

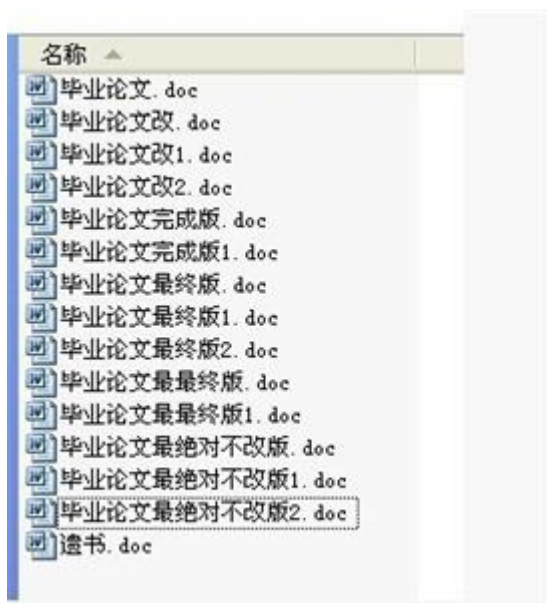
Git 与 Repo 入门

版本控制

版本控制是什么已不用在说了，就是记录我们对文件、目录或工程等的修改历史，方便查看更改历史，备份以便恢复以前的版本，多人协作。。。

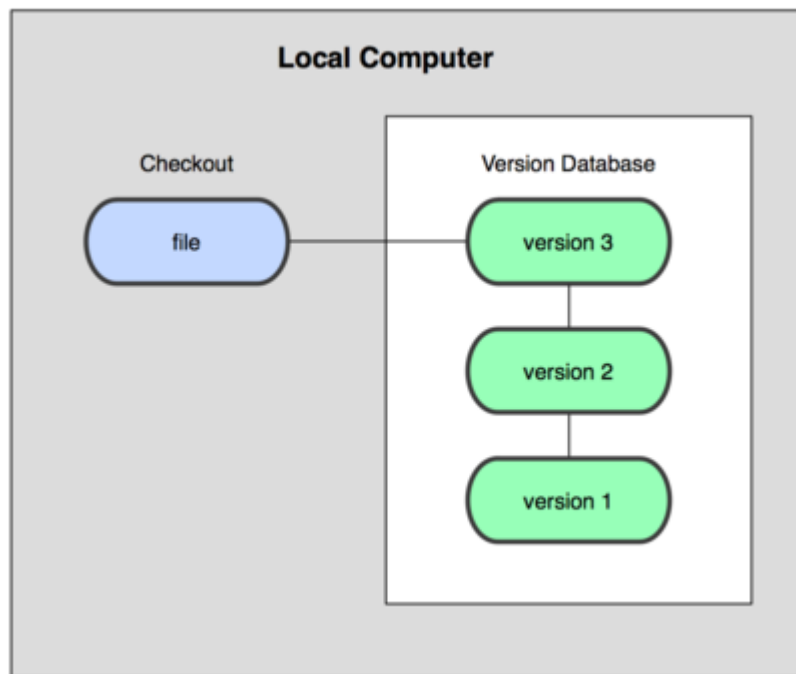
一、原始版本控制

最原始的版本控制是纯手工的版本控制：修改文件，保存文件副本。有时候偷懒省事，保存副本时命名比较随意，时间长了就不知道哪个是新的，哪个是老的了，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版作了什么修改了，当几个版本过去后，很可能就是下面的样子了：



二、本地版本控制

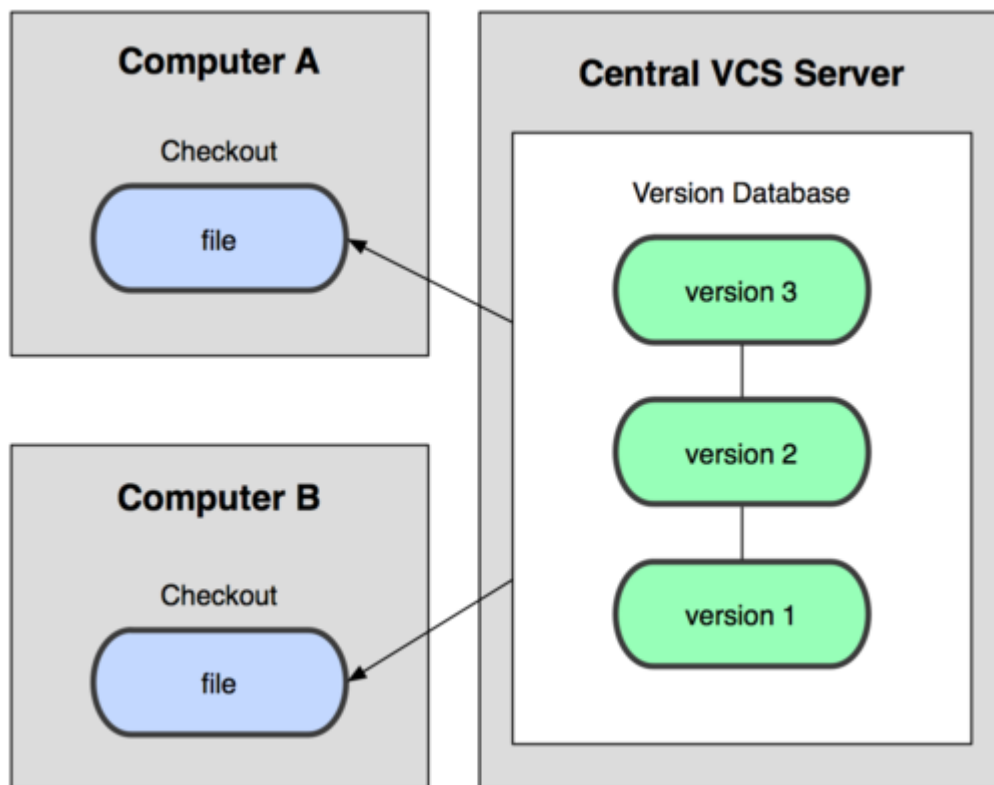
手工管理比较麻烦且混乱，所以出现了本地版本控制系统，记录文件每次的更新，可以对每个版本做一个快照，或是记录补丁文件。比如 RCS。



三、集中版本控制

但是本地版本控制系统偏向于个人使用，或者多个使用的人必须要使用相同的设备，如果需要多人协作就不好办了，于是，集中化的版本控制系统（ Centralized Version Control Systems，简称 CVCS ）应运而生，比如 Subversion，Perforce。

在 CVCS 中，所有的版本数据都保存在服务器上，一起工作的人从服务器上同步更新或上传自己的修改。



但是，所有的版本数据都存在服务器上，用户的本地设备就只有自己以前所同步的版本，如果不连网的话，用户就看不到历史版本，也无法切换版本验证问题，或在不同分支工作。。

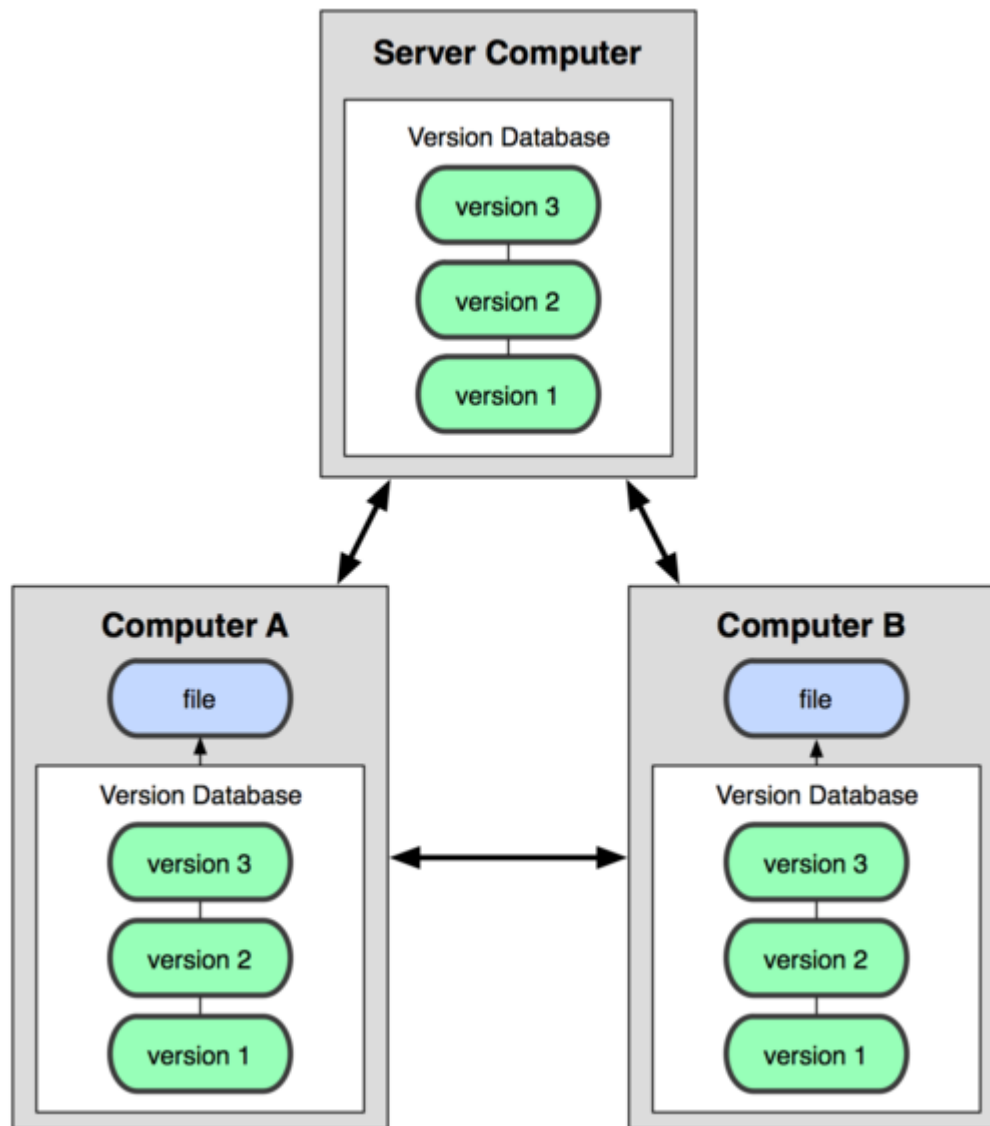
而且，所有数据都保存在单一的服务器上，有很大的风险这个服务器会损坏，这样就会丢失所有的数据，当然可以定期备份。

四、分布式版本控制

针对 CVCS 的以上缺点 ,出现了分布式版本控制系统(Distributed Version Control System,简称 DVCS) 如 GIT ,Mercurial。

DVCS 不是复制指定版本的快照 ,而是把所有的版本信息仓库全部同步到本地 ,这样就可以在本地查看所有版本历史 ,可以离线在本地提交 ,只需在连网时 push 到相应的服务器或其他用户那里。由于每个用户那里保存的都是所有的版本数据 ,所以 ,只要有一个用户的设备没有问题就可以恢复所有的数据。

当然 ,这增加了本地存储空间的占用。



GIT

必须要了解 GIT 的原理，才能知道每个操作的意义是什么，才能更容易地理解在什么情况下用什么操作，而不是死记命令。

当然，第一步是要获得一个 GIT 仓库。

一、获得 GIT 仓库

有两种获得 GIT 仓库的方法，一是在需要用 GIT 管理的项目的根目录执行：

```
git init
```

执行后可以看到，仅仅在项目目录多出了一个.git 目录，关于版本等的所有信息都在这个目录里面。

另一种方式是克隆远程目录，由于是将远程服务器上的仓库完全镜像一份至本地，而不是取某一个特定版本，所以用 clone 而不是 checkout：

```
git clone <url>
```

二、GIT 中版本的保存

记录版本信息的方式主要有两种：

1. 记录文件每个版本的快照
2. 记录文件每个版本之间的差异

GIT 采用第一种方式。像 Subversion 和 Perforce 等版本控制系统都是记录文件每个版本之间的差异，这就需要对比文件两个版本之间的具体差异，但是 GIT 不关心文件两个版本之间的具体差别，而是关心文件的整体是否有改变，若文件被改变，在添加提交时就生成文件新版本的快照，而判断文件整体是否改变的方法就是用 SHA-1 算法计算文件的校验和。

GIT 能正常工作完全信赖于这种 SHA-1 校验和，当一个文件的某一个版本被记录之后会生成这个版本的一个快照，但是一样要能引用到这个快照，GIT 中对快照的引用，对每个版本的记录标识全是通过 SHA-1 校验和来实现的。

当一个文件被改变时，它的校验和一定会被改变（理论上存在两个文件校验和相同，但机率小到可以忽略不计），GIT 就以此判断文件是否被修改，及以些记录不同版本。

在工作目录的文件可以处于不同的状态，比如说新添加了一个文件，GIT 发觉了这个文件，但这个文件是否要纳入 GIT 的版本控制还是要由我们自己决定，比如编译生成的中间文件，我们肯定不想纳入版本控制。下面就来看下文件状态。

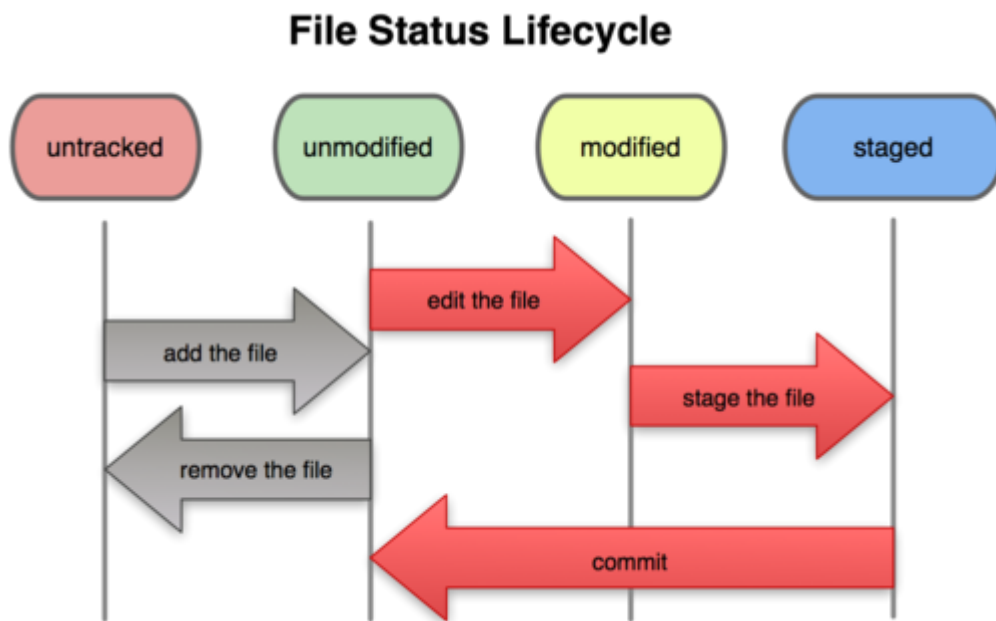
三、GIT 文件操作

版本控制就是对文件的版本控制，对于 Linux 来说，设备，目录等全是文件，要对文件进行修改、提交等操作，首先要知道文件当前在什么状态，不然可能会提交了现在还不想提交的文件，或者要提交的文件没提交上。

文件状态

GIT 仓库所在的目录称为工作目录，这个很好理解，我们的工程就在这里，工作时也是在这里做修改。

在工作目录中的文件被分为两种状态，一种是已跟踪状态(tracked)，另一种是未跟踪状态(untracked)。只有处于已跟踪状态的文件才被纳入 GIT 的版本控制。如下图：

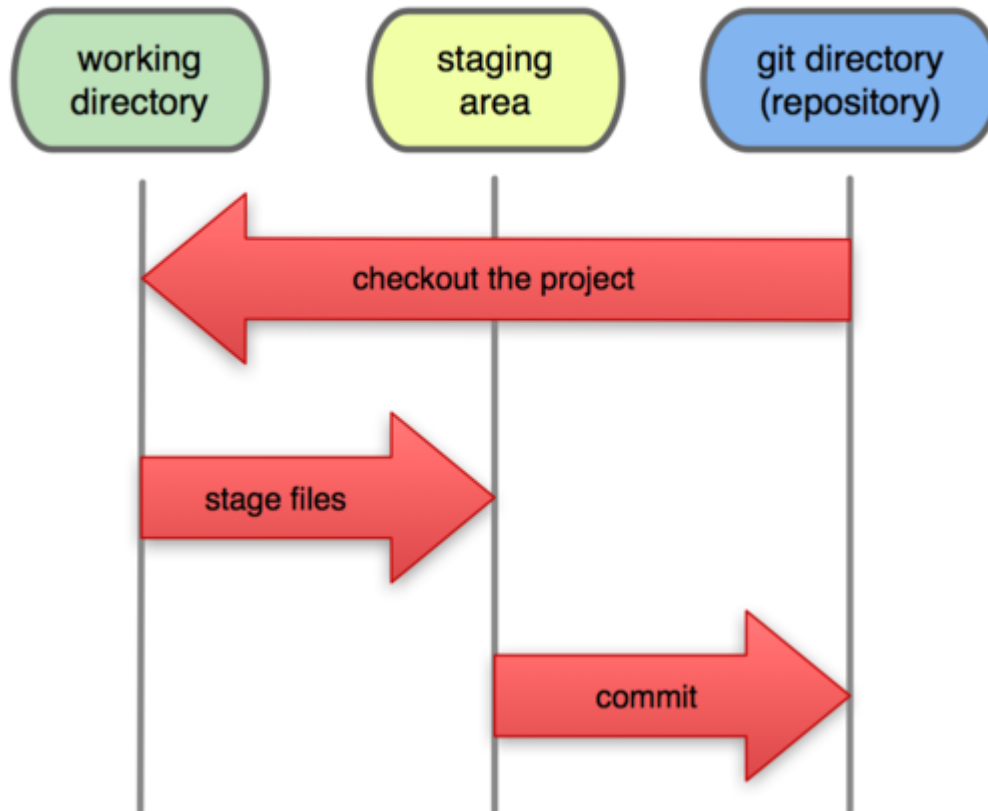


当我们往工作目录添加一个文件的时候，这个文件默认是未跟踪状态的，我们肯定不希望编译生成的一大堆临时文件默认被跟踪还要我们每次手动将这些文件清除出去。用以下命令可以跟踪文件：

```
git add <file>
```

上图中右边 3 个状态都是已跟踪状态，其中的灰色箭头只表示 untracked<-->tracked 的转换而不是 untracked<-->unmodified 的转换，新添加的文件肯定算是被修改过的。那么，staged 状态又是什么呢？这就要搞清楚 GIT 的三个工作区域：本地数据（仓库）目录，工作目录，暂存区，如下图所示：

Local Operations



git directory 就是我们的本地仓库.git 目录，里面保存了所有的版本信息等内容。

working directory ,工作目录 ,就是我们的工作目录 ,其中包括未跟踪文件及已跟踪文件 ,而已跟踪文件都是从 git directory 取出来的文件的某一个版本或新跟踪的文件。

staging area , 暂存区 , 不对应一个具体目录 , 其时只是 git directory 中的一个特殊文件。

当我们修改了一些文件后 , 要将其放入暂存区然后才能提交 , 每次提交时其实都是提交暂存区的文件到 git 仓库 , 然后清除暂存区。而 checkout 某一版本时 , 这一版本的文件就从 git 仓库取出来放到了我们的工作目录。

文件状态的查看

那么 , 我们怎么知道当前工作目录的状态呢 ? 哪些文件已被暂存 ? 有哪些未跟踪的文件 ? 哪些文件被修改了 ? 所有这些只需要一个命令 , git status , 如下图所示:

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   fileb
#
# 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:   fileb
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       filea
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

GIT 在这一点做得很好，在输出每个文件状态的同时还说明了怎么操作，像上图就有怎么暂存、怎么跟踪文件、怎么取消暂存的说明。

文件暂存

在上图中我们可以很清楚地看到，filea 未跟踪，fileb 已被暂存（changes to be committed），但是怎么还有一个 fileb 是 modified 但 unstaged 呢？这是因为当我们暂存一从此文件时，暂存的是那一文件当时的版本，当暂存后再次修改了这个文件后就会提示这个文件暂存后的修改是未被暂存的。

接下来我们就看怎么暂存文件，其实也很简单，从上图中可以看到 GIT 已经提示我们了：use "git add <file>..." to update what will be committed，通过

```
git add <file>...
```

就可以暂存文件，跟踪文件同样是这一个命令。在这个命令中可以使用 glob 模式匹配，比如"file[ab]"，也可以使用"git add .". 添加当前目录下的所有文件。

取消暂存文件是

```
git reset HEAD <file>...
```

若修改了一个文件想还原修改可用

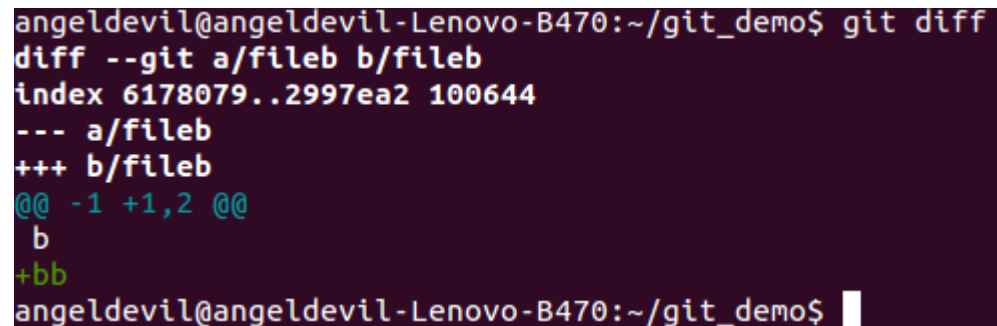
```
git checkout -- <file>...
```

查看文件修改后的差异

当我们修改过一些文件之后，我们可能想查看我们都修改了什么东西，用"git status"只能查看对哪些文件做了改动，如果要看改动了什么，可以用：

```
git diff
```

比如下图：

A terminal window screenshot showing the output of the 'git diff' command. The prompt is 'angeldevil@angeldevil-Lenovo-B470:~/git_demo\$'. The command entered is 'git diff'. The output shows a diff for 'a/fileb' and 'b/fileb' with a line change from 'a' to 'b'.

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git diff
diff --git a/fileb b/fileb
index 6178079..2997ea2 100644
--- a/fileb
+++ b/fileb
@@ -1,2 @@
  a
+bb
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```


---a 表示修改之前的文件，+++b 表示修改后的文件，上图表示在 fileb 的第一行后添加了一行"bb"，原来文件的第一行扩展为了修改后的 1、2 行。

但是，前面我们明明用"git status"看到 filesb 做了一些修改后暂存了，然后又修改了 fileb，理应有两次修改的，怎么只有一个？

因为"git diff"显示的是文件修改后还没有暂存起来的内容，那如果要比较暂存区的文件与之前已经提交过的文件呢，毕竟实际提交的是暂存区的内容，可以用以下命令：

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git diff --cached
diff --git a/fileb b/fileb
new file mode 100644
index 0000000..6178079
--- /dev/null
+++ b/fileb
@@ -0,0 +1 @@
+b
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

/dev/null 表示之前没有提交过这一个文件，这是将是第一次提交，用：

```
git diff --staged
```

是等效的，但 GIT 的版本要大于 1.6.1。

再次执行"git add"将覆盖暂存区的内容。

忽略一些文件

如果有一些部件我们不想纳入版本控制，也不想每次"git status"时看到这些文件的提示，或者很多时候我们为了方便会使用"git add ."添加所有修改的文件，这时就会添加上一些我们不想添加的文件，怎么忽略这些文件呢？

GIT 当然提供了方法，只需在主目录下建立".gitignore"文件，此文件有如下规则：

- 所有以#开头的行会被忽略
- 可以使用 glob 模式匹配
- 匹配模式后跟反斜杠 (/) 表示要忽略的是目录
- 如果不要忽略某模式的文件在模式前加"!"

比如：

```
# 此为注释 - 将被 Git 忽略  
*.a # 忽略所有 .a 结尾的文件  
!lib.a # 但 lib.a 除外  
/TODO # 仅仅忽略项目根目录下的 TODO 文件,不包括 subdir/TODO  
build/ # 忽略 build/ 目录下的所有文件  
doc/*.txt # 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
```

移除文件

当我们要删除一个文件时，我们可能就直接用 GUI 删除或者直接 `rm [file]`了，但是看图：

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ rm filea
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    filea
#
no changes added to commit (use "git add" and/or "git commit -a")
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

我们需要将文件添加到暂存区才能提交，而移除文件后是无法添加到暂存区的，那么怎么移除一个文件让 GIT 不再将其纳入版本控制呢？上图中 GIT 已经给出了说明：

```
git rm <file>...
```

执行以上命令后提交就可以了，有时我们只是想将一些文件从版本控制中剔除出去，但仍保留这些文件在工作目录中，比如我们一不小心将编译生成的中间文件纳入了版本控制，想将其从版本控制中剔除出去但在工作目录中保留这些文件（不然再次编译可要花费更多时间了），这时只需要添加"--cached"参数。

如果我们之前不是通过"git rm"删除了很多文件呢？比如说通过 patch 或者通过 GUI，如果这些文件命名没有规则，一个一个地执行"git rm"会搞死人的，这时可以用以下命令：

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git rm $(git ls-files --deleted)
rm 'filea'
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    filea
#
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

移动文件

和移除文件一样，移动文件不可以通过 GUI 直接重命名或用"mv"命令，而是要用"git mv"，不然同移除文件一样你会得到如下结果：

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ mv filea filed
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git st
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    filea
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       filed
no changes added to commit (use "git add" and/or "git commit -a")
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

如果要重命名文件可以使用

```
git mv old_name new_name
```

这个命令等效于

```
mv old_name new_name
```

```
git rm old_name
```

```
git add new_name
```

交互式暂存

使用 `git add -i` 可以开启交互式暂存，如图所示，系统会列出一个功能菜单让选择将要执行的操作。

```
angeldevil@angeldevil:~/git_demo$ git add -i
          staged      unstaged path
  1:    unchanged      +1/-0 a

*** Commands ***
  1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked
  5: [p]atch      6: [d]iff       7: [q]uit       8: [h]elp
What now> █
```

移除所有未跟踪文件

```
git clean [options]
```

 一般会加上参数 `-df`，`-d` 表示包含目录，`-f` 表示强制清除。

储藏-Stashing

可能会遇到这样的情况，你正在一个分支上进行一个特性的开发，或者一个 Bug 的修正，但是这时突然有其他的事情急需处理，这时该怎么办？不可能就在这个工作进行到一半的分支上一起处理，先把修改的 Copy 出去？太麻烦了。这种情况下就要用到 Stashing 了。假如我们现在的工作目录是这样子的

```
$ git status
# On branch master
# Changes to be committed:
#
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:
#   index.html
#
#   Changed but not updated:
#
#   (use "git add <file>..." to update what will be committed)
#
#   modified:
#   lib/simplegit.rb
```


此时如果想切换分支就可以执行以下命令

```
$ 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")
```

这时你会发现你的工作目录变得很干净了，就可以随意切分支进行其他事情的处理了。

我们可能不只一次进行"git stash"，通过以下命令可以查看所有 stash 列表

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
```

当紧急事情处理完了，需要重新回来这里进行原来的工作时，只需把 Stash 区域的内容取出来应用到当前工作目录就行，命令就是

```
git stash apply
```

如果不基参数就应用最新的 stash，或者可以指定 stash 的名字，如：stash@{1}，可能通过

```
git stash show
```

显示 stash 的内容具体是什么，同 git stash apply 一样，可以选择指定 stash 的名字。

git stash apply 之后再 git stash list 会发现，apply 后的 stash 还在 stash 列表中，如果要将其从 stash 列表中删除可以用

```
git stash drop
```

丢弃这个 stash，stash 的命令参数都可选择指定 stash 名字，否则就是最新的 stash。

一般情况下 apply stash 后应该就可以把它从 stash 列表删除了，先 apply 再 drop 还是比较繁琐的，使用以下一条命令就可以同时完成这两个操作

```
git stash pop
```

如果我们执行 git stash 时工作目录的状态是部分文件已经加入了暂存区，部分文件没有，当我们执行 git stash apply 之后会发现所有文件都变成了未暂存的，如果想维持原来的样子操持原来暂存的文件仍然是暂存状态，可以加上--index 参数

```
git stash apply --index
```

还有这么一种情况，我们把原来的修改 stash 了，然后修复了其他一些东西并进行了提交，但是，这些提交的文件有些在之前已经被 stash 了，那么 git stash apply 时就很可能会遇到冲突，这种情况下就可以在 stash 时所以提交的基础上新建一个分支，然后再 apply stash，当然，这两个步骤有一人简单的完成方法

```
git stash branch <branch name>
```

四、提交与历史

了解了文件的状态，我们对文件进行了必要的修改后，就要把我们所做的修改放入版本库了，这样以后我们就可以在需要的时候恢复到现在的版本，而要恢复到某一版，一般需要查看版本的历史。

提交

提交很简单，直接执行"git commit"。执行 git commit 后会调用默认的或我们设置的编译器要我们填写提示说明，但是提交说明最好按 GIT 要求填写：第一行填简单说明，隔一行填写详细说明。因为第一行在一些情况下会被提取使用，比如查看简短提交历史或向别人提交补丁，所以字符数不应太多，40 为好。下面看一下查看提交历史。

查看提交历史

查看提交历史使用如下图的命令

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git log
commit b1f1ecb70af3c3ce3e945af17f31a56c23551bea
Author: angeldevil <your_email@xxx.com>
Date: Thu Aug 8 11:57:37 2013 +0800

    Second commit

commit 48154652ceb7a74887ef98720b3770ed0a4eff30
Author: angeldevil <your_email@xxx.com>
Date: Thu Aug 8 11:57:06 2013 +0800

    Initiaial commit
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

如图所示，显示了作者，作者邮箱，提交说明与提交时间，"git log"可以使用放多参数，比如：

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git log -1
commit b1f1ecb70af3c3ce3e945af17f31a56c23551bea
Author: angeldevil <your_email@xxx.com>
Date: Thu Aug 8 11:57:37 2013 +0800

    Second commit
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

仅显示最新的 1 个 log，用"-n"表示。

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git log --pretty=oneline --abbrev-
commit
b1f1ecb Second commit
4815465 Initiaial commit
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

显示简单的 SHA-1 值与简单提交说明，oneline 仅显示提交说明的第一行，所以第一行说明最好简单点方便在一行显示。

"git log --graph"以图形化的方式显示提交历史的关系，这就可以方便地查看提交历史的分支信息，当然是控制台用字符画出来的图形。

"git log"的更多参数可以查看命令帮助。

不经过暂存的提交

如果我们想跳过暂存区直接提交修改的文件，可以使用"-a"参数，但要慎重，别一不小心提交了不想提交的文件

```
git commit -a
```

如果需要快捷地填写提交说明可使用"-m"参数

```
git commit -m 'commit message'
```

修订提交

如果我们提交过后发现有个文件改错了，或者只是想修改提交说明，这时可以对相应文件做出修改，将修改过的文件通过"git add"添加到暂存区，然后执行以下命令：

```
git commit --amend
```

然后修改提交说明覆盖上次提交，但只能重写最后一次提交。

重排提交

通过衍合(rebase)可以修改多个提交的说明，并可以重排提交历史，拆分、合并提交，关于 rebase 在讲到分支时再说，这里先看一下重排提交。

假设我们的提交历史是这样的：

```
angeldevil@angeldevil:~/git_demo$ git log
commit 725699b06ae71fff4bb221198ef6c55a591e640b
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 16:17:56 2013 +0800

    Third commit

commit f22a874536aeda084506f9029014e013ae4b12c4
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 15:49:54 2013 +0800

    Second commit

commit 3366e1123010e7d67620ff86040a061ae76de0c8
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 15:49:20 2013 +0800

    Initial commit
angeldevil@angeldevil:~/git_demo$
```

如果我们想重排最后两个提交的提交历史，可以借助交互式 rebase 命令：

```
git rebase -i HEAD~2 或 git rebase -i 3366e1123010e7d67620ff86040a061ae76de0c8
```

HEAD~2 表示倒数第三个提交，这条命令要指定要重排的最旧的提交的父提交，此处要重排 Second commit 与 Third commit，所以要指定 Initial commit 的 Commit ID。如图所示：


```
angeldevil@angeldevil: ~/git_demo
pick f22a874 Second commit
pick 725699b Third commit

# Rebase 3366e11..725699b onto 3366e11
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
~
~
"/git_demo/.git/rebase-merge/git-rebase-todo" 20L, 659C      1,1      全部
```

注释部分详细说明了每个选项的作用，如果我们想交互这两个提交，只需把开头的这两行交换下位置就 OK 了，交换位置后保存，然后看下提交历史：

```
angeldevil@angeldevil:~/git_demo$ git log
commit 6d43f3339fe2937239de5c03186cc9a38d60cc1a
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 15:49:54 2013 +0800

    Second commit

commit 2afc61784a03a4a3b21576580e53f713d9f096d0
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 16:17:56 2013 +0800

    Third commit

commit 3366e1123010e7d67620ff86040a061ae76de0c8
Author: angeldevil <angeldeviljy@gmail.com>
Date: Sat Nov 23 15:49:20 2013 +0800

    Initial commit
angeldevil@angeldevil:~/git_demo$
```

可以看到提交历史已经变了，而且最新的两个提交的 Commit ID 变了，如果这些提交已经 push 到了远程服务器，就不要用这个命令了。

删除提交与修改提交说明

如果要删除某个提交 ,只需要删除相应的行就可以了 ,而要修改某个提交的提交说明的话 ,只需要把相应行的 pick 改为 reward。

合并提交-squashing

合并提交也很简单，从注释中的说明看，只需要把相应的行的 pick 改为 squash 就可以把这个提交全并到它上一行的提交中。

拆分提交

至于拆分提交，由于 Git 不可能知道你要做哪里把某一提交拆分开，所以我们就需要让 Git 在需要拆分的提交处停下来，由我们手动修改提交，这时要把 pick 改为 edit，这样 Git 在处理到这个提交时会停下来，此时我们就可以进行相应的修改并多次提交来拆分提交。

撤销提交

前面说了删除提交的方法，但是如果是多人合作的话，如果某个提交已经 Push 到远程仓库，是不可以用那种方法删除提交的，这时就要撤销提交

```
git revert <commit-id>
```

这条命令会把指定的提交的所有修改回滚，并同时生成一个新的提交。

Reset

git reset 会修改 HEAD 到指定的状态，用法为

```
git reset [options] <commit>
```

这条命令会使 HEAD 指向指定的 Commit，一般会用到 3 个参数，这 3 个参数会影响到工作区与暂存区中的修改：

- --soft: 只改变 HEAD 的 State，不更改工作区与暂存区的内容
- --mixed(默认): 撤销暂存区的修改，暂存区的修改会转移到工作区
- --hard: 撤销工作区与暂存区的修改

cherry-pick

当与别人合作开发时，会向别人贡献代码或者接收别人贡献的代码，有时候可能不想完全 Merge 别人贡献的代码，只想要其中的某一个提交，这时就可以使用 cherry-pick 了。就一个命令

```
git cherry-pick <commit-id>
```

filter-branch

这条命令可以修改整个历史，如从所有历史中删除某个文件相关的信息，全局性地更换电子邮件地址。

五、GIT 分支

分支被称之为 GIT 最强大的特性，因为它非常地轻量级，如果用 Perforce 等工具应该知道，创建分支就是克隆原目录的一个完整副本，对于大型工程来说，太费时费力了，而对于 GIT 来说，可以在瞬间生成一个新的分支，无论工程的规模有多大，因为 GIT 的分支其实就是一指针而已。在了解 GIT 分支之前，应该先了解 GIT 是如何存储数据的。

前面说过，GIT 存储的不是文件各个版本的差异，而是文件的每一个版本存储一个快照对象，然后通过 SHA-1 索引，不只是文件，包换每个提交都是一个对象并通过 SHA-1 索引。无论是文本文件，二进制文件还是提交，都是 GIT 对象。

GIT 对象

每个对象(object) 包括三个部分：**类型**，**大小**和**内容**。大小就是指内容的大小，内容取决于对象的类型，有四种类型的对象："blob"、"tree"、"commit" 和"tag"。

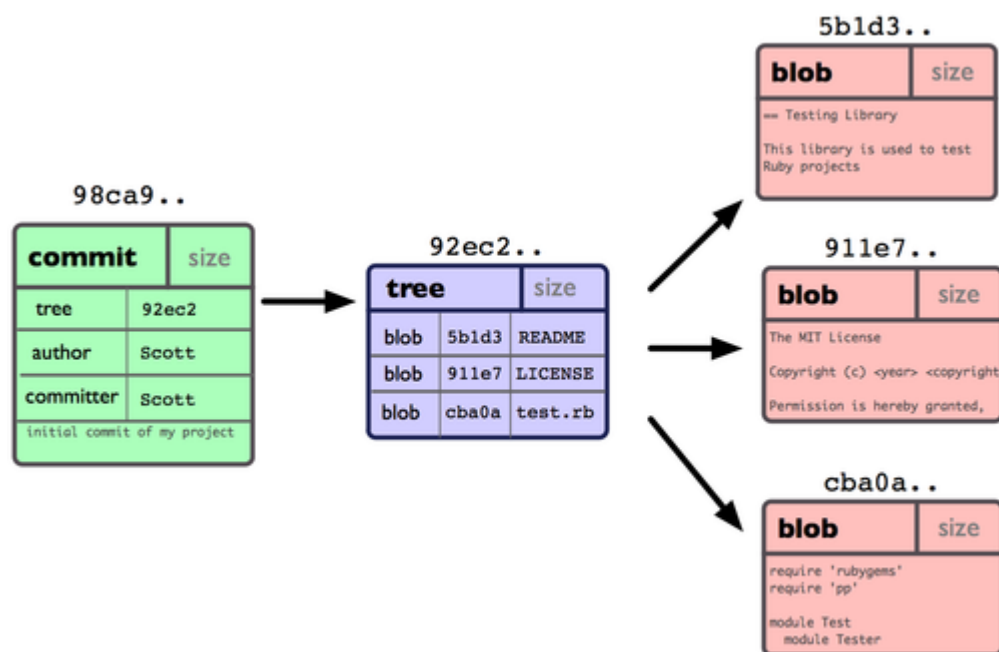
- **"blob"** 用来存储文件数据，通常是一个文件。
- **"tree"** 有点像一个目录，它管理一些 **"tree"** 或是 **"blob"**（就像文件和子目录）
- 一个 **"commit"** 指向一个"tree"，它用来标记项目某一个特定时间点的状态。它包括一些关于时间点的元数据，如提交时间、提交说明、作者、提交者、指向上次提交（commits）的指针等等。
- 一个 **"tag"** 是用来标记某一个提交(commit) 的方法。

比如说我们执行了以下代码进行了一次提交：

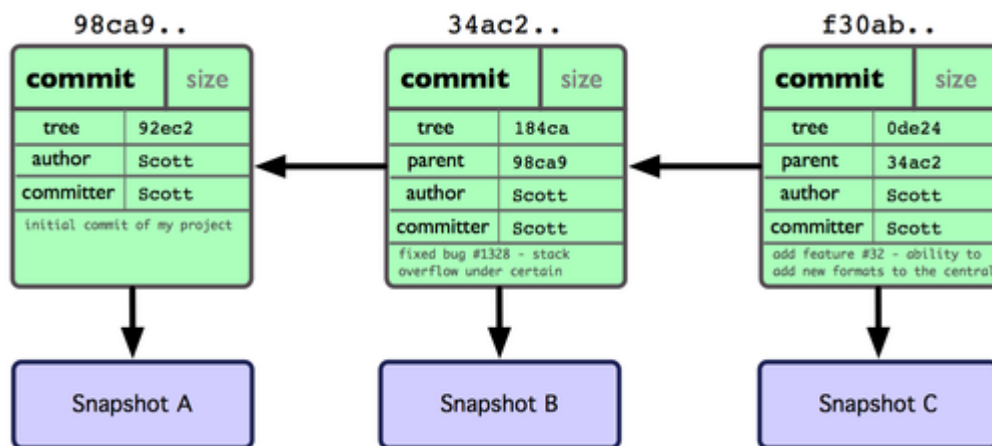
```
$ git add README test.rb LICENSE2
```

```
$ git commit -m 'initial commit of my project'
```

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

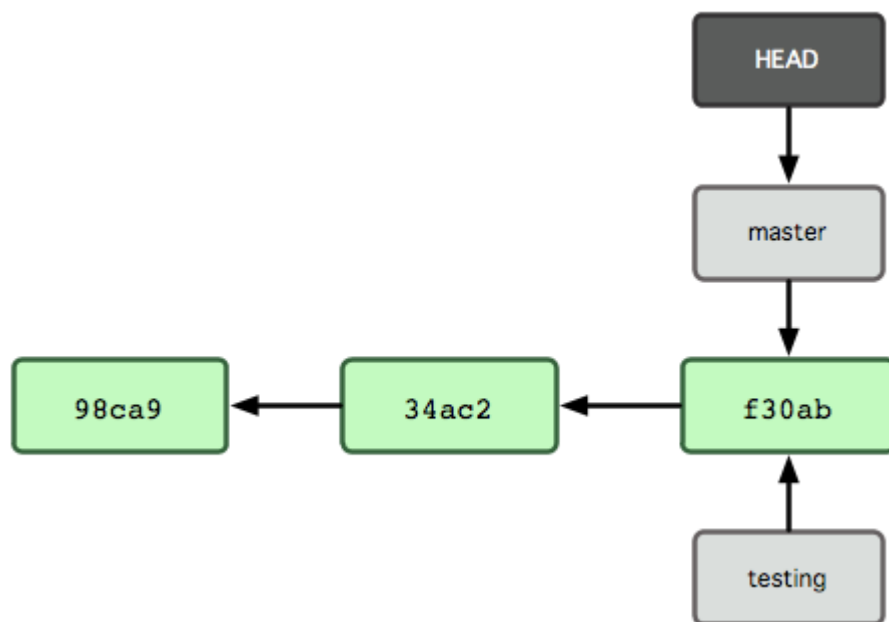


如果进行多次提交，仓库的历史会像这样：



分支引用

所谓的 GIT 分支，其实就是一个指向某一个 Commit 对象的指针，像下面这样，有两个分支，master 与 testing：



而我们怎么知道当前在哪一个分支呢？其实就是很简单地使用了一个名叫 HEAD 的指针，如上图所示。HEAD 指针的值可以作为一个 SHA-1 值或是一个引用，看以下例子：

angeldevil@angeldevil-Lenovo-B470: ~/git_demo

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git log --pretty=oneline
```

```
6c163c06ccfa70c4247f1e6cd5f27b610105d04d rename
```

```
b1f1ecb70af3c3ce3e945af17f31a56c23551bea Second commit
```

```
48154652ceb7a74887ef98720b3770ed0a4eff30 Initial commit
```

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ cat .git/HEAD
```

```
ref: refs/heads/master
```

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git checkout b1f1ecb70
```

```
Note: checking out 'b1f1ecb70'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b new_branch_name
```

```
HEAD is now at b1f1ecb... Second commit
```

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ cat .git/HEAD
```

```
b1f1ecb70af3c3ce3e945af17f31a56c23551bea
```

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

git 的所有版本信息都保存了 Working Directory 下的 .git 目录，而 HEAD 指针就保存在 .git 目录下，如上图所有，目前为止已经有 3 个提交，通过查看 HEAD 的值可以看到我们当前在 master 分支：refs/heads/master，当我们通过 git checkout 取出某一特定提交后，HEAD 的值就是成了我们 checkout 的提交的 SHA-1 值。

记录我们当前的位置很简单，就是能过 HEAD 指针，HEAD 指向某一提交的 SHA-1 值或是某一分支的引用。

新建分支

```
git branch <branch-name>
```

有时需要在新建分支后直接切换到新建的分支，可以直接用 checkout 的 -b 选项

```
git checkout -b <branch-name>
```

删除分支

```
git branch -d <branch-name>
```

如果在指定的分支有一些 unmerged 的提交，删除分支会失败，这里可以使用-D 参数强制删除分支。

```
git branch -D <branch-name>
```

检出分支或提交

检出某一分支或某一提交是同一个命令

```
git checkout <branch-name> | <commit>
```

分支合并(merge)

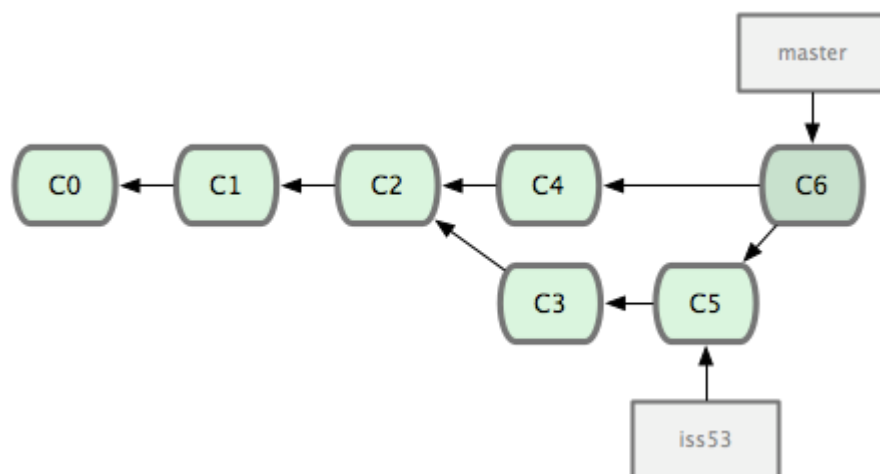
当我们新建一个分支进行开发，并提交了几次更新后，感觉是时候将这个分支的内容合回主线了，这时就可以取出主线分支，然后把分支的更新 merge 回来：

```
git checkout master
```

```
git merge testing
```

如果 master 分支是 testing 分支的直接上游，即从 master 延着 testing 分支的提交历史往前走可以直接走到 testing 分支的最新提交，那么系统什么也不需要，只需要改变 master 分支的指针即可，这被称之为"Fast Forward"。

但是，一般情况是这样的，你取出了最新的 master 分支，比如说 master 分支最新的提交是 C2 (假设共 3 次提交 C0<-C1<-C2)，在此基础上你新建了分支，当你在分支上提交了 C3、C5 后想将 br1 时 merge 回 master 时，你发现已经有其他人提交了 C4，这时候就不能直接修改 master 的指针了，不然会丢失别人的提交，这个时候就需要将你新建分支时 master 所在的提交 (C2) 后的修改 (C4)，与你新建分支后在分支上的修改 (C3、C5) 做合并，将合并后的结果作为一个新的提交提交到 master，GIT 可以自动推导出应该基于哪个提交进行合并 (C2)，如果没有冲突，系统会自动提交新的提交，如果有冲突，系统会提示你解决冲突，当冲突解决后，你就可以将修改加入暂存区并提交。提交历史类似下面这样 (图来自 Pro-Git)：



merge后的提交是按时间排序的 比如下图 我们在rename提交处新建分支 test ,在 test上提交 Commit from branch test ,然后回到 master 提交 commit in master after committing in branch ,再将 test 分支 merge 进 master ,这时看提交提交历史 , Commit from branch test 是在 commit in master...之前的 尽管在 master 上我们是在 rename 的基础上提交的 commit in master... 而 GIT 会在最后添加一个新的提交 (Merge branch 'test') 表示我们在此处将一个分支 merge 进来了。这种情况会有一个问题 , 比如说在 rename 提交处某人 A 从你这里 Copy 了一个 GIT 仓库 , 然后你 release 了一个 patch (通过 git format-patch) 给 A , 这时候 test 分支还没有 merge 进来 , 所以 patch 中只包含提交 : commit in master... 然后你把 test 分支 merge 了进来又给了 A 一个 patch , 这个 patch 会包含提交 : Commit from branch test , 而这个 patch 是以 rename 为 base 的 , 如果 commit in master... 和 Commit

from branch test 修改了相同的文件，则第二次的 patch 可能会打不上去，因为以 rename 为 base 的 patch 可能在新的 Code 上找不到在哪个位置应用修改。

```
angeldevil@angeldevil-Lenovo-B470:~/git_demo$ git log --pretty=oneline
3c19992ab46389fe742a91aae491a08a9999359a Merge branch 'test'
64c1a96a8fb82aa6329eccc0b7d2bfb9bc7376 commit in master after committing in br
ebf98e17b62fb547941f5d9358546e6a4836ba28 Commit from branch test
6c163c06ccfa70c4247f1e6cd5f27b610105d04d rename
b1f1ecb70af3c3ce3e945af17f31a56c23551bea Second commit
48154652ceb7a74887ef98720b3770ed0a4eff30 Initail commit
angeldevil@angeldevil-Lenovo-B470:~/git_demo$
```

分支衍合(rebase)

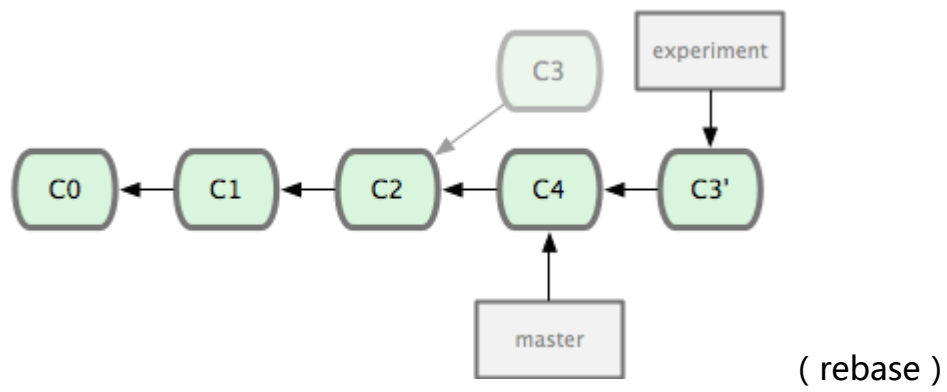
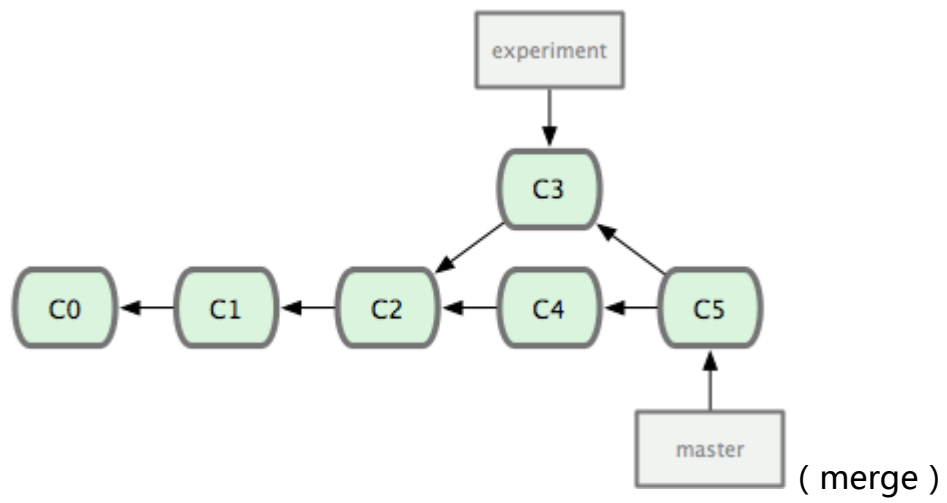
有两种方法将一个分支的改动合并进另一个分支，一个就是前面所说的分支合并，另一个就是分支衍合，这两种方式有什么区别呢？

分支合并 (merge) 是将两个分支的改动合并到一起，并生成一个新的提交，提交历史是按时间排序的，即我们实际提交的顺序，通过 `git log --graph` 或一些图形化工具，可能很明显地看到分支的合并历史，如果分支比较多就很混乱，而且如果以功能点新

建分支，等功能点完成后合回主线，由于 merge 后提交是按提交时间排序的，提交历史就比较乱，各个功能点的提交混杂在一起，还可能遇到上面提到的 patch 问题。

而分支衍合 (rebase) 是找到两个分支的共同祖先提交，将要被 rebase 进来的分支的提交依次在要被 rebase 到的分支上重演一遍，即回到两个分支的共同祖先，将 branch (假如叫 experiment) 的每次提交的差异保存到临时文件里，然后切换到要衍合入的分支 (假如是 master)，依次应用补丁文件。experiment 上有几次提交，在 master 就生成几次新的提交，而且是连在一起的，这样合进主线后每个功能点的提交就都在一起，而且提交历史是线性的

对比 merge 与 rebase 的提交历史会是下图这样的 (图来自 Pro-GIt)：



rebase 后 C3 提交就不存在了，取而代之的是 C3'，而 master 也成为了 experiment 的直接上游，只需一次 Fast Forward (git merge) 后 master 就指向了最新的提交，就可以删除 experiment 分支了。

衍合--onto

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

这条命令的意思是：检出 server 分支与 client 分支共同祖先之后 client 上的变化，然后在 master 上重演一遍。

父提交

HEAD 表示当前所在的提交，如果要查看当前提交父提交呢？git log 查看提交历史，显然太麻烦了，而且输入一长串的 Commit-ID 也不是一个令人愉悦的事。这时可借助两个特殊的符号：~与^。

^ 表示指定提交的父提交，这个提交可能由多个父提交，^之后跟上数字表示第几个父提交，不跟数字等同于^1。

~n 相当于 n 个^，比如~3=^^^，表示第一个父提交的第一个父提交的第一个父提交。

远程分支

远程分支以(远程仓库名)/(分支名)命令，远程分支在本地无法移动修改，当我们 clone 一个远程仓库时会自动在本地生成一个名叫 original 的远程仓库，下载远程仓库的所有数据，并新建一个指向它的分支 original/master，但这个分支我们是无法修改的，所以需要在本地上重新一个分支，比如叫 master，并跟踪远程分支。

Clone 了远程仓库后，我们还会在本地新建其他分支，并且可能也想跟踪远程分支，这时可以用以下命令：

```
git checkout -b [branch_name] --track|-t <remote>/<remote-branch>
```

和新建分支的方法一样，只是加了一个参数--track 或其缩写形式-t，可以指定本地分支的名字，如果不指定就会被命名为 remote-branch。

要拉取某个远程仓库的数据，可以用 git fetch:

```
git fetch <remote>
```

当拉取到了远程仓库的数据后只是把数据保存到了一个远程分支中，如 original/master，而这个分支的数据是无法修改的，此时我们可以把这个远程分支的数据合并到我们当前分支

```
git merge <remote>/<remote-branch>
```

如果当前分支已经跟踪了远程分支，那么上述两个部分就可以合并为一个

```
git pull
```

当在本地修改提交后，我们可能需要把这些本地的提交推送到远程仓库，这里就可以用 git push 命令，由于本地可以由多个远程仓库，所以需要指定远程仓库的名字，并同时指定需要推的本地分支及需要推送到远程仓库的哪一个分支

```
git push <remote> <local-branch>:<remote-branch>
```

如果本地分支与远程分支同名，命令可以更简单

```
git push <remote> <branch-name> 等价于 git push <remote> refs/heads/<branch-name>:refs/for/<branch-name>
```

如果本地分支的名字为空，可以删除远程分支。

前面说过可以有不止一个远程分支 f，添加远程分支的方法为

```
git remote add <short-name> <url>
```

六、标签-tag

作为一个版本控制工具，针对某一时间点的某一版本打 tag 的功能是必不可少的，要查看 tag 也非常简单，查看 tag 使用如下命令

```
git tag
```

参数"-l"可以对 tag 进行过滤

```
git tag -l "v1.1.*"
```

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。

轻量级标签只需在 `git tag` 后加上 tag 的名字，如果 tag 名字

```
git tag <tag_name>
```

含附注的标签需要加上参数 `-a` (annotated)，同时加上 `-m` 跟上标签的说明

```
git tag -a <tag_name> -m "<tag_description>"
```

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` (signed)

查看标签的内容用

```
git show <tag_name>
```

验证已签署的标签用 `-v` (verify)

```
git tag -v <tag_name>
```

有时在某一个版本忘记打 tag 了，可以在后期再补上，只需在打 tag 时加上 `commit-id`

要将 tag 推送到远程服务器上，可以用

```
git push <remote> <tag_name>
```

或者可以用下面的命令推送所有的 tag

```
git push <remote> --tags
```

七、Git 配置

使用"git config"可以配置 Git 的环境变量，这些变量可以存放在以下三个不同的地方：

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

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

最基础的配置是配置 git 的用户，用来标识作者的身份

```
git config --global user.name <name>
```

```
git config --global user.email <email>
```

文本编辑器也可以配置，比如在 git commit 的时候就会调用我们设置的文本编辑器

```
git config --global core.editor vim
```

另一个常用的是 diff 工具，比如我们想用可视化的对比工具

```
git config --global merge.tool meld
```

要查看所有的配置，可以用

```
git config --list
```

或者可以在 git config 后加上配置项的名字查看具体项的配置

```
git config user.name
```

作为一个懒人，虽然 checkout、status 等命令只是一个单词，但是还是嫌太长了，我们还可以给命令设置别名如

```
git config --global alias.co checkout
```

这样 git co 就等于 git checkout

前面说地，git 配置项都保存在那 3 个文件里，可以直接打开相应的配置文件查看配置，也可以直接修改这些配置文件来配置 git，想删除某一个配置，直接删除相应的行就行了

```
[user]
  name = angeldevil
  email = angeldeviljy@gmail.com
[core]
  editor = vim
[color]
  ui = auto
[alias]
  co = checkout
  st = status
  br = branch
~
```

八、其他

关于 GIT 各命令的说明可以查看相关帮助文档，通过以下方法：

```
git help <command>或 git <command> --help
```

REPO

```
repo start <topic_name>
```

开启一个新的主题，其实就是每个 Project 都新建一个分支。

```
repo init -u <url> [OPTIONS]
```

在当前目录下初始化 repo，会在当前目录生成一个.repo 目录，像 Git Project 下的.git 一样，-u 指定 url，可以加参数-m 指定 manifest 文件，默认是 default.xml，.repo/manifests 保存 manifest 文件。.repo/projects 下所有的 project 的数据信息，repo 是一系列 git project 的集合，每个 git project 下的.git 目录中的 refs 等目录都是链接到.repo/manifests 下的。

```
repo manifest
```

可以根据当前各 Project 的版本信息生成一个 manifest 文件

```
repo sync [PROJECT1...PROJECTN]
```

同步 Code。

```
repo status
```

查看本地所有 Project 的修改，在每个修改的文件前有两个字符，第一个字符表示暂存区的状态。

-	no change	same in HEAD and index
A	added	not in HEAD, in index
M	modified	in HEAD, modified in index
D	deleted	in HEAD, not in index
R	renamed	not in HEAD, path changed in index
C	copied	not in HEAD, copied from another in index
T	mode changed	same content in HEAD and index, mode changed
U	unmerged	conflict between HEAD and index; resolution required

每二个字符表示工作区的状态

letter	meaning	description
-	new/unknown	not in index, in work tree
m	modified	in index, in work tree, modified

letter	meaning	description
d	deleted	in index, not in work tree

repo prune <topic>

删除已经 merge 的分支

repo abandon <topic>

删除分支，无论是否 merged

repo branch 或 repo branches

查看所有分支

repo diff

查看修改

```
repo upload
```

上传本地提交至服务器

```
repo forall [PROJECT_LIST] -c COMMAND
```

对指定的 Project 列表或所有 Project 执行命令 COMMAND，加上 -p 参数可打印出 Project 的路径。

```
repo forall -c 'git reset --hard HEAD;git clean -df;git rebase --abort'
```

这个命令可以撤销整个工程的本地修改。

说明：文中关于 Git 的知识大多来自 Pro-Git，这本书感觉不错，想学习的可以找来看：[Pro-Git](#)