图灵电子书

Git 小书

准能帮你省下不少**时间**

#刘传君

版权信息

书名：Git小书

作者：刘传君

ISBN：EA022

目录

前言
准备
介绍
暂存区
撤销
分支
修订的标识
标签
多仓库
协议
    Git协议
    SSH 协议
    HTTP
后记： 信任网络

前言

献给

启明星闪亮的清晨

我和**Git**

一直以来，我的团队使用SVN作为代码版本管理工具。日复一日，提交、提交、再提交，每个产品版本开发完成会打个标签、偶尔回溯修订历史查找Bug，一切都简洁、快速、美好。SVN很棒，它逐步的替代掉了我司内之前的CVS、VCS等同类工具。

然后，Git出现。基于对新事物的渴望、以及追逐偶像的动力，我决定自己去试试它。我把独立进行的、处于商业化早期的研究项目放到github上，克隆到我几台工作机上，然后使用github desktop在Mac、Windows上访问git，并且暂存、提交、推送、拉取。github GUI当然不错，但是我喜欢命令行，因为简洁，不多占用窗口，也能力全面。于是，我就去学了命令行。我立马感觉Git不像我期望的那么酷。

概念多而繁杂，我的感受就是这样：SHA1? Hunk? 远程分支? 引用? 这都是啥啊。当然这只是难而已。真正让人大跌眼镜的是，默认情况下，不少命令的输出看起来有些碎碎念的心烦：比如常见的git status，它不但很啰嗦地显示状态，也会把文件撤销和提交的命令提示出来。不够酷啊！这还不算，当我觉得郁闷时，有人继续补刀，高调地宣布：学习Git的最好方式是理解它的实现。你怎么不去上天？

我丢下git命令行一段时间，然后再捡起来，再丢下，心理很矛盾，有时候觉得git没有大家说的那么好，有时候觉得是不是自己很笨，并且认为我是程序员，不想去碰这些house-keeping类的工作。然后，某天我比较空，完整地清理了下目前为止对git的看法，得出这样的结论：不管git困难还是简单，我这样浅尝辄止、动辄想要输出价值观的思路和做法，是永远也学不会的。

于是我决定放下令人裹足不前的自尊和骄傲，花费了4个月的时间，看了数百页的资料，做了无数的实验，在一台主机上搭建N个仓库互相推送拉取，然后我把分支、合并、rebase如同提交一样，变成我的日常工作。于是，我有了新的感受——正如c语言一样，Git有无数的缺点，它也在继续演进中，但是它的分布式仓库和分支用起来太舒服了，特别是对于经常需要远程工作的人：

1. 我可以在本地无需服务器就可以完成大部分常用的版本管理；
2. 可以随时因为某个想法而开一个分支，成功完成后合并到主干，或者失败直接丢弃，开分支除非主动推送否则根本不会影响到其他人；
3. 和他人协作只要通过接受pull request即可接受别人的贡献，而无需给他开一个账号。

这样自我折腾后，回头再看Linus在Google做的Git专题演讲(2007年)，我秒懂了他提到的信任网络。我会在后记内就此话题多说几句。

Git的复杂度

掌握Git并不容易。和SVN相比的话，Git的复杂程度翻了几倍。这不可避免地会令人怀疑且裹足不前。

幸好，Git的复杂是可以划分的。首先是本质复杂度。本质复杂度是不可避免的。多引入一个实体，就意味着同时引入一组概念、以及它和老实体之间的一组关系，所以Git就是更难，哪怕封装得再好也是。作为是一个分布式的、单机可分支的工具，它引入了新的实体对象必然比SVN复杂，比如：

1. 为了管理提交的颗粒度，它引入了暂存区（stage），引入了块（Hunk）；
2. 为了能够分布，它的提交标示符无法采用简单数字，而是采用了一个看起来令人困惑的字符串（SHA1)。

另外，它也引入了不少非本质复杂度。此类型复杂度在于设计上的缺陷。比如：

1. 有些命令引入的功能太多。看看git reset子命令即可了解我的意思。

2. 概念命名随意。比如暂存区有多个名字，包括Index，Stage，Cache。
3. 参数设计随意。例如git branch -b等价于git branch后跟git checkout。

Git从一个本不知名的工具，到如今的风头正健，本身依然还在演进之中，非本质复杂度会随着这个过程而逐步降低。

Git刷新了我对版本工具的传统认识——充分利用分布特性来减少依赖，使用分支特性随心创造而无后顾之忧。所以，我认为我花费了不少时间来学习Git是值得的。

关于作者

作者：刘传君

创建过产品，创过业。不好动，读书机器。倾慕unix哲学，以此书略表致敬。

可以通过 1000copy#gmail.com 联系到我。

准备

安装

因为有些操作系统已经内置了git，所以可以使用命令来做验证：

```
git --verion
```

如果打印出了版本就说明已经安装了。如果出现的是一个错误信息，就需要安装它。我日常工作的电脑是OS X，可以使用brew来安装它：

```
brew install git
```

Linux也是内置的。如果没有，可以实验apt-get命令安装。如果是Windows，那么你可以下载Git for Windows二进制安装包，并且像是其他Windows应用一样点击下一步、下一步、或许还有N个下一步，然后完成。

本书使用的Git为2.7 ：

```
git --version

git version 2.7.4 (Apple Git-66)
```

案例都是在此版本样验证通过的，我并没有在更低版本上做过验证，因此建议你使用当前Git的最高版本。

配置

使用Git之前，它需要知道使用它的用户是谁，已经如何联系。可以使用如下的命令，配置用户名称和电子邮件：

```
git config --global user.name &amp;quot;Your Name&amp;quot;
git config --global user.email &amp;quot;your_email@whatever.com&amp;quot;
```

这两个参数会在每次提交时记录到历史内；当推送到其他仓库时，接收者也可以由此得知推送者是谁以及如何联系他，因此对于多人协作而言，这一组配置是非常重要的。我的配置是这样的：

```
git config --global user.name &amp;quot;1000copy&amp;quot;
git config --global user.email &amp;quot;1000copy@gmail.com&amp;quot;
```

如果你只是验证我的案例，不妨使用我的配置。如果在正式使用Git，你应该按照自己的情况来做配置。

创建和修改文件

为了演示Git的能力和方法，我常常会使用形如：

```
Line1
Line2
Line3
```

这样的内容文件作为版本管理对象。我使用命令行对这样的文件进行创建和修改，常用的命令就是echo命令和sed命令。

创建文件

我们可以使用echo创建文件:

```
echo Line1 > file
```

echo命令会把随后的参数作为内容写入> 之后指定的文件，如果没有就创建，否则覆盖。

添加行到文件

可以在现有文件中添加新行：

```
echo Line2 >> file
echo Line3 >> file
```

echo会把随后的参数指定的文本添加到>> 之后的文件的尾部。

修改文件

使用sed可以修改文件。比如说要修改file文件内的Line1为LineI，那么只要：

```
sed 's/Line1/LineI/g' file
```

sed 是Stream EDitor的缩写，它可以对文件做修改。具体的语法和使用请看手册页。

在OS X上运行的话，sed命令必须通过参数**-i** 指定备份文件，以便误操作后使用备份文件恢复。但是在本书的实验中并无此必要，因此可以传递一个空字符串给它，关掉这个特性：

```
sed -i '' 's/line1/lineI/g' file
```

介绍

本章会通过一些基本Git命令来完成一个基本的、但是完整的Git工作流程:

1. 创建Git版本仓库（Repository）；
2. 创建文件，把它加入跟踪，并多次修改、暂存、提交文件改变到仓库；
3. 查看版本修订历史，查阅修订的相对变化。

在此过程中，我会引出Git的最基本的概念，如仓库、工作区、暂存、提交、修订等。现在开始实验。

创建版本仓库

在你的电脑上找到或者创建一个空目录，把它作为工作目录（我的目录是workbench），进入工作目录后，执行：

```
git init pot
```

命令 git-init 用来从头创建一个新仓库，参数为仓库名，这里定义仓库名为pot。命令成功执行后在workbench目录内产生了一个叫做pot的新目录。我们看看pot内有什么：

```
cd pot
ls -A
```

输出：

```
.git
```

可以看到一个名为.git的隐藏目录。对于此目录我们只要知道：Git把本

仓库的所有版本跟踪信息都放置到此目录内。除了`.git` 目录外，在pot内的全部文件和目录整体构成用户的工作区。你可以在这里完成创建、修改和删除等文件操作。

创建文件

我们已经在准备 一章提到，echo可以用来创建文件和在文件尾部添加内容。现在我们使用echo来创建文件，并加入一行文字：

```
echo line1 > file1
```

我们可以使用git-status命令来查看版本仓库的状态：

```
$ git status -s
?? file1
```

命令git-status输出有一行文字，被空格分开成两部分，分别为文件所处状态和文件名，此处的文件状态为`??` 表示未跟踪状态。这个命令行输出告诉我们，文件file1处于未跟踪状态。

跟踪新文件，提交修改

可以通过git-add命令把新文件加入跟踪：

```
git add file1
```

再次使用git-status命令来查看状态：

```
$ git status -s
A file1
```

命令行输出的`A` 表示状态为已添加 (added)，就是说，file1已经被加入跟踪了。

现在使用git-commit做提交：

```
git commit -m&amp;quot;init&amp;quot;
```

命令git-commit需要参数-m指定提交消息，这里的提交消息是`init`，它由双引号包括起来以便可以包含空格。

我们再做一次修改，然后暂存并提交：

```
echo line2 >> file1
git add file1
git commit -m&amp;quot;r1&amp;quot;
```

喂，等等。这里读者可能会有疑问：目前总共做了2次提交（commit），第二次提交还是需要git-add来添加。这就奇怪了：不是在第一次提交时就把file1加入跟踪了吗？

两次git-add命令看起来一模一样，但是因为文件当前所处状态的不同，而导致它做的事情并不相同：

1. 第一次git-add，是把未跟踪的文件加入暂存区；
2. 第二次git-add，是把文件的修改加入暂存区。

Git引入了暂存区 的概念，git-commit执行时，仅仅把已经放置入暂存区的文件提交到仓库；文件要想被提交，首先需要通过git-add加入暂存区。就是说。Git是有这样的提交三部曲的：修改或者创建新文件、加入暂存区、提交到仓库。

好了，现在我们知道Git会在提交之前加入一个叫做暂存 的环节了。我们先记住这样的结论，随后章节会对暂存这个环节的价值做进一步分析。

现在我们再做一次修改，然后暂存并提交：

```
echo line3 >> file1
git add file1
git commit -m&amp;quot;r2&amp;quot;
```

看看仓库修订

现在停下来，稍作思绪的清理：我们做了三次提交，每个修改文件加入一行，提交消息分别为init、r1、r2。实际上仓库里面果然有这些修订吗？我们可以使用`git-log`来做验证。此命令会以修订时间为依据做反序排序，打印所有修订的概要信息。

```
$ git log  head -1
```

在此命令中，我们：

1. 第一个参数是head，它被用来指向最近一次修订。这个特殊符号是引用 的一种。类似的，分支名称也是引用 的一种
2. 使用了参数`-N`来指定输出的总数量，其中的N可以是任何一个正整数

这样，此命令就容易看懂了：它从最近一次修订开始，列出1个修订的信息。

输出信息看起来是这样的：

```
commit 6bc5e700ef6cd8c1b3ed3dd015ab8f4e9df67411
Author: 1000copy <1000copy@gmail.com>
Date:   Thu May 19 20:24:03 2016 +0800

    r2
```

输出的解析如下：

1. 第一行以Commit开始，跟着一个40位的字符串作为标识符。表明它是一个提交对象（commit）。对象标识为d2fe108e92df8d827f6ec237db85693dbd6a1eab。
2. 第二行显示的是本次修订的作者和电子邮件。作者是1000copy。正是我使用git-config命令配置的作者的值。电子邮件为1000copy@gmail.com。正是我使用git-config命令配置的电子邮件的

值。
3. 第三行是本次修订的日期时间。
4. 接下来一个空行随后就是本次修订的消息。我们的第三次提交，消息为r2。

我们随后会单独谈谈那个长长的40位字符串。现在你只要知道:

1. 此字符串用来唯一表示一个修订。
2. 此字符串每次提交是不同的。同样的文件内容，在不同的用户、不同的时间它也是不同的。

这意味着，你在你的电脑上执行我给出的命令，这个标识符虽然也是40位字符长度，但是值一定是和我不同的。因此，在接下来的命令中如果需要把40位标识符作为参数运行命令的话，那么需要替换为你的git-log输出的对应的标识符，而不可以直接照搬我的。

我们使用head表示最近修订，也可以使用最近修订的标识符来指向修订，因此：

```
$ git log  head -1
```

在当前的场景下也可以写成：

```
$ git log  6bc5e700ef6cd8c1b3ed3dd015ab8f4e9df67411 -1
```

事实上，可以通过diff命令来验证两个命令的输出：

```
$ diff <(git log  head -1) <(git log  head -1)
```

执行完毕后，输出是空的，这表明在括号内的两个命令的输出完全一致。

要输出修订的信息，也可以使用git-show命令，并给命令加上参数`--quiet`。再次使用diff命令验证，空输出表明两个命令的输出完全相同：

```
$ diff <(git log  head -1) <(git show  head --quiet)
```

命令git-show可以显示一个指定修订的详细情况，如果不使用--quiet参数，就会显示此修订和它的前一个修订之间的差异。

我们到目前为止，共有三个修订在仓库内。我们不希望使用哪个丑陋的字符串做标识符，那么可以换用另一个表示法。Git使用head~N 这样的格式，来表示从最近提交开始倒数第N个修订，具体说：

1. 使用head指向最近修订。
2. 使用head~1 指示最近修订的前一个修订
3. 使用head~2 指示最近修订的前两个修订。

以此类推。

因此：

```
$ git show  head~1 --quiet
```

在当前共三个修订的场景下会打印第2次修订。

```
$ git show  head~2 --quiet
```

在当前共三个修订的场景下会打印第1次修订。

可以使用git-log 命令列出全部修订，输出：

```
commit 6bc5e700ef6cd8c1b3ed3dd015ab8f4e9df67411
Author: 1000copy <1000copy@gmail.com>
Date:   Thu May 19 20:24:03 2016 +0800

    r2

commit 1113041465c48519fc5a46eb0bed004af6d6c0d0
Author: 1000copy <1000copy@gmail.com>
Date:   Thu May 19 20:24:03 2016 +0800

    r1
```

```
commit 4c34c35cb760a71cd811b6defc9bf18b703d34fd
Author: 1000copy <1000copy@gmail.com>
Date:   Thu May 19 20:17:25 2016 +0800

    init
```

它简单地把全部3次修订反序打印出来。要是我们希望每次执行git-log，只打印修订标识符和消息，而不显示作者和邮件（毕竟单独的仓库内，这些信息在每次修订显示时基本都是一样的），可以加上参数--pretty=oneline：

```
git log  --pretty=oneline
```

输出：

```
6bc5e700ef6cd8c1b3ed3dd015ab8f4e9df67411 r2
1113041465c48519fc5a46eb0bed004af6d6c0d0 r1
4c34c35cb760a71cd811b6defc9bf18b703d34fd init
```

这样的输出会更加简明一点。我们可以看到，输出共有三行，每行两列，每一行代表一个修订，按照创建时间倒序显示。第一列为修订标示符，因为它是Git使用了SHA1算法来生成的，因此也会称它为SHA1标识。第二列为提交消息。

查看差异

可以使用git-diff命令来查看两个修订中某一文件的的差异，比如:

```
$ git diff head~1 head  file1
```

意思是查看file1文件在当前修订和前一个修订中的差异。输出是：

```
diff --git a/file1 b/file1
index c0d0fb4..83db48f 100644
--- a/file1
```

```
+++ b/file1
@@ -1,2 +1,3 @@
 line1
 line2
+line3
```

可以看到，输出中给最后一行的+Line3 表示加入了一个新行，行内容为line3。

缩写的修订标识符

回头来看看这次的命令git log --pretty=oneline 的输出，看起来修订标示符也太长了。幸好它是可以缩写的：

```
git log --abbrev-commit --pretty=oneline

6bc5e70 r2
1113041 r1
4c34c35 init
```

选项--abbrev-commit 指示缩写Commit的标识符值。这个值一般使用七个字符的缩写。不过有时为了避免缩写后可能的冲突，会增加字符数。通常8 到10个字符就已经足够在一个项目中避免标识符冲突。也可以使用 git log --oneline，效果相同。

使用这个缩写修订标识符可以引用对应的修订。比如，我可以使用6bc5e70作为参数来查看最后一次修订的信息：

```
git show 6bc5e70
```

当然缩写的修订标示符和完整的修订标识符是等效的：

```
diff <(git show 6bc5e700ef6cd8c1b3ed3dd015ab8f4e9df67411) <(git show 6bc5e7
```

更简易的**log**

我们一再看到修订标识符的丑模样，它对人类来说太不友好了。我们应该尽可能使用类似head，head~1这样的引用 去指代修订；类似的想法，是使用提交消息 来指代修订。在本书的案例中，提交消息都是经过有意的设计，以便看到它就可以知道它指代的修订。为此，使用一个仅仅输出提交消息 的命令是必要的。你只要给git-log一个`--pretty=format:'%s'` 参数即可达成期望：

我们可以做一个对比，当执行：

```
git log --abbrev-commit --pretty=oneline
```

得到如下结果时：

```
6bc5e70 r2
1113041 r1
4c34c35 init
```

对应的执行：

```
git log --pretty=format:'%s'
```

输出的就只有提交消息，每个修订一行：

```
r2
r1
init
```

使用此命令和参数的配合，足够让我们避开修订标示符的编写和阅读负担，而不影响我清晰地传递案例中的修订历史给你了。

父修订

我们已经创建了一个仓库，其中有三个修订。且我多次提到了修订的前一个修订 这样的说法。在Git系统内，指代前一个修订是有特定名词的——它叫做父修订 。我们共三次修订，每个修订(除了第一个）都是依赖于它的父修订的。 为了书写方便，我们使用每个修订的对应提交消息来指代此修订，图示为这样的关系：

```
init <- r1 <- r2
```

修订init是第一次修订，也叫做初始修订，它是没有父修订的。修订r2的父修订为r1，相应地，修订r1的父修订为init。

除了初始修订外，我们现在看到的每个修订都有且只有一个父修订。在Git系统中，一个修订是可以有多个父修订的，这种情况出现在分支合并的场景下，会在分支 一章来详细说明。

命令缩写

我们一直在使用带有很长参数的`git-log` 命令来查看提交历史，这样很麻烦。幸好Git可以提供命令别名，从而使用短命令来替代比较长的命令和对应的参数。

hist

使用别名定义可以把常用的`git-log` 调整为简短的别名。执行此命令：

```
git config --global alias.hist 'log --pretty=format:&quot;%h %ad | %s%d [%a
```

我们不再使用oneline为参数`--pretty` 的值，而是使用format来做字符串格式化的输出。其中的：

```
%h 为提交的SHA1的缩写值
%ad 为日期
%s  为提交消息
%d  引用名，一般显示分支名，以及head指向的分支名
%an 为作者
--graph 图形化输出分支
```

使用`git config -l` 可以查阅所有配置。如果发现如下行就可以验证我们的配置已经成功：

```
alias.hist=log --pretty=format:&quot;%h %ad | %s%d [%an]&quot; --graph --da
```

执行如下命令，来感受一下配置后的效果：

```
git hist -2
```

输出：

```
* b638182 2016-05-31 | r2 (HEAD -> roma, master) [1000copy]
* 8743400 2016-05-31 | r1 [1000copy]
```

配置后`git-log` 本来可以使用的参数hist，命令内可以继续使用。比如`-2` 限定输入的修订历史条目数不超过2。

mist

命令git-hist的输出是全面的，然而也是冗余的。我们在本书内为了简便（避开巨长的修订标识符），会使用这样的命令和参数的组合：

```
git log --pretty=format:'%s'
```

我也会给它配置一个别名：

```
git config --global alias.mist 'log --pretty=format:&quot;%s&quot;'
```

此命令仅仅列出每个修订的提交消息:

```
git mist -2
```

```
r2
r1
```

随后章节也会采用**git-mist** 命令做历史查询，特别是无需修订标识符即可说明问题的场合。

暂存区

我们已经了解到了Git的“修改、暂存、提交”三部曲，所有需要提交到仓库的修改首先需要加入暂存区。那么，问题就来了：我干么需要暂存区而不是直接提交呢？

存在的必要性

要明白这一安排的必要，我们先得弄明白，任何一个版本管理工具在选择提交本地修改时有一个基本的原则：每一次提交，应尽量包含且仅包含一个功能或者一个Bug修正。这样做的好处就是，在回溯历史修订的时候，找到的那次修订就是你想要的功能特性或者Bug修改涉及到的代码修改。

遵循这样的原则，我们来看一个典型的工作场景。你今天上班一直在进行一个功能的代码编写，当快要完成此功能时，临时发现有一个Bug，顺手也就把它做了修正，然后你去继续编写并完成功能。快要下班时，通过git-status命令列出修改的文件清单，假设是这样的：
git status -s M bugfile1 M bugfile2 M featurefile1 M featurefile1

其中有几个文件的修改是为了Bug修正，另外几个文件的修改则是为了功能特性。按照前述的原则，你当然应该把它们分两次提交。你可以

```
git add bugfile1 bugfile2
git commit -m&amp;quot;bug fix for client &amp;quot;
```

然后：

```
git add featurefile1 featurefile2
git commit -m&amp;quot;great feature &amp;quot;
```

有了暂存区，我们可以在其中加入我们待提交的修改，然后看看不对的话，还可以从暂存区内移除这个修改，直到对清单完全满意才真正的提

交。暂存区的概念特别像是超市的购物车，你可以把喜欢的货品丢进去，走着走着，看到更好的就再丢进去，把现在认为不够好的拿出去，或者改主意了就取出其中的某些货品。有了暂存区，对修改进行挑挑拣拣就变得很轻松，像是这样：

```
# 添加 bugfile1
git add bugfile1

# 添加 featurefile1
git add featurefile1

# 这是功能修改，不该放到这里，把它拿出来
git reset featurefile1

# 添加 bugfile2
git add bugfile2
git commit -m&quot;bug fix for client &quot;
```

这里提到了git-reset命令，它可以把修改从暂存区移除。我们会在撤销一节对它做进一步介绍。

有了暂存区，我们可以把当前的工作区修改分为几组放入暂存区然后提交。这就是Git常常被提到的分批提交的概念。

Hunk拆分

上节的分批提交 仅仅演示到文件层面。实际上，Git还提供了比文件更细的提交颗粒度。就是说，在一个文件内的多个修改点，也可以被拆分后分别暂存。Git称这样在文件内的更细颗粒度的修改为块（hunk）。

假设我们的修改都在一个文件内，文件名为file：

1. 修改文件，开发功能
2. 修改过程中在同一文件内发现有个小bug，顺手改掉，此修改为B
3. 功能完成了，先把功能所属的修改行暂存起来然后提交
4. 其他修改再次被暂存，然后作为另外一个提交

最后得到一个功能和Bug修正分开为两次修订的历史。

来个实验。我们创建并进入仓库，然后创建文件并完成初始提交：

```
echo line1  > file
echo line2 >>file
echo line3 >> file
echo line4 >> file

git add file
git commit -m&amp;quot;init&amp;quot;
```

修改1为了feature A，

```
sed -i '' 's/line1/lineI/g' file
```

修改2为了Bug B，
sed -i '' 's/line4/lineIIII/g' file

于是，在一个文件内的有2处修改，逻辑上分属Bug修订和功能特性。它们是可以作为不同的提交的。做法是使用git-add命令，加入-p参数：

```
git add -p
```

命令执行后，会显示当前工作区内文件file自上次提交以来的修改:

```
...
@@ -1,4 +1,4 @@
-line1
+lineI
 line2
 line3
-line4
+lineIIII
```

并等待键入一个子命令:

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

在提示的[y,n,q,a,d,/,s,e,?] 内显示可以用的单字母命令，其中的 s 子命令可以拆分的意思。我们在此场景下，输入s字符，即代码拆分单文件内的修改为两个Hunk:

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? s
```

输出：

```
Split into 2 hunks.
...
```

表明已经将本文件内的2处修改划分为2个块。接下来可以通过g命令（goto）列出Hunk清单和对应编号:

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? g
  1:  -1,3 +1,3          -line1
  2:  -2,3 +2,3          -line4
```

输入字符1，表示跳到第1个Hunk：

```
go to which hunk? 1
```

输入字符y（yes），即可暂存当前块：

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
```

Git随即显示接下来的块，并继续询问命令。本次交互式暂存，我们已经把Bug修正涉及得到的修改（hunk）暂存完毕，因此可以退出此交互模式。输入字符q（quit）退出本次交互：

```
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? q
```

查询下状态：

```
 git status -s
```

输出：
MM file

表明file同时处于修改和暂存状态。这是一个有趣的状态显示，它的意思是file文件的修改有些已经暂存，有些还没有。

现在提交暂存区修改到feature A：

```
git commit -m&amp;quot;feature A&amp;quot;
```

再来一次交互式添加：

```
git add -p
```

输出会显示文件file和最近一次前的修改：

```
...
@@ -1,4 +1,4 @@
 lineI
 line2
 line3
-line4
+lineIIII
```

并且通过询问，期望用户输入命令来处理此块：

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

通过y命令把剩下的修改加入到暂存区。然后提交此块：

```
git commit -m&amp;quot;bug b&amp;quot;
```

查看历史

```
git mist

bug b
feature A
init
```

仓库内后两次修订，其实是把同一个文件内的不同修改点通过交互模式切分为两次修改，并分别暂存和提交。

对**git-add**的常见误解

命令git-add的功能常常被解释为把文件添加到暂存区。然而这个说法是不够准确的。其实它暂存的是修改而不是文件。这话怎么解释？我们依然用实验来说明问题。

创建实验环境。首先创建file2，并暂存、再修改、提交：

```
echo line1 > file2
git add file2
echo line2 >> file2
git commit -m&quot;stage testcase&quot;
```

提交后再看看状态：

```
git status -s
```

输出：
M: file2

感觉不太对：我已经暂存file2了，怎么它还是修改状态?

同样是一个文件，第一次修改被暂存了，随后此暂存被提交，这个流程是正确的。而第二次修改并没有通过命令git-add暂存，因此随后的git-commit并不能把此修改一并提交，所以你看到了，它依然是修改状态。

可以执行如下命令：

```
git diff
```

验证第二次修改（+line2）是没有被提交：

```
...
@@ -1 +1,2 @@
 line1
+line2
```

要想第二次修改也可以被提交，记得先暂存：

```
git add file2
git commit -m&amp;quot;stage2&amp;quot;
```

提交的简化

很多时候你希望把工作区的修改作为整体一起提交。我们可以让命令git-commit帮忙，只要给它提供参数`-a`，它会把本地的所有修改首先暂存，然后提交。

我们重新创建一个仓库p2作为实验环境：
git init p2 cd p2 echo line1 > file1 git add file1 git commit file1 echo line2 >> file1

即可组合提交此次修改：

```
git commit -m&amp;quot;line2&amp;quot;  -a
```

但是它不能把未跟踪文件也一起提交。因此如果文件没有加入跟踪，就得首先使用命令git-add把它加入跟踪。

撤销

Git可以撤销操作。它让常常会犯错的我们可以吃个后悔药，或者坐上时光机回到过去。本章我们会逐个看曾经学习的命令操作是如何被撤销的。

撤销工作区文件修改

我们可能在工作区内做了一番修改后才发现这些修改是错误的，丢弃这些修改是最简单的重来的方法。此时可以使用git-checkout命令来丢弃本地修改。

依然用实验说明问题。首先，创建并进入仓库：

```
git init p1 && cd p1
```

创建3个文件并提交到仓库内：

```
echo line1 > file1
echo line1 > file2
echo line1 > file3
git add file1 file2 file3
git commit -m&quot;initial&quot;
```

再次修改。使用命令验证状态：

```
echo line2>> file1
echo line2>> file2
echo line2>> file3
git status -s

M file1
M file2
M file3
```

随即发现这次对file1的修改是错误的，我想重头再来。那么就可以丢弃此次修改：

```
git checkout -- file1
```

命令git-checkout可以传递一个或者多个文件名作为参数，然后这些被指定的文件中的修改会被全部丢弃。

然而参数“--”有点奇怪。git-checkout命令有多重能力，把文件名当成参数传给它的话，它会完成丢弃修改的功能；如果把分支名当成参数传递给它的话，它会完成分支切换功能。此符号的存在正是为了区别两种情况，Git会知道，在"--"后面出现的参数是文件名称,而不是分支名称。分支的概念我们将来再提。

查询状态：

```
git status -s
```

输出：

```
M file2
M file3
```

证明file已经不在已修改状态了。使用cat file1可以此文件的内容已经恢复到上一次提交的内容。

要是你发现你修改了很多文件，这些修改全部都是不应该的，想要完全重头再来，那么可以传递.给git-checkout命令：

```
git checkout -- .
```

查询状态： git status -s 是没有任何输出的。这说明全部修改都已经被成功的丢弃了。需要注意的是，这里的“." 指示为整个目录，包括它的所有子目录以及更深目录的文件的。下面的实验可以对此参数的影响范围做一个验证。

首先，创建并进入仓库：

```
git init p2 && cd p2
```

执行以下命令：

```
echo line1 > file1
echo line1 > file2
mkdir d1
echo line1 > d1/file11
git add .
git commit -m&amp;quot;init&amp;quot;
```

随后我们继续修改文件：

```
echo line2 >> file1
echo line2 >> d1/file11
```

现在我们可以撤销整个工作区的修改：

```
git checkout -- .
```

执行git status -s，发现没有任何输出，说明我们已经舍弃完成。

命令git-checkout是非常危险的，因为一旦撤销完成就无法再还原你的修改了。

把文件移出暂存区

使用git-reset命令，可以从暂存区把文件移出来。如果你在暂存区内放入了本次不应该暂存的修改，你就需要git-reset来帮忙了。

我们来验证。创建并进入仓库：

```
git init p3 && cd p3
```

创建并暂存3个文件：

```
echo line1 > file1
echo line1 > file2
echo line1 > file3
git add file1 file2 file3
git status -s
```

现在暂存内有三个文件：

```
A  file1
A  file2
A  file3
```

此时我们如果需要把file1移除暂存区，那么：

```
git reset file1
```

使用 git status -s 可以看到

```
A  file2
A  file3
?? file1
```

表示文件再次回到未跟踪状态。

如果不指定文件参数，命令git-reset会把暂存区完全清空：

```
git reset
git status -s
```

输出： ?? file1 ?? file2 ?? file3

撤销提交

即使修改已经提交到仓库，这个操作也是可以撤销的。

看实验。创建并进入仓库：

```
git init p4 && cd p4
```

并创建一个文件，把它做三次修改和提交：

```
echo line1 > file1
git add .
git commit -m&amp;quot;r1&amp;quot;
echo line2 >> file1
git commit -m&amp;quot;r2&amp;quot; -a
echo line3 >> file1
git commit -m&amp;quot;r3&amp;quot; -a
```

此时仓库内的file1内容为

```
line1
line2
line3
```

我们可以使用命令来查看最后一个提交：

```
git mist
```

输出:

```
r3
r2
r1
```

提交完后，我发现最后一次提交时的修改并不恰当，得把它从修订历史

中移除，以免对合作者引发误导。那么：那么命令git-reset就可以派上用场，只要把修订引用传递给命令，它会把历史撤销到指定的修订处：

```
git reset head~1
```

验证历史：

```
git mist
```

输出：

```
r2
r1
```

表明第三次提交已经被撤销。你可以通过

```
git diff
```

了解仓库内文件和本地文件的差异：

```
@@ -1,2 +1,3 @@
 line1
 line2
+line3
```

因为撤销的缘故，新添加line3的文本文件回到了“未提交”的状态。

恢复撤销

如果你撤销了提交之后，随即发现自己再次错了，那么随时可以使用git-reset把撤销本身给撤销掉。以当前的仓库为例，我们只要把本来的第三次修订传递给git-reset命令即可。问题是当我们做了git-reset之后我们已经查询到本来的修订历史了。

git提供了命令git-reflog。它可以列出全部的操作，包括撤销操作。回忆一下，我们做了三次提交（Commit），一次撤销（reset）。现在试试此命令：

```
git reflog
```

输出：

```
6c42ded HEAD@{0}: reset: moving to head^
3077323 HEAD@{1}: commit: r3
4639b19 HEAD@{2}: commit: r2
11e5868 HEAD@{3}: commit (initial): r1
```

记录了我们刚刚提到的全部3次提交和一次撤销（git-reset）！你不但可以从这里查到第三次修订的标示符（3077323），还可以第二列中形如HEAD{N}的符号。HEAD@{}内的数字N，就是git-reflog列表中的次序，从0开始计算。所以，你还可以:

```
git reset head@{1}
```

完成对撤销的撤销。现在执行： git mist 输出： r3 r2 r1

喔噢，被撤销的修订历史r3，现在回来了。

撤销提交内的部分文件

每次提交可以包含多个文件，如果我们只要撤销其中一个（部分）文件，怎么办？

看实验。创建并进入仓库：

```
git init p5 && cd p5
```

创建两个文件，暂存并提交，再次修改并暂存提交：

```
echo line1 > file1
echo line1 > file2
git add file1 file2
git commit -m&amp;quot;initial&amp;quot;

echo line2 >> file1
echo line2 >> file2
git commit -m&amp;quot;revision 2&amp;quot;  -a
```

取消整个提交，然后添加全部文件，并把file1从暂存区移除：

```
git reset head~
git add .
git reset file1
git commit -m&amp;quot;revision 2&amp;quot;
```

查看状态；

```
git status -s
```

符合期望：

```
 M file1
```

分支

在一个Git仓库内，程序员可以创建和工作于多个分支，各个分支之间是隔离的。也就是说，在某个分支上进行的任何修改、暂存、提交都不会影响到其他分支。本节内通过实验来创建分支、在分支上的修改提交、合并分支、删除分支。

假设我们已经有了3行文本文件，每一行都是line+一个阿拉伯数字：

```
line1
line2
line3
```

现在，我希望把代码中的阿拉伯数字改为罗马数字。最终会改成这样：

```
lineI
lineII
lineIII
```

我们来看如何利用Git分支命令来完成此项需求变更。

创建分支

首先，创建并进入仓库：

```
git init p1 && cd p1
```

准备阿拉伯数字风格的文件，并提交到仓库内：

```
echo line1  > file1
echo line2 >> file1
echo line3 >> file1
git add .
git commit -m&amp;quot;init&amp;quot;
```

首先，我们查看下分支列表:

```
git branch
```

输出：

```
* master
```

命令git-branch在不加参数的情况下，可以把当前仓库的所有分支打印出来。我们在输出中看到当前仓库仅有一个分支，名字为`master` 。在分支名称前的* 表示此分支为当前分支。分支master为默认分支，在创建仓库时就已经被创建，在不涉及多分支的情况下，我们的所有提交都是提交到此分支上的。

我们然后创建并切换到新分支：

```
git checkout -b roma
```

命令git-checkout用来切换分支，加上`-b` 参数要求在切换之前首先创建分支，随后的参数给出分支的名字，此处的新分支名字为roma。

再次打印仓库的全部分支：

```
git branch
```

输出：

```
* roma
  master
```

现在我们拥有两个分支。当前分支为roma，是这次为了分支开发而刚刚创建的。

修改代码

现在我在roma分支上多次修改代码渐进目标。

修改、暂存、提交三部曲

```
sed -i '' 's/line1/lineI/g' file1
git add file1
git commit -m&amp;quot;roma 1&amp;quot;
```

再次三部曲

```
sed -i '' 's/line2/lineII/g' file1
git add file1
git commit -m&amp;quot;roma 2&amp;quot;
```

三次三部曲

```
sed -i '' 's/line3/lineIII/g' file1
git add file1
git commit -m&amp;quot;roma 3&amp;quot;
```

在新的分支roma上，我已经对文件file1完成了三次修改、暂存和提交，在仓库内的roma分支历史上已经有了4次修订：

```
git mist
```

输出：

```
roma 3
roma 2
roma 1
init
```

切换分支

现在，roma分支的工作完成。我们对修改很满意，所以决定合并此分支的修改到master上。为此，我切换回master分支：

```
git checkout master
```

查看下文件

```
cat file1
```

输出:

```
line1
line2
line3
```

显然，分支master上的file1还是阿拉伯数字版本的。无论我们在分支roma上做多少次的修改、暂存、提交，都不会因此影响到master分支内的修订，因此，Git提供的隔离分支开发特性是有效的。

现在，我们合并roma分支的成果到master主干上来：

```
git merge roma
```

命令git-merge用于合并指定分支（此处第一个参数：roma）到当前分支（master）：

```
cat file1
```

输出：

```
lineI
```

```
lineII
lineIII
```

roma分支上修改的内容已经合并到当前分支内。

分支roma已经合并到主线，此分支就不必留，所以可以删除它了：

```
git branch -d roma
```

命令git-branch的参数-d 表明删除随后参数指定的分支。于是分支roma被删除。

解决冲突

我们刚刚做的分支开发，因为仅仅有一个分支roma在修改，所以并不会在两个分支之间产生修改的冲突。但是，如果仓库内的多分支都有修改、且它们修改了同一块代码的话，在合并分支的时候必然会引发冲突。在分支开发过程中冲突一般是不可避免的，但是解决冲突并不困难。我们来做个实验，故意在分支开发中引发冲突并解决它。

首先，创建并进入仓库：

```
git init p2 && cd p2
```

修改、暂存、提交文件：

```
echo line1 > file1
echo line2 >> file1
git add file1
git commit -m&amp;quot;init&amp;quot;
```

创建并切换到分支roma上，修改阿拉伯数字为罗马数字：

```
git checkout -b roma
sed -i '' &amp;quot;s/line2/lineII/g&amp;quot; file1
```

```
git add file1
git commit -m&amp;quot;roma 1&amp;quot;
```

回到master分支上工作，修改阿拉伯数字为英文单词：

```
git checkout master
sed -i '' &amp;quot;s/line2/lineTwo/g&amp;quot; file1
git add file1
git commit -m&amp;quot;commit 2&amp;quot;
```

执行两个分支的合并，把roma分支合并到master分支上：

```
git merge roma
```

输出：

```
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

说明在执行合并命令时冲突发生了，并且git视图自动合并但是失败了。现在我们来查看下冲突后的文件的样子：

cat file1

输出：

```
line1
<<<<<<< HEAD
lineTwo
=======
lineII
>>>>>>> roma
```

要是不懂Git如何标记冲突的话，这个输出看起来会有点让人犯晕。所

以，首先我们得弄明白冲突标记方法。Git会在冲突文件内，使用特定的符号来标记冲突区：

1. 开始标识为：“<<<<<<<”标记冲突区开始，此行内这个标识后跟着当前分支名称(此处为HEAD)。
2. 结束标识为：“>>>>>>>”标记冲突区结束，此行内这个标识后跟随的是被合并的分支名(此处为roma)。
3. 在冲突区内，它内部再次被分隔符“=======”分为两个部分，分隔符之上为当前分支(master)的修改内容；分隔符之下为被合并分支(roma)修改的内容。

我们解决冲突的方式就是手工修改这个冲突区，改成我们本来希望的样子。

假设采用roma分支的修改，那么我们可以把整个冲突区删除，随即提交，从而完成冲突解决:

```
echo line1 > file1
echo lineII>> file1
git commit -m &quot;conflict solved&quot; -a
```

合并完毕后，分支的成果并合并到当前分支，但是被合并的分支依然存在：

```
git branch
```

输出： * master roma

如果你觉得此分支已经不再使用，你需要自己删除它。

rebase

命令git-rebase也可以合并一个分支的开发成果到另外一个分支。不同的是，被rebase的分支的历史会被整体搬移到当前分支上。

现在来构建一个有多分支的仓库作为实验环境，以此验证rebase的功

能。

创建并进入仓库：

```
git init p3 && cd p3
```

修改、暂存、提交：

```
echo line1 > file1
git add .
git commit -m&amp;quot;r1&amp;quot;
```

然后创建并切换到roma分支，随后修改暂存提交：

```
git checkout -b roma

echo lineI > file1
git commit -m&amp;quot;rI&amp;quot; -a
```

切换回master分支，修改暂存提交：

```
git checkout  master

echo line2 >> file1
git commit -m&amp;quot;r2&amp;quot; -a
```

我们可以通过git-log查看创造出来的环境。首先查看master分支：

```
git mist
```

输出：

```
r2
r1
```

随后，查看rome分支：

```
git checkout roma
git mist
```

输出：

```
rI
r1
```

环境构建完成。现在我要把roma分支上的开发成果合并到master之上：

```
git rebase roma
```

此命令（git-rebase)的提示信息极为冗长，不过目前我们只要关心其中一行：

```
CONFLICT (content): Merge conflict in file1
```

在file1文件内，我们依然遇到了冲突：

```
<<<<<<< 42f5367dd03582b0008627adebac8ca3d1ec509a
line1
line2
=======
lineI
>>>>>>> rI
```

此时如果执行分支查询命令：

```
git branch
```

会发现Git提示没有当前分支，因为rebase还没有完成：

```
* (no branch, rebasing master)
master
roma
```

现在我们解决冲突（为了方便我们直接写入合并后的新内容）：

```
echo line1  > file1
echo line2  >> file1
echo lineI  >> file1
```

然后加入此文件改变，使用git-add标记冲突处理完成，并带--continue参数执行rebase（表示继续未完成的rebase）：

```
git add file1
git rebase --continue
```

这样，整个rebase的过程完成。现在我们再来查看历史：

```
git mist
```

输出：

```
rI
r2
r1
```

这个输出说明，rI的父修订从r1变成了r2。git-rebase会把当前分支的全部修订搬移到指定分支上，两个分支的历史合并为一条单线。

撤销**git-rebase**

命令git-rebase也是可以撤销的。你只要执行命令git-reflog，从它的输出中找到你需要的撤销点。就以当前的操作环境为例:

```
git reflog
```

输出：

```
c726012 HEAD@{0}: rebase finished: returning to refs/heads/roma
c726012 HEAD@{1}: rebase: rI
b638182 HEAD@{2}: rebase: checkout master
f6ac70f HEAD@{3}: checkout: moving from master to roma
b638182 HEAD@{4}: commit: r2
8743400 HEAD@{5}: checkout: moving from roma to master
f6ac70f HEAD@{6}: commit: rI
8743400 HEAD@{7}: checkout: moving from master to roma
8743400 HEAD@{8}: commit (initial): r1
```

命令git-reflog列出了全部的操作历史，并且最新的操作列在最前面。现在我们希望恢复到rebase之前，也就是：

```
42f5367 HEAD@{4}: commit: r2
```

你可以使用缩写的修订标示符b638182，或者使用更加易懂的HEAD@{4}来指代撤销点，并把它作为参数传递给git-reset：

```
git reset --hard HEAD@{4}
```

使用参数`--hard` 告诉git-reset命令不仅修改仓库的当前修订位置，也会同时使用此修订的文件来覆盖工作区和暂存区的文件。

查看历史：

```
git mist
```

输出：
r2 r1

表明仓库修订回到r2。使用命令`$cat file1`，从而可以发现file1此修

订下的内容是妥当的：

```
line1
line2
```

修订的标识

仓库就是由彼此相关的一组修订构成。因此Git必须给出一种或者多种区别不同修订的方法，使用它们作为参数传递给需要定位修订的命令。比如当需要查看某个修订的信息时，就得把修订的定位方法传递给git-show命令。

我们之前已经提到，Git采用一个长度为40的、通过SHA1算法生成的字符串作为基础标识符，使用此标识符或者它的缩写可以唯一定位一个修订，因此：

1. 标示符
2. 缩写标识

是Git中最基础的修订标识方法。它存在的问题是本身毫无语义，难以阅读和辨识。为解决此问题，Git提供了引用标识法，作为被命名、有语义的标识符，可指向某个具体的修订。引用标示符包括：

1. head引用。指向当前分支的最近修订。
2. 分支引用。指向该分支的最近修订，比如master作为引用标识符，它指向分支master的最近一次修订。

我们已经多次使用引用来指向修订，这样做常常是比较便利的，特别是相对SHA1标示符而言。Git在基础标示符和引用标识符的基础上，提供了组合标识符：

1. "^" 组合标识。格式为 修订标识符+“^”+Number 。如果仓库存在分支合并，并导致某个修订有多个父修订的情况下，可以使用此方法指定该修订的第Number父修订。
2. “~” 组合标识。格式为修订标识符+“~”+Number 。可以使用此方法指定前Number个祖先修订。
3. “@” 组合标识。格式为修订标识符 +“@”+“{”+Number+"}" 。Git会记录所有操作（包括提交、撤销等）数月，这些操作由近及远以渐增数字次序记录。可以使用此方法指定倒数第Number个操作指向的修订。

所有的标志统称为修订标示符，它包括了基础标识符、引用标识符、组合标识符。这些文字并不好懂，幸好我们可以通过实验，简单明了地认知和学习这些概念。

环境准备

为了说明问题，我们需要构造一个不太简单的仓库:

```
git init p1 && cd p1
```

在分支master上提交两次，提交消息分别为r1,r2:

```
echo line1 > file1
git add .
git commit -m&amp;quot;r1&amp;quot;

echo line2 >> file1
git commit -m&amp;quot;r2&amp;quot; -a
```

创建roma分支，并在此分支上提交两次，提交消息分别为rI，rII:

```
git checkout -b roma

echo lineI > file1
git commit -m&amp;quot;rI&amp;quot; -a

echo lineII >> file1
git commit -m&amp;quot;rII&amp;quot; -a
```

回到master分支，提交一次，提交消息为r3:

```
git checkout  master

echo line3 >> file1
git commit -m&amp;quot;r3&amp;quot; -a
```

合并roma到master，处理冲突后提交，提交消息为m1:

```
git merge roma
// merge
git commit -m&amp;quot;m1&amp;quot; -a
```

提交到master，提交消息为m2:

```
echo line4 >> file1
git commit -m&amp;quot;m2&amp;quot; -a
```

我们做了这些工作会创建一个仓库和一组相关联的修订，以图形方式表达，就是这样：

```
master : r1 <-- r2 <--    r3      <-- m1 <-- m2
roma               <-- rI <-- rII <--
```

此图形中，我使用修订消息来标识不同的修订，这样做确实并不严谨，但是可以非常方便地传递Git知识给你。

我们再次强化父修订的概念：

1. 每个修订都会指向它的前一个修订，被指向的修订就是父修订。在本案例中，r1就是r2的父修订
2. 在有分支合并的情况下，父修订可以不止一个，比如m1就有两个父修订，分别是r3、rII。
3. 如果两个修订之间隔了多层，就构成了祖先关系。比如r3修订的第2级祖先为r1，rII的第3级祖先为r1。

构建好了这个仓库、并且准备了必要的概念后，我们就可以做一系列的修订标识符相关的验证了。

基础标示符：**SHA1**标识

每个修订都有一个SHA1的长度为40的字符串标示符，可以通过git-log命

令查看。执行命令：

```
git log --pretty=oneline
```

输出：

```
ccfef91d47fd3a57f6e55d79bf12edec66673980 m2
f748b5b4020dc1993d2e4ae313da8eef818bb667 m1
14a749ac00845638e9a3f0569dbc17dc30b34ad1 r3
96f2ba93f2795b118e19f9f9bb5f05875bf9a0a8 rII
965714812a614ae84e836275e30d83effc18235e r2
b478ec85b1462e126f4926e467c79c4a12de1a73 rI
e674b5b974d7b2c344aa4a2d919748ba3cb1bb38 r1
```

此命令打印全部修订信息，每行一个修订信息，每个修订信息由基础标识符和提交消息构成。

SHA1算法

我们稍停片刻，了解下SHA1算法。它是一个非常有趣的算法，可以输入任何一个文件，并且无论文件多么小或者多么的大，都会输出一个40位长的字符串。如果你随即把输入文件内容做了改动，无论改动比例是大还是小，再次执行此算法，输出的40位字符串也会和上次并不相同。就是说，输入变化了，输出就一定变化。SHA1算法这个奇妙的特性可以用来做防篡改。比如软件作者发布软件包的时候，同时给出一个此软件包对应的SHA1字符串；下载的用户可以对此软件包应用SHA1算法，得到一个SHA1字符串并和作者提供的SHA1字符串两相比较，如果不同即可知道此软件包肯定被篡改过。

我们可以使用openssl提供的sha1算法，输入一个字符串作为源，查看它的SHA1:

```
echo -n &amp;quot;yourpassword&amp;quot; | openssl sha1
(stdin)= b48cf0140bea12734db05ebcdb012f1d265bed84
```

不管执行几次，结果是一样的：

```
echo -n &amp;quot;yourpassword&amp;quot; | openssl sha1
(stdin)= b48cf0140bea12734db05ebcdb012f1d265bed84
```

但是对输入做点修改，输出就会不同:

```
echo -n &amp;quot;yourpassword1&amp;quot; | openssl sha1
(stdin)= 9780c67b7b3ab282c91891fa49110be00890a72a
```

来一个比较大的字符串，输出一样是40位：

```
for i in {1..1000000}; do echo &amp;quot;yourpassword{$i}&amp;quot;; done |  openss
```

这样我们就可以从黑盒角度，充分地了解sha1算法的几个重要特性。

因为Git使用SHA1算法来计算修订标识符时，给算法的输入中包含了作者信息、提交者信息和提交时间、父修订标识符等。因此，即使提交消息、作者和提交者相同，在不同的时间创建的SHA1也是不同的，这是由SHA1算法和Git提供的算法输入内容可以得出的推论。因此，如果你完整照搬我提供的命令的话，你也会发现，你的git-log中输出的SHA1字符串和我的并不相同，在需要引用SHA1值的场合，你需要引入你自己的执行结果才可以。

基础标示符：继续

现在我们来使用下SHA1字符串，把它作为参数传递给git-show命令，用来显示修订信息。这里我们把最后一个修订的标识符（我的是"ccfef91d47fd3a57f6e55d79bf12edec66673980"）作为命令的参数：

```
git show ccfef91d47fd3a57f6e55d79bf12edec66673980
```

此命令可以显示此修订的信息包括作者、提交消息、和上次提交的差别等：

```
commit ccfef91d47fd3a57f6e55d79bf12edec66673980
Author: 1000copy <1000copy@gmail.com>
Date:   Sun Apr 3 17:53:06 2016 +0800

    m2

diff --git a/file1 b/file1
index 7f160f6..82b1b22 100644
--- a/file1
+++ b/file1
@@ -3,3 +3,4 @@ line2
 lineI
 lineII
 line3
+line4
```

我们确实发现 ccfef91d47fd3a57f6e55d79bf12edec66673980指向的提交的消息就是m2，这和我们的期望是一致的。

缩写的修订标识符

SHA1标识是可以缩写的，前提是缩写的位数不至于引发标识符的二义性。使用--abbrev-commit参数可以得到缩写的、但是不会冲突的修订标识符：

```
git log --abbrev-commit --pretty=oneline
```

输出：

```
ccfef91 m2
f748b5b m1
14a749a r3
96f2ba9 rII
9657148 r2
b478ec8 rI
e674b5b r1
```

使用`--abbrev-commit` 参数会输出缩写的标识符，输出依然还是修订

标示符和提交消息的列表，但是SHA1标识变短了。我们依然使用git-show命令来查看这些标示符对应的修订信息:

```
git show ccfef91
```

输出和 `git show ccfef91d47fd3a57f6e55d79bf12edec66673980` 是完全一致的。就是说，在此仓库的场景下，提交标识符使用完整的SHA1字符串和缩写后的7位得到的效果是一样的。

引用标识符：**head**

当前分支的最近一次修订总是非常常用的，因此Git使用“head”指向这个特别的修订。以本次构建的仓库为例，head就指向最后一次提交，它的标识符缩写后是ccfef91。我们可以使用head作为git-show的参数，验证这个事实:

```
git show --oneline --quiet  head
```

输出：

```
ccfef91 m2
```

这次加入了参数`--oneline`，告诉git-show命令只要打印提交标识符和提交消息的清单即可，参数`--quiet` 加入，就不会显示与我们而言多余的diff信息了。

此命令和：

```
git show --oneline --quiet ccfef91
```

输出是一样的：

```
ccfef91 m2
```

引用head在每次提交后都会是变化的，总之都会指向当前分支的最近一次修订。不过这样说不够严谨，因为还有一种叫做detached head的特殊情况，不过暂时我们不去管它。

引用标识符：分支

可以使用分支名称作为引用，指向此分支的最后一次修订。再次使用`git show`验证这个事实：

```
git show --oneline --quiet  master
```

输出:

```
ccfef91 m2
```

和如下命令：

```
git show --oneline --quiet ccfef91
```

是一样的：

```
ccfef91 m2
```

“~”方法

可以使用修订标识符+"~"+Number 的方式来访问指定修订的前Number个祖先提交。比如head~3 就是head向前数过去的第三个祖先的修订，在我们的案例仓库内，它就是r2：

```
git show --oneline --quiet  head~3
```

输出验证确实如此：

```
9657148 r2
```

“^” 方法

可以使用修订标识符+^+Number 的方式指示修订的第N个父修订。在有分支合并的修订情况下，当前修订的父修订可以有多个。Number的值会用来区别不同的父修订。

比如：修订M1的父修订就有两个。分别为rII、r3。M1可以用head~1表达，它有两个父修订，使用head~1^1表示r3,使用head~1^2 表示rII。验证如下：

```
git show --oneline --quiet  head~1^1
14a749a r3

git rita$ git show --oneline --quiet  head~1^2
96f2ba9 rII
```

head变化规律

head指向当前分支的最后一次提交，这是Git中的非常重要的一个概念。每次提交、撤销、分支切换都可能引发head指向的变化：

1. 提交时，head指向当前分支的最后一次修订。
2. 撤销时，head指向`git reset` 参数要求的修订。
3. 分支转换时，head指向`git checkout` 参数要求的分支的最后一次提交。

为了验证head的变化规律，做实验之前，我们需要知道`git rev-parse` 命令的一些使用方法：

```
查看HEAD指向的修订SHA1
    git rev-parse HEAD
查看第一个参数指定的分支所指向的修订SHA1，如
    git rev-parse master
查看当前分支名称
    git rev-parse --abbrev-ref HEAD
```

每次一组命令执行完毕，我们都会执行此一组三命令，以它们的输出来确认我们仓库和head的指向位置、分支指向的修订和当前分支名称。

接下来，我们以实验来验证我们的观点。创建仓库，第一次提交：

```
git init p1 && cd p1
echo line1 > file1
git add .
git commit -m&amp;quot;r1&amp;quot;
```

执行以下命令组合：

```
git rev-parse HEAD
git log --pretty=oneline
git rev-parse --abbrev-ref HEAD
```

输出分别为：

```
5df27b9624bdf033b4150baa3242e52ab6aee973

5df27b9624bdf033b4150baa3242e52ab6aee973 r1

master
```

说明当前head指向master分支的最新修订。

第二次提交：

```
echo line2 >> file1
git commit -m&amp;quot;r2&amp;quot; -a
```

命令组的输出为： 56285f691bcf0a4ae3a86f1bac7de4421bef30a3

```
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1
```

```
master
```

说明当前head指向master分支第二个提交，即最新修订。

转换分支：

```
git checkout -b roma
```

命令组的输出：

```
56285f691bcf0a4ae3a86f1bac7de4421bef30a3

56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

roma
```

说明当前head指向roma的分支的最新修订。

分支roma提交：

```
echo lineI > file1
git commit -m&amp;quot;rI&amp;quot; -a
```

命令组的输出：

```
70a0bf7a7fa591e631782a16a57f805ad25cd948

70a0bf7a7fa591e631782a16a57f805ad25cd948 rI
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

roma
```

说明当前head指向roma的最近修订。

再次提交：

```
echo lineII >> file1
git commit -m&amp;quot;rII&amp;quot; -a
```

命令组的输出：

```
fc028aaf7de3e4e883985a347db51c7cefdf6b71

fc028aaf7de3e4e883985a347db51c7cefdf6b71 rII
70a0bf7a7fa591e631782a16a57f805ad25cd948 rI
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

roma
```

说明当前head指向roma的最近修订。

切换分支回到master：

```
git checkout  master

echo line3 >> file1
git commit -m&amp;quot;r3&amp;quot; -a
```

命令组的输出： eb27a92e300dafccd66b1b862b1321d38a095d5e

```
eb27a92e300dafccd66b1b862b1321d38a095d5e r3
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

master
```

合并：

```
git merge roma
// merge
git commit -m&amp;quot;m1&amp;quot; -a
```

命令组的输出：

```
fb815c7ffbf5141efb7cc06d72a3502dc2d1e0b1

fb815c7ffbf5141efb7cc06d72a3502dc2d1e0b1 m1
eb27a92e300dafccd66b1b862b1321d38a095d5e r3
fc028aaf7de3e4e883985a347db51c7cefdf6b71 rII
70a0bf7a7fa591e631782a16a57f805ad25cd948 rI
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

master
```

说明当前head指向 master 的分支的最近修订。

```
echo line4 >> file1
git commit -m&amp;quot;m2&amp;quot; -a
```

命令组的输出：

```
5bf8d66959f324980389c3cf124658b09615de7d

5bf8d66959f324980389c3cf124658b09615de7d m2
fb815c7ffbf5141efb7cc06d72a3502dc2d1e0b1 m1
eb27a92e300dafccd66b1b862b1321d38a095d5e r3
fc028aaf7de3e4e883985a347db51c7cefdf6b71 rII
70a0bf7a7fa591e631782a16a57f805ad25cd948 rI
56285f691bcf0a4ae3a86f1bac7de4421bef30a3 r2
5df27b9624bdf033b4150baa3242e52ab6aee973 r1

master
```

说明当前head指向master合并后的分支的最近修订。总之，head总是指向当前分支的最近修订的。

标签

Git标签用于版本标记。比如发出版本1.0后就可以打个标签，这样以后如果需要1.0当时的代码，可以checkout它出来，修改此版本的bug就比较容易了。

现在我来打一个新标签：

```
$ git tag v1.0
```

可以执行

```
$ git tag
```

输出：

```
v1.0
```

假设我们继续修改文件，并且提交：

```
echo line3 >> file1
git add file1
git commit -m&quot;commit 3&quot;
```

当我正在修改得不亦乐乎时，却有客户反馈说使用的1.0有错，于是我获得1.0的代码来修改以便解决bug。像是这样：

```
git checkout tags/v1.0
git checkout -b bugfix
sed -i.bak 's/line1/lineI' file1
git add file1
git commit -m&quot;bug fix&quot;
```

对v1.0的bug修改，现在就在分支bugfix内了。测试通过后，我就可以把此分支的修改成果合并到主干上：

```
git merge bugfix
```

现在对bugfix的修改也同时反映到master上：

```
$ cat file1
lineI
lineII
line3
```

多仓库

Git通过多个仓库之间拉取和推送变化来达成多用户协作。我们就来试试两人协作。场景假设如下：

1. 用户A、B，分别工作于仓库repo1、repo2。
2. 创建一个共享仓库repo。
3. 用户A修改提交到repo1后，可以推送变化到共享仓库repo。
4. 用户B从共享仓库repo拉取更新到repo2。

仓库之间传输数据需要使用协议。Git支持4种协议，包括：本地协议、HTTP协议、Git协议、SSH协议。我们首先使用本地协议，并且在一台电脑上模拟两个用户的协作过程。

实验

首先建立并进入一个新目录（我实验用的目录为~/git）。我们实验要用到的3个仓库都会放到此目录内。

随后我们创建仓库repo1：

```
mkdir repo1
cd repo1
git init
```

接着创建用户A。它是一个局部用户（不要使用--global参数即可），仅仅用于当前仓库：

```
git config  user.name &amp;quot;a&amp;quot;
git config  user.email &amp;quot;a@whatever.com&amp;quot;
```

以用户A的身份，提交r1：

```
echo line1 > file1
```

```
git add .
git commit -m&quot;r1&quot;
```

以克隆的方式来创建一个共享仓库：

```
cd ..
git clone --bare repo1 repo.git
```

这里的参数--bare指定克隆后的仓库仅仅用于共享。共享仓库和一般仓库的差别，在于前者的仓库中没有工作目录。我们也遵循惯例，在共享仓库的目录名内加上.git后缀以示区别。

建立repo2仓库。方法就是克隆共享仓库repo：

```
git clone repo.git repo2
```

现在，我们有三个仓库了。其中一个是共享仓库：

```
ls

repo.git    repo1       repo2
```

然后，我们为repo2建立配置用户B：

```
cd repo2
git config  user.name &quot;b&quot;
git config  user.email &quot;b@whatever.com&quot;
```

我们在目录repo2内，使用命令查看file1文件内容：

```
cat file1
```

输出： line1

说明repo1的内容已经被克隆到repo2内。显示仓库内的版本历史：

```
git hist
* ff283ef 2016-04-13 | r1 (HEAD -> master, origin/master, origin/HEAD) [a]
```

说明历史内的作者是妥当的：尽管现在在用户B的仓库内，但是作者是正确地显示为用户A的。

我们还可以查看远程仓库情况：

```
git remote -v
origin  /Users/lcjun/git/repo (fetch)
origin  /Users/lcjun/git/repo (push)
```

输出说明，有一个名字为origin，位置在 /Users/lcjun/git/repo 的远程仓库，它可以用来拉取，也可以用来推送。

克隆出来的仓库，可以看到的只有一个分支。想要看到所有的分支可以加上-a参数:

```
  git branch -a
  * master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

其中以remotes开头的分支都是远程分支，指向远程仓库对应的分支。

目前为止，用户A已经提交了自己的更新r1，并推送到共享仓库repo.git。用户B已经拉取了共享仓库repo，并建立了自己的工作目录。现在我们来个反向操作，用户B提交更新，由用户A拉取修改。

仓库别名

在仓库repo2内，用户B做些修改并推送到共享仓库：

```
echo line2 >> file1
git commit -m&amp;quot;r2&amp;quot; -a
```

```
git push
```

用户A进入自己的仓库，然后拉取变化：

```
cd ~/git/repo1
git pull /Users/lcjun/git/repo.git  master
```

命令git-pull的第一个参数是仓库 。因为我们使用的协议是本地协议，所以仓库的表示方式就是一个目录，它指向了共享仓库repo.git的本地位置(就是目录位置，/Users/lcjun/git/repo.git)。第二个参数是此仓库内要拉取的分支名称（这里是master）。于是git-pull看到仓库位置的格式是一个目录名，就可以由此知道使用本地协议来拉取指定仓库的指定分支。

在使用本地协议的时候，用目录名称来指定远程仓库位置是行得通的，但是有些不方便，因为目录名常常太长且未必是简单明了的。因此Git允许通过命令git-remote给仓库一个别名，像是这样：

```
 git remote add origin /Users/lcjun/git/repo.git
```

随后你就可以使用origin来指代/Users/lcjun/git/repo.git了：

```
git pull origin master
```

这里的参数origin并无特别之处，你可以随便给它一个名字，只要好记即可。如果你发现不喜欢origin，可以删除它和对应目录的关联，然后重新派发一个更帅的名字，比如upstream：

```
git remote remove origin
git remote add upstream /Users/lcjun/git/repo.git
```

当然，实际上我会继续使用origin，因为大家都这么用，是一个很好的命名惯例。

然后我们可以查看下拉取后的文件变化：

```
cat file1
```

输出：

```
line1
line2
```

这文件的变化说明，用户B的修改已经传递给用户A了。然后我们可以查看历史，了解文件的修订情况：

```
git hist
```

可以看到历史中第一个修订的作者用户A，第二个修订的作者则是用户B：

```
* eaefe76 2016-04-13 | r2 (HEAD -> master, origin/master) [b]
* c554f06 2016-04-13 | r1 [a]
```

我们以一台电脑上模拟两个用户的协作，其中涉及的推送和拉取的概念，也适用于其他协议的仓库协作。

协议

在上一节，我们已经通过一个共享仓库作为中介，以本地协议的通讯方式，达成两人的代码协作了。

Git除了本地协议外，还支持3种协议，包括HTTP协议、Git协议、SSH协议。我们继续了解Git协议的使用方法。

Git协议

使用git-daemon命令就可以完成仓库的托管，因此是非常方便的。此命令会启动一个监守程序，等待采用Git协议来连接的客户端；一旦连接成功，Git客户端就可以访问被托管了的、且对外公开的仓库。

现在实验验证。首先，我们创建一个干净的新目录（我的是~/git），然后在其内创建一个叫做repo.git的共享仓库：

```
git init repo.git --bare
```

此仓库为空，因为我们只是为了验证git协议的有效性，因此对仓库是否有内容这一点并不在意。随后，执行如下命令就可以完成对此目录下仓库的托管：

```
cd ~/git
git daemon --verbose --export-all --base-path=. --enable=receive-pack
```

这里的命令git-daemon需要做些解释。参数说明：

```
--base-path=. 指定基础目录，客户端连接指定的目录都相对于此目录来定位目录。当前命令
--enable=receive-pack 允许匿名推送
--export-all 公开基础目录内的全部仓库
```

启动了git监守程序后，客户端运行git-clone即可拉取仓库：

```
git clone git://localhost/repo.git
```

我们使用的主机地址为localhost，就是意味着拉取的仓库位于本机；你可以使用IP地址来访问其他主机的、采用Git协议的托管仓库了。现在你已经有了两个仓库，分别是

```
~/git/repo.git
~/git/repo
```

进入repo，可以查询它的远程仓库：

```
git remote -v
```

输出：

```
origin    git://localhost/repo.git (fetch)
origin    git://localhost/repo.git (push)
```

你可以使用一般的对仓库的修改、暂存、提交的命令，也可以克隆新的仓库，并且在仓库之间推送和拉取变化了。不同的是，这一次你使用的是git协议。

SSH 协议

在Linux和OS X上使用SSH协议非常方便，因为操作系统已经内置SSH支持。我会以OS X为例演示效果。同样是为了演示和测试的方便，远程主机使用localhost来模拟，主机内的ssh账号为lcjun。

首先激活sshd监守程序：

```
sudo systemsetup -setremotelogin on
```

远程登录（登录到localhost，模拟它为远程主机）：

```
ssh localhost
```

在ssh内创建一个共享仓库：

```
git init repo.git --bare
```

退出ssh：

```
exit
```

本机内，访问远程ssh主机（也是本机）做克隆：

```
ssh://localhost/~/repo.git
```

使用git-remote来查询远程仓库：

```
$git remote -v
```

输出： origin ssh://localhost/Users/lcjun/repo.git (fetch) origin ssh://localhost/Users/lcjun/repo.git (push)

现在，ssh远程仓库也已经和本地关联起来了。

HTTP

HTTP协议下的Git托管是相对最复杂的，因为涉及到了：

1. Web Server的配置方法
2. CGI的概念
3. 特定的OS以及版本的配置方法差异、以及目录组织的差异

本节内容假设你已经具备以上知识，并且仅仅给出特定操作系统、特定Web服务器下的配置。我的具体实验环境如下：

1. 操作系统为OS X EI Captain
2. Git版本为2.6.3
3. Web服务器采用的是Apache 2.4
4. 本机IP地址为192.168.3.11

工作目标

我会以本机同时扮演服务器和客户端的角色，在这样的环境下，我希望可以使用:

```
git clone http://git.example.com/repo.git
```

的命令，以http协议方法访问服务器上的repo.git仓库。留意URL的协议部分是http。现在让我们开始正式的配置。

设置友好的域名解析

在本机内找到hosts文件，并加入如下一行，以便支持通过主机名（git.example.com)访问本主机

```
192.168.3.11   git.example.com
```

用过ping来验证：

```
ping git.example.com
```

如果ping通了，就说明配置已经生效。

启动并验证**Apache Web**服务器可用

首先启动apache:

```
$ sudo apachectl start
```

并且通过curl 命令验证是可以访问的：

```
$ curl git.example.com
```

输出应该是：

```
<html><body><h1>It works!</h1></body></html>
```

连接**Git CGI**模块

接下来的非常关键了。Git提供了一个叫做git-http-backend 的CGI模块，它负责解析Git客户端发来的HTTP请求，并且给出恰当的、Git客户断可以识别的HTTP响应。必须要通过对Apache的配置，让Apache服务器可以把来自Git客户端的HTTP请求传递给此CGI模块。信息流向是这样的：

```
git 客户端   <--> apache server <--> git-http-backend <--> git http协议托管仓
```

首先找到http.conf，此文件是apache的配置文件，我们对apache的配置，常常就是通过修改此文件来完成的。在我的主机上，http.conf的位置在/etc/apache2/httpd.conf 。在不同的系统上配置文件位置可能不同，可以通过如下命令找到它：

```
$ find / -name httpd.conf  2>/dev/null
```

加载依赖模块

因为git-http-backend 依赖于这三个模块：cgi,env,alias，所以，要让git-http-backend 起作用，必须配置Apache加载它们。方法是修改Apache的配置文件http.conf ,确保形如：

```
LoadModule cgi_module libexec/apache2/mod_cgi.so
LoadModule env_module libexec/apache2/mod_env.so
LoadModule alias_module libexec/apache2/mod_alias.so
```

行是没有被标注的。如果被标注（行首有一个#），那么通过去掉“#”来解除标注。

配置虚拟主机和**CGI**

随后在此配置文件的文件末尾，添加如下内容：

```
<VirtualHost 192.168.3.11:80>
    DocumentRoot /var/www/git
    ServerName git.example.com
    <Directory &quot;/var/www/git&quot;>
        Options +Indexes
        Require all granted
    </Directory>
    <Directory &quot;/usr/local/Cellar/git/2.6.3/libexec/git-core/&quot;>
       Options ExecCGI Indexes
       Order allow,deny
       Allow from all
       Require all granted
    </Directory>
    SetEnv GIT_HTTP_EXPORT_ALL
    SetEnv GIT_PROJECT_ROOT /var/www/git
    ScriptAlias /git/ /usr/local/Cellar/git/2.6.3/libexec/git-core/git-http
</VirtualHost>
```

我们来看看此配置做了些什么：

1. 添加了一个虚拟主机

   它通过标签`VirtualHost` 添加了一个名为`git.example.com` 的虚拟主机，地址为本地IP，端口为80，且其文档根目录为`/var/www/git` 。

2. 加载CGI

   这段配置内通过指令`ScriptAlias` 添加了一个脚本别名，别名指向git-http-backend的CGI程序。git-http-backend 在不同的系统上可能位置不同，可以使用这个命令查找到它：

$ find / -name git-http-backend 2>/dev/null

3. 指定了执行CGI的权限（通过标签`Directory`）

   为了让此CGI 可以运行，也必须配套地把CGI所在目录设置为允许执行CGI（Options ExecCGI）。

4. 通过设置环境变量，设定Git托管的基础目录以及托管规则(通过指令SetEnv)

   环境变量GIT_PROJECT_ROOT是必须设置的，`git-http-backend` 使用此环境变量来定位托管仓库的基础目录了。环境变量GIT_HTTP_EXPORT_ALL设置就意味着在基础目录内的所有仓库都是可以对外共享的。

创建共享仓库和设置访问权限

首先找到Apache的Web文档根目录。我的对应目录为/var/www。如果你的并不相同，那么请在随后的目录使用中替换为你的。我会在Web文档根目录内建立一个子目录，以此目录为容器，加入我们的Git仓库。具体目录为/var/www/git。现在，我们在此目录内创建一个名为repo.git的仓库：

```
mkdir -p /var/www/git/repo.git
cd /var/www/git/repo.git
git init --bare
```

然后记得允许Apache进程可以访问此基础目录。我的Apache用户和用户组都是_www。

```
$ chown -R _www:_www /var/www/git/
```

你的系统可能不同与我，但是可以使用命令查找出来。找到apache 用户的方法：

```
$ ps aux | egrep '(apache|httpd)'.
```

把上面命令得到的Apache用户名替代到id命令的参数内，就可以找到apache用户组名称：

```
$ id user-name
```

进入需要克隆的git仓库所在的目录并执行`git update-server-info`：

```
   cd /var/www/git/repo.git
   sudo git update-server-info
```

重启apache：

```
sudo apachectl restart
```

验证我们的配置成功

随后，我就可以真地去克隆一个在apache服务后的git仓库了：

```
git clone http://git.example.com/repo.git
```

输出:

```
Cloning into 'repo'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

查错方法

这个配置过程相对复杂，难免出现不期望的错误，如果你按照我的配置方法，却无法出现期望的结果，可以使用一些命令来定位问题，帮助调试。

比如查看Apache运行错误：

```
sudo tail /private/var/log/apache2/error_log
```

查看Apache访问日志，就可以知道Git的http协议访问样子：

```
sudo tail /private/var/log/apache2/access_log

192.168.3.11 - - [11/May/2016:17:34:40 +0800] &amp;quot;GET /repo.git/info/refs
...
```

我需要多次修改http.conf ，所以我常常会使用：

```
 apachectl configtest
```

验证下配置是正确的。

在我的主机上，我在升级git的时候，曾经出过这样的报错：

```
$brew upgrade git
# Git package broken in 10.11 El Capitan
brew reinstall git
...
==> make prefix=/usr/local/Cellar/git/2.4.3 sysconfdir=/usr/local/etc CC=cl
./git-compat-util.h:219:10: fatal error: 'openssl/ssl.h' file not found
#include <openssl/ssl.h>
        ^
```

我尝试了改变下命令行参数的方式：

```
brew reinstall git --with-brewed-openssl
```

结果是可以升级到最新的git版本的（git 2.6.4）。

后记： 信任网络

Linus 是Git的创造者，他创造Git首先是为了解决Linux的开发过程管理。所以想要懂得Git的优秀，得弄明白Linus创造Git时的工作场景。

1. Linux开发者是分布的。1000多人的Linux开发团队是分布在世界各地的。使用Git也就不必依赖中心服务器、不必需要很好的网络，在自己的电脑上就有完整的仓库可以做大部分日常的版本管理。那些集中式的工具如SVN显然是不合适的，因为单点故障大家就会无法提交，当然也会无法开分支，对于那些把分支融入日常工作流中的人来说，这是特别无法忍受的。

2. 剔除害群之马很简单。Linus想要封杀特别容易出漏子的程序员，只要不拉取他的代码即可，实际上Linus就这样干过。如果是SVN，要封杀一个提交者，就需要撤销此人的账号或者限定他的访问范围，并且从仓库中移除麻烦的代码提交。就是说，封杀的方法在Git而言，是不做某事即可，这样的工作流程就避开了很多“政治”问题。

3. 可以使用信任网络。Linus在早期接受Linux补丁时，确实是会仔细完整地看完代码，评估质量再合并。然后如今的Linux太大了，贡献者也非常多，Linus一个人看不完这些补丁，他只能基于信任而拉取少数人的代码补丁。而这些少数被信任的人，也会发现有些程序员特别优秀，他认可后，也只要选择拉取他们的实现。于是，每个Linux的贡献者都只是拉取他们信任的程序员的实现。这样的信任网络是可以层次化的，对应于1000多人的开发者来说，这样做确实可以通过分层的信任网络达成大规模的团队协作。

所以，对于Linux团队来说，Git是必须的。它的设计，首先是关于人、关于最佳工作流程的。Linus不仅仅创建了Linux，也管理着一个巨大的团队，做法就是用一套工具提供信任网络，避开“政治”问题的工作流程。说Linus是此团队的“仁君”这一点其实并非缪赞。想想混乱的现实世界，我确实对这个高手的团队管理方法感到敬佩。

这些内容，是我看了Linus在Google的演讲后写的，其中部分内容就是

对Linus谈话中涉及到设计思想的一个在文字在先而已。

最后来说说我自己。我在第一本电子书《HTTP小书》 中提到，我写书是为了暂时离开我反感的“协作”，但是也提到了我写书的理念，在这里再次重复下：

“说起来都是写书，我会希望有何不同呢？这真是一个好问题。我的回答是，我会努力提升一本书的信息密度。我秉持的原则是用更少的文字和代码来表达更多的信息量。我会大胆砍掉我花了很多时间写的、但是看起来不够好的段落甚至章节，留下我认为的最佳的内容，表现出来的就是我把这本书写得更薄，而不是相反。在信息爆炸的年代，信息的精简变成新的奢侈”。

我会有意识地精简概念，使用简洁的命令和输出，避开那些常常毫无意思、只是增加识别阅读负担的图形，从而提供给你一个更好的阅读体验。

刘传君 2016年05月31日