit.rb
    ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
    ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13) command("git show #{tree}")
    ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
    ^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
    9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
    79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17) command("git log #{tree}")
    9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
    9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
    42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
    42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{path}")
    42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

请注意第一个域里是最后一次修改该行的那次提交的 SHA-1 值。接下去的两个域是从那次提交中抽取的值——作者姓名和日期——所以你可以方便地获知谁在什么时候修改了这一行。在这后面是行号和文件的内容。请注意`^4832fe2`提交的那些行，这些指的是文件最初提交的那些行。那个提交是文件第一次被加入这个项目时存在的，自那以后未被修改过。这会带来小小的困惑，因为你已经至少看到了Git使用`^`来修饰一个提交的SHA值的三种不同的意义，但这里确实就是这个意思。

另一件很酷的事情是在 Git 中你不需要显式地记录文件的重命名。它会记录快照然后根据现实尝试找出隐式的重命名动作。这其中有一个很有意思的特性就是你可以让它找出所有的代码移动。如果你在`git blame`后加上`-C`，Git会分析你在标注的文件然后尝试找出其中代码片段的原始出处，如果它是从其他地方拷贝过来的话。最近，我在将一个名叫`GITServerHandler.m`的文件分解到多个文件中，其中一个是`GITPackUpload.m`。通过对`GITPackUpload.m`执行带`-C`参数的blame命令，我可以看到代码块的原始出处：

```
$ git blame -C -L 141,153 GITPackUpload.m
    f344f58d GITServerHandler.m (Scott 2009-01-04 141)
    f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
    f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
    70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMMI
    ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
    ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;
    ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
    ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
    ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMMI
```

```
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

这真的非常有用。通常，你会把你拷贝代码的那次提交作为原始提交，因为这是你在这个文件中第一次接触到那几行。Git可以告诉你编写那些行的原始提交，即便是在另一个文件里。

二分查找

标注文件在你知道问题是哪里引入的时候会有帮助。如果你不知道，并且自上次代码可用的状态已经经历了上百次的提交，你可能就要求助于`bisect`命令了。`bisect`会在你的提交历史中进行二分查找来尽快地确定哪一次提交引入了错误。

例如你刚刚推送了一个代码发布版本到产品环境中，对代码为什么会表现成那样百思不得其解。你回到你的代码中，还好你可以重现那个问题，但是找不到在哪里。你可以对代码执行bisect来寻找。首先你运行`git bisect start`启动，然后你用`git bisect bad`来告诉系统当前的提交已经有问题了。然后你必须告诉bisect已知的最后一次正常状态是哪次提交，使用`git bisect good [good_commit]`：

```
$ git bisect start
    $ git bisect bad
    $ git bisect good v1.0
    Bisecting: 6 revisions left to test after this
    [ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git 发现在你标记为正常的提交(v1.0)和当前的错误版本之间有大约12次提交，于是它检出中间的一个。在这里，你可以运行测试来检查问题是否存在于这次提交。如果是，那么它是在这个中间提交之前的某一次引入的；如果否，那么问题是在中间提交之后引入的。假设这里是没有错误的，那么你就通过`git bisect good`来告诉 Git 然后继续你的旅程：

```
$ git bisect good
    Bisecting: 3 revisions left to test after this
    [b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

现在你在另外一个提交上了，在你刚刚测试通过的和一个错误提交的中点处。你再次运行测试然后发现这次提交是错误的，因此你通过`git bisect bad`来告诉Git：

```
$ git bisect bad
    Bisecting: 1 revisions left to test after this
    [f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

这次提交是好的，那么 Git 就获得了确定问题引入位置所需的所有信息。它告诉你第一个错误提交的 SHA-1 值并且显示一些提交说明以及哪些文件在那次提交里修改过，这样你可以找出缺陷被引入的根源：

```
$ git bisect good
    b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
    commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
    Author: PJ Hyett <pjhyett@example.com>
    Date: Tue Jan 27 14:48:32 2009 -0800

    secure this thing

    :040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
    f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config
```

当你完成之后，你应该运行`git bisect reset`来重设你的HEAD到你开始前的地方，否则你会处于一个诡异的地方：

```
$ git bisect reset
```

这是个强大的工具，可以帮助你检查上百的提交，在几分钟内找出缺陷引入的位置。事实上，如果你有一个脚本会在工程正常时返回0，错误时返回非0的话，你可以完全自动地执行`git bisect`。首先你需要提供已知的错误和正确提交来告诉它二分查找的范围。你可以通过`bisect start`命令来列出它们，先列出已知的错误提交再列出已知的正确提交：

```
$ git bisect start HEAD v1.0
    $ git bisect run test-error.sh
```

这样会自动地在每一个检出的提交里运行`test-error.sh`直到Git找出第一个破损的提交。你也可以运行像`make`或者`make tests`或者任何你所拥有的来为你执行自动化的测试。

6.6 子模块

经常有这样的事情，当你在一个项目上工作时，你需要在其中使用另外一个项目。也许它是一个第三方开发的库或者是你独立开发和并在多个父项目中使用的。这个场景下一个常见的问题产生了：你想将两个项目单独处理但是又需要在其中一个中使用另外一个。

这里有一个例子。假设你在开发一个网站，为之创建Atom源。你不想编写一个自己的Atom生成代码，而是决定使用一个库。你可能不得不像CPAN install或者Ruby gem一样包含来自共享库的代码，或者将代码拷贝到你的项目树中。如果采用包含库的办法，那么不管用什么办法都很难去定制这个库，部署它就更加困难了，因为你必须确保每个客户都拥有那个库。把代码包含到你自己的项目中带来的问题是，当上游被修改时，任何你进行的定制化的修改都很难归并。

Git 通过子模块处理这个问题。子模块允许你将一个 Git 仓库当作另外一个Git仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。

子模块初步

假设你想把 Rack 库（一个 Ruby 的 web 服务器网关接口）加入到你的项目中，可能既要保持你自己的变更，又要延续上游的变更。首先你要把外部的仓库克隆到你的子目录中。你通过`git submodule add`将外部项目加为子模块：

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
    Initialized empty Git repository in /opt/subtest/rack/.git/
    remote: Counting objects: 3181, done.
    remote: Compressing objects: 100% (1534/1534), done.
    remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
    Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
    Resolving deltas: 100% (1951/1951), done.
```

现在你就在项目里的`rack`子目录下有了一个 Rack 项目。你可以进入那个子目录，进行变更，加入你自己的远程可写仓库来推送你的变更，从原始仓库拉取和归并等等。如果你在加入子模块后立刻运行`git status`，你会看到下面两项：

```
$ git status
    # On branch master
```

```
    # Changes to be committed:
    # (use "git reset HEAD <file>..." to unstage)
    #
    # new file: .gitmodules
    # new file: rack
    #
```

首先你注意到有一个`.gitmodules`文件。这是一个配置文件，保存了项目 URL 和你拉取到的本地子目录

```
$ cat .gitmodules
    [submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

如果你有多个子模块，这个文件里会有多个条目。很重要的一点是这个文件跟其他文件一样也是处于版本控制之下的，就像你的`.gitignore`文件一样。它跟项目里的其他文件一样可以被推送和拉取。这是其他克隆此项目的人获知子模块项目来源的途径。

`git status`的输出里所列的另一项目是 rack 。如果你运行在那上面运行`git diff`，会发现一些有趣的东西：

```
$ git diff --cached rack
    diff --git a/rack b/rack
    new file mode 160000
    index 0000000..08d709f
    --- /dev/null
    +++ b/rack
    @@ -0,0 +1 @@
    +Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

尽管`rack`是你工作目录里的子目录，但 Git 把它视作一个子模块，当你不在那个目录里时并不记录它的内容。取而代之的是，Git 将它记录成来自那个仓库的一个特殊的提交。当你在那个子目录里修改并提交时，子项目会通知那里的 HEAD 已经发生变更并记录你当前正在工作的那个提交；通过那样的方法，当其他人克隆此项目，他们可以重新创建一致的环境。

这是关于子模块的重要一点：你记录他们当前确切所处的提交。你不能记录一个子模块的`master`或者其他的符号引用。

当你提交时，会看到类似下面的：

```
$ git commit -m 'first commit with submodule rack'
    [master 0550271] first commit with submodule rack
    2 files changed, 4 insertions(+), 0 deletions(-)
    create mode 100644 .gitmodules
    create mode 160000 rack
```

注意 rack 条目的 160000 模式。这在Git中是一个特殊模式，基本意思是你将一个提交记录为一个目录项而不是子目录或者文件。

你可以将`rack`目录当作一个独立的项目，保持一个指向子目录的最新提交的指针然后反复地更新上层项目。所有的Git命令都在两个子目录里独立工作：

```
$ git log -1
    commit 0550271328a0038865aad6331e620cd7238601bb
    Author: Scott Chacon <schacon@gmail.com>
    Date: Thu Apr 9 09:03:56 2009 -0700
```

```
    first commit with submodule rack
    $ cd rack/
    $ git log -1
    commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
    Author: Christian Neukirchen <chneukirchen@gmail.com>
    Date: Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

克隆一个带子模块的项目

这里你将克隆一个带子模块的项目。当你接收到这样一个项目，你将得到了包含子项目的目录，但里面没有文件：

```
$ git clone git://github.com/schacon/myproject.git
    Initialized empty Git repository in /opt/myproject/.git/
    remote: Counting objects: 6, done.
    remote: Compressing objects: 100% (4/4), done.
    remote: Total 6 (delta 0), reused 0 (delta 0)
    Receiving objects: 100% (6/6), done.
    $ cd myproject
    $ ls -l
    total 8
    -rw-r--r-- 1 schacon admin 3 Apr 9 09:11 README
    drwxr-xr-x 2 schacon admin 68 Apr 9 09:11 rack
    $ ls rack/
    $
```

`rack`目录存在了，但是是空的。你必须运行两个命令：`git submodule init`来初始化你的本地配置文件，`git submodule update`来从那个项目拉取所有数据并检出你上层项目里所列的合适的提交：

```
$ git submodule init
    Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
    $ git submodule update
    Initialized empty Git repository in /opt/myproject/rack/.git/
    remote: Counting objects: 3181, done.
    remote: Compressing objects: 100% (1534/1534), done.
    remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
    Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
    Resolving deltas: 100% (1951/1951), done.
    Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

现在你的`rack`子目录就处于你先前提交的确切状态了。如果另外一个开发者变更了 rack 的代码并提交，你拉取那个引用然后归并之，将得到稍有点怪异的东西：

```
$ git merge origin/master
    Updating 0550271..85a3eee
    Fast forward
    rack | 2 +-
    1 files changed, 1 insertions(+), 1 deletions(-)
    [master*]$ git status
    # On branch master
    # 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: rack
    #
```

你归并来的仅仅上是一个指向你的子模块的指针；但是它并不更新你子模块目录里的代码，所以看起来你的工作目录处于一个临时状态：

```
$ git diff
    diff --git a/rack b/rack
    index 6c5e70b..08d709f 160000
    --- a/rack
    +++ b/rack
    @@ -1 +1 @@
    -Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
    +Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

事情就是这样，因为你所拥有的指向子模块的指针和子模块目录的真实状态并不匹配。为了修复这一点，你必须再次运行`git submodule update`：

```
$ git submodule update
    remote: Counting objects: 5, done.
    remote: Compressing objects: 100% (3/3), done.
    remote: Total 3 (delta 1), reused 2 (delta 0)
    Unpacking objects: 100% (3/3), done.
    From git@github.com:schacon/rack
    08d709f..6c5e70b master -> origin/master
    Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

每次你从主项目中拉取一个子模块的变更都必须这样做。看起来很怪但是管用。

一个常见问题是当开发者对子模块做了一个本地的变更但是并没有推送到公共服务器。然后他们提交了一个指向那个非公开状态的指针然后推送上层项目。当其他开发者试图运行`git submodule update`，那个子模块系统会找不到所引用的提交，因为它只存在于第一个开发者的系统中。如果发生那种情况，你会看到类似这样的错误：

```
$ git submodule update
    fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
    Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

你不得不去查看谁最后变更了子模块

```
$ git log -1 rack
    commit 85a3eee996800fcfa91e2119372dd4172bf76678
    Author: Scott Chacon <schacon@gmail.com>
    Date: Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

然后，你给那个家伙发电子邮件说他一通。

上层项目

有时候，开发者想按照他们的分组获取一个大项目的子目录的子集。如果你是从 CVS 或者 Subversion 迁移过来的话这个很常见，在那些系统中你已经定义了一个模块或者子目录的集合，而你想延续这种类型的工作流程。

在 Git 中实现这个的一个好办法是你将每一个子目录都做成独立的 Git 仓库，然后创建一个上层项目的 Git 仓库包含多个子模块。这个办法的一个优势是你可以在上层项目中通过标签和分支更为明确地定义项目之间的关系。

子模块的问题

使用子模块并非没有任何缺点。首先，你在子模块目录中工作时必须相对小心。当你运行`git submodule update`，它会检出项目的指定版本，但是不在分支内。这叫做获得一个分离的头——这意味着 HEAD 文件直接指向一次提交，而不是一个符号引用。问题在于你通常并不想在一个分离的头的环境下工作，因为太容易丢失变更了。如果你先执行了一次`submodule update`，然后在那个子模块目录里不创建分支就进行提交，然后再次从上层项目里运行`git submodule update`同时不进行提交，Git会毫无提示地覆盖你的变更。技术上讲你不会丢失工作，但是你将失去指向它的分支，因此会很难取到。

为了避免这个问题，当你在子模块目录里工作时应使用`git checkout -b work`创建一个分支。当你再次在子模块里更新的时候，它仍然会覆盖你的工作，但是至少你拥有一个可以回溯的指针。

切换带有子模块的分支同样也很有技巧。如果你创建一个新的分支，增加了一个子模块，然后切换回不带该子模块的分支，你仍然会拥有一个未被追踪的子模块的目录

```
$ git checkout -b rack
    Switched to a new branch "rack"
    $ git submodule add git@github.com:schacon/rack.git rack
    Initialized empty Git repository in /opt/myproj/rack/.git/
    ...
    Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
    Resolving deltas: 100% (1952/1952), done.
    $ git commit -am 'added rack submodule'
    [rack cc49a69] added rack submodule
    2 files changed, 4 insertions(+), 0 deletions(-)
    create mode 100644 .gitmodules
    create mode 160000 rack
    $ git checkout master
    Switched to branch "master"
    $ git status
    # On branch master
    # Untracked files:
    # (use "git add <file>..." to include in what will be committed)
    #
    # rack/
```

你将不得不将它移走或者删除，这样的话当你切换回去的时候必须重新克隆它——你可能会丢失你未推送的本地的变更或分支。

最后一个需要引起注意的是关于从子目录切换到子模块的。如果你已经跟踪了你项目中的一些文件但是想把它们移到子模块去，你必须非常小心，否则Git会生你的气。假设你的项目中有一个子目录里放了 rack 的文件，然后你想将它转换为子模块。如果你删除子目录然后运行`submodule add`，Git会向你大吼：

```
$ rm -Rf rack/
    $ git submodule add git@github.com:schacon/rack.git rack
    'rack' already exists in the index
```

你必须先将`rack`目录撤回。然后你才能加入子模块：

```
$ git rm -r rack
    $ git submodule add git@github.com:schacon/rack.git rack
    Initialized empty Git repository in /opt/testsub/rack/.git/
    remote: Counting objects: 3184, done.
    remote: Compressing objects: 100% (1465/1465), done.
    remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
    Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
    Resolving deltas: 100% (1952/1952), done.
```

现在假设你在一个分支里那样做了。如果你尝试切换回一个仍然在目录里保留那些文件而不是子模块的分支时——你会得到下面的错误：

```
$ git checkout master
    error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

你必须先移除`rack`子模块的目录才能切换到不包含它的分支：

```
$ mv rack /tmp/
    $ git checkout master
    Switched to branch "master"
    $ ls
    README rack
```

然后，当你切换回来，你会得到一个空的`rack`目录。你可以运行`git submodule update`重新克隆，也可以将`/tmp/rack`目录重新移回空目录。

6.7 子树合并

现在你已经看到了子模块系统的麻烦之处，让我们来看一下解决相同问题的另一途径。当 Git 归并时，它会检查需要归并的内容然后选择一个合适的归并策略。如果你归并的分支是两个，Git使用一个*递归*策略。如果你归并的分支超过两个，Git采用*章鱼*策略。这些策略是自动选择的，因为递归策略可以处理复杂的三路归并情况——比如多于一个共同祖先的——但是它只能处理两个分支的归并。章鱼归并可以处理多个分支但是但必须更加小心以避免冲突带来的麻烦，因此它被选中作为归并两个以上分支的默认策略。

实际上，你也可以选择其他策略。其中的一个就是*子树*归并，你可以用它来处理子项目问题。这里你会看到如何换用子树归并的方法来实现前一节里所做的 rack 的嵌入。

子树归并的思想是你拥有两个工程，其中一个项目映射到另外一个项目的子目录中，反过来也一样。当你指定一个子树归并，Git可以聪明地探知其中一个是另外一个的子树从而实现正确的归并——这相当神奇。

首先你将 Rack 应用加入到项目中。你将 Rack 项目当作你项目中的一个远程引用，然后将它检出到它自身的分支：

```
$ git remote add rack_remote git@github.com:schacon/rack.git
    $ git fetch rack_remote
    warning: no common commits
    remote: Counting objects: 3184, done.
    remote: Compressing objects: 100% (1465/1465), done.
    remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
    Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
    Resolving deltas: 100% (1952/1952), done.
    From git@github.com:schacon/rack
    * [new branch] build -> rack_remote/build
    * [new branch] master -> rack_remote/master
    * [new branch] rack-0.4 -> rack_remote/rack-0.4
    * [new branch] rack-0.9 -> rack_remote/rack-0.9
    $ git checkout -b rack_branch rack_remote/master
    Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
    Switched to a new branch "rack_branch"
```

现在在你的`rack_branch`分支中就有了Rack项目的根目录，而你自己的项目在`master`分支中。如果你先检出其中一个然后另外一个，你会看到它们有不同的项目根目录：

```
$ ls
    AUTHORS KNOWN-ISSUES Rakefile contrib lib
    COPYING README bin example test
    $ git checkout master
    Switched to branch "master"
    $ ls
    README
```

要将 Rack 项目当作子目录拉取到你的`master`项目中。你可以在 Git 中用`git read-tree`来实现。你会在第9章学到更多与`read-tree`和它的朋友相关的东西，当前你会知道它读取一个分支的根目录树到当前的暂存区和工作目录。你只要切换回你的`master`分支，然后拉取`rack`分支到你主项目的`master`分支的`rack`子目录：

```
$ git read-tree --prefix=rack/ -u rack_branch
```

当你提交的时候，看起来就像你在那个子目录下拥有Rack的文件——就像你从一个tarball里拷贝的一样。有意思的是你可以比较容易地归并其中一个分支的变更到另外一个。因此，如果 Rack 项目更新了，你可以通过切换到那个分支并执行拉取来获得上游的变更：

```
$ git checkout rack_branch
    $ git pull
```

然后，你可以将那些变更归并回你的 master 分支。你可以使用`git merge -s subtree`，它会工作的很好；但是 Git 同时会把历史归并到一起，这可能不是你想要的。为了拉取变更并预置提交说明，需要在`-s subtree`策略选项的同时使用`--squash`和`--no-commit`选项。

```
$ git checkout master
    $ git merge --squash -s subtree --no-commit rack_branch
    Squash commit -- not updating HEAD
    Automatic merge went well; stopped before committing as requested
```

所有 Rack 项目的变更都被归并可以进行本地提交。你也可以做相反的事情——在你主分支的`rack`目录里进行变更然后归并回`rack_branch`分支，然后将它们提交给维护者或者推送到上游。

为了得到`rack`子目录和你`rack_branch`分支的区别——以决定你是否需要归并它们——你不能使用一般的`diff`命令。而是对你想比较的分支运行`git diff-tree`：

```
$ git diff-tree -p rack_branch
```

或者，为了比较你的`rack`子目录和服务器上你拉取时的`master`分支，你可以运行

```
$ git diff-tree -p rack_remote/master
```

6.8 总结

你已经看到了很多高级的工具，允许你更加精确地操控你的提交和暂存区。当你碰到问题时，你应该可以很容易找出是哪个分支什么时候由谁引入了它们。如果你想在项目中使用子项目，你也已经学会了一些方法来满足这些需求。到此，你应该能够完成日常里你需要用命令行在 Git 下做的大部分事情，并且感到比较顺手。

下一节上一节首页（目录）| 返回 码云