2 Git 基础

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

2.1 取得项目的 Git 仓库

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

在工作目录中初始化新仓库

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：

```
$ git init
```

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

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

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

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

从现有仓库克隆

如果想对某个开源项目出一份力，可以先把该项目的 Git 仓库复制一份出来，这就需要用到 `git clone` 命令。如果你熟悉其他的 VCS 比如 Subversion，你可能已经注意到这里使用的是 `clone` 而不是 `checkout`。这是个非常重要的差别，Git 收取的是项目历史的所有数据（每一个文件的每一个版本），服务器上有的数据克隆之后本地也都有了。实际上，即便服务器的磁盘发生故障，用任何一个克隆出来的客户端都可以重建服务器上的仓库，回到当初克隆时的状态（虽然可能会丢失某些服务器端的挂钩设置，但所有版本的数据仍旧还在，有关细节请参考第四章）。

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

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

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

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

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

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

2.2 记录每次更新到仓库

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

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

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

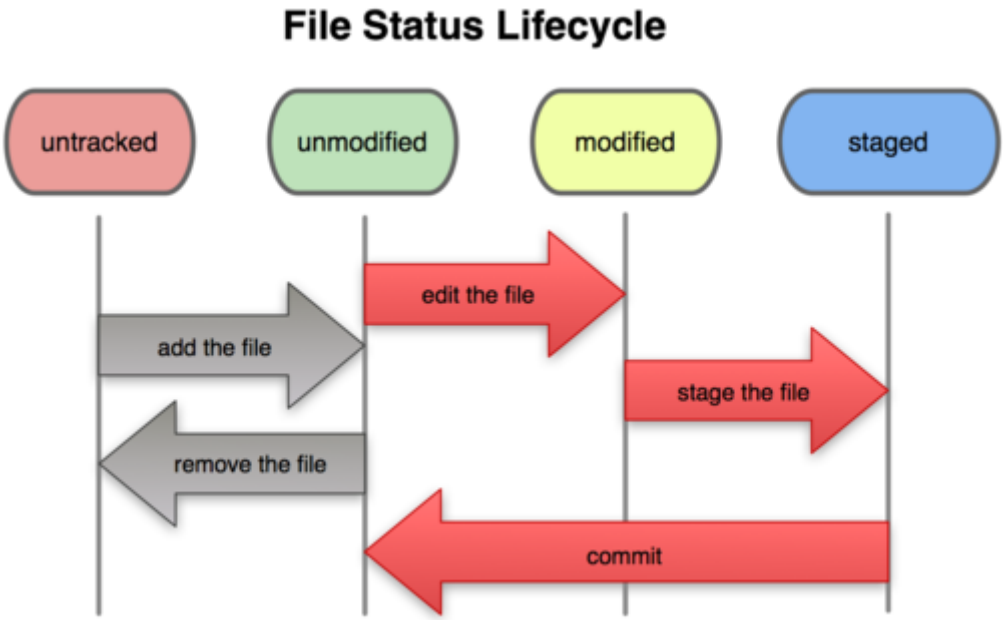

图 2-1. 文件的状态变化周期

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)
    #
    # 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)
    #
    # 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
```

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

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：

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

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

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

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

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

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

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

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

现在运行 `git diff` 看暂存前后的变化：

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

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

然后用 `git diff --cached` 查看已经暂存起来的变化：

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

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

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

提交更新

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

```
$ git commit
```

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

编辑器会显示类似下面的文本信息（本例选用 Vim 的屏显方式展示）：

```
# Please enter the commit message for your changes. Lines starting
    # with '#' will be ignored, and an empty message aborts the commit.
    # On branch master
    # Changes to be committed:
    # (use "git reset HEAD <file>..." to unstage)
    #
    # new file: README
    # modified: benchmarks.rb
    ~
    ~
    ~
    ".git/COMMIT_EDITMSG" 10L, 283C
```

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

另外也可以用 -m 参数后跟提交说明的方式，在一行命令中提交更新：

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

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

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

跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。Git 提供了一个跳过使用暂存区域的方式，只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤：

```
$ git status
    # On branch master
    #
    # Changes not staged for commit:
    #
    # modified: benchmarks.rb
    #
    $ git commit -a -m 'added new benchmarks'
    [master 83e38c7] added new benchmarks
    1 files changed, 5 insertions(+), 0 deletions(-)
```

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

移除文件

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

如果只是简单地从工作目录中手工删除文件，运行 `git status` 时就会在 “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)
    #
    # deleted: grit.gemspec
    #
```

然后再运行 `git rm` 记录此次移除文件的操作：

```
$ git rm grit.gemspec
    rm 'grit.gemspec'
    $ git status
    # On branch master
    #
    # Changes to be committed:
    # (use "git reset HEAD <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
    # Your branch is ahead of 'origin/master' by 1 commit.
    #
    # 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。

2.3 查看提交历史

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

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

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

然后在此项目中运行 `git log`，应该会看到下面的输出：

```
$ git log
    commit ca82a6dff817ec66f44342007202690a93763949
    Author: Scott Chacon <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
    @@ -5,7 +5,7 @@ require 'rake/gempackagetask'
    spec = Gem::Specification.new do |s|
    - s.version = "0.1.0"
    + s.version = "0.1.1"
    s.author = "Scott Chacon"

    commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    Author: Scott Chacon <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
```

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

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

    changed the version number

    Rakefile | 2 +-
    1 files changed, 1 insertions(+), 1 deletions(-)

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

    removed unnecessary test code

    lib/simplegit.rb | 5 -----
    1 files changed, 0 insertions(+), 5 deletions(-)

    commit a11bef06a3f659402fe7563abf99ad00de2209e6
    Author: Scott Chacon <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(+), 0 deletions(-)
```

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

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

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

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

表 2-1 列出了常用的格式占位符写法及其代表的意义。

```
选项 说明
    %H 提交对象（commit）的完整哈希字串
    %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。
    --stat 显示每次更新的文件修改统计信息。
    --shortstat 只显示 --stat 中最后的行数修改添加移除统计。
    --name-only 仅在提交信息后显示已修改的文件清单。
    --name-status 显示新增、修改、删除的文件清单。
    --abbrev-commit 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
    --relative-date 使用较短的相对时间显示（比如，“2 weeks ago”）。
    --graph 显示 ASCII 图形表示的分支合并历史。
    --pretty 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和
format（后跟指定格式）。
```

限制输出长度

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

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

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

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

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

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

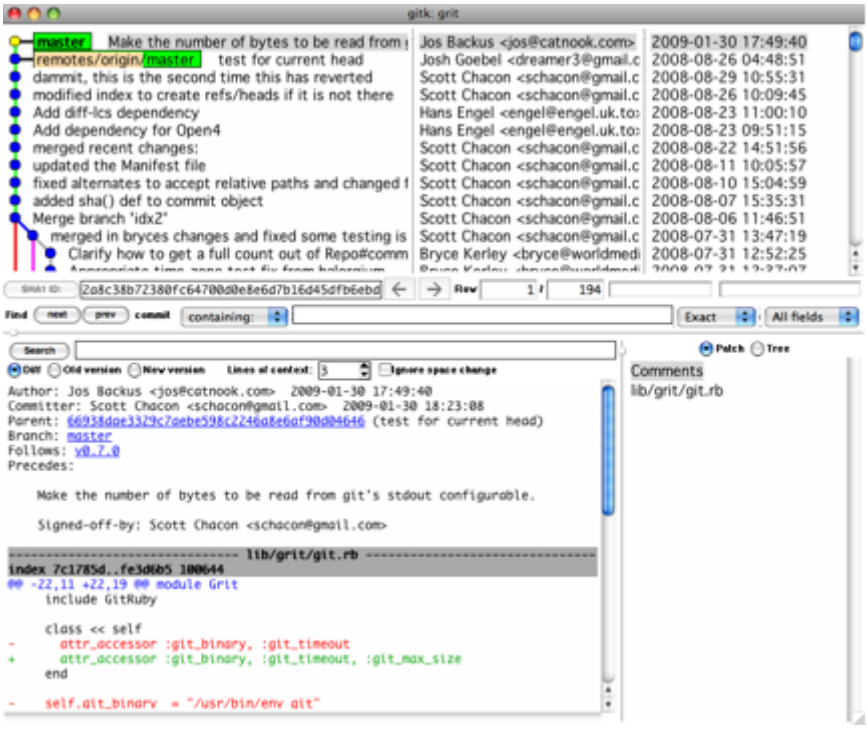

图 2-2. gitk 的图形界面

上半个窗口显示的是历次提交的分支祖先图谱，下半个窗口显示当前点选的提交对应的具体差异。

2.4 撤消操作

任何时候，你都有可能需要撤消刚才所做的某些操作。接下来，我们会介绍一些基本的撤消操作相关的命令。请注意，有些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
    benchmarks.rb: locally modified
    $ 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
    #
```

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

取消对文件的修改

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

```
# 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（关于数据恢复的内容见第九章）。所以，你可能失去的数据，仅限于没有提交过的，对 Git 来说它们就像从未存在过一样。

2.5 远程仓库的使用

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

查看当前的远程库

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

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

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

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

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

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

这样一来，我就可以非常轻松地从这些用户的仓库中，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。有关推送数据到远程仓库的详细内容见第三章。

查看远程仓库信息

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

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

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

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

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

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

远程仓库的删除和重命名

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

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

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

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

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

2.6 打标签

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

列显已有的标签

列出现有标签的命令非常简单，直接运行 `git tag` 即可：

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

显示的标签按字母顺序排列，所以标签的先后并不表示重要程度的轻重。

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

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

新建标签

Git 使用的标签有两种类型：轻量级的（lightweight）和含附注的（annotated）。轻量级标签就像是个不会变化的分支，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'
```

稍后我们再学习如何验证已经签署的标签。

轻量级标签

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

```
$ git tag v1.4-lw
    $ git tag
    v0.1
    v1.3
    v1.4
    v1.4-lw
    v1.5
```

现在运行 `git show` 查看此标签信息，就只有相应的提交对象摘要：

```
$ git show v1.4-lw
    commit 15027957951b64cf874c3557a0f3547bd83b3ff6
    Merge: 4a447f7... a6b4c97...
    Author: Scott Chacon <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。

2.7 技巧和窍门

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

自动补全

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
```

这个技巧不错吧，可以节省很多输入和查阅文档的时间。

Git 命令别名

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

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

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

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

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

这样一来，下面的两条命令完全等同：

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

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

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

然后要看最后一次的提交信息，就变得简单多了：

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

2.8 小结

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

下一节上一节首页（目录）| 返回 码云