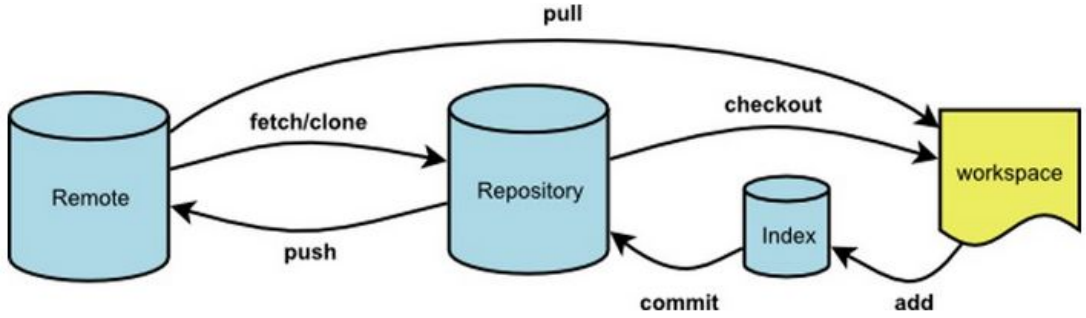
1. Git 简介

git 很显著的一个优点就是版本的分支（branch）和合并（merge）十分方便，传统版本管理软件，分支操作实际上会生成一份现有代码的物理拷贝，这是个昂贵的过程，而 Git 只生成一个指向当前版本（又称"快照"）的指针，难以置信的轻量级使其新建操作几乎可以在瞬间完成，分支切换也是相当的容易。让我们开始理解分支的概念并熟练运用从而真正体会 Git 的如此强大而独特。

最开始的 Subversion 是集中的版本控制系统，它有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。这种集中式版本控制系统最显而易见的缺点是中央服务器的单点故障。如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。要 是中央服务器的磁盘发生故障，碰巧没做备份，或者备份不够及时，就还是会有丢失数据的风险。最坏的情况是彻底丢失整个项目的所有历史更改记录，而被客户端 提取出来的某些快照数据除外，但这样的话依然是个问题，你不能保证所有的数据都已经有人事先完整提取出来过。本地版本控制系统也存在类似问题，只要整个项 目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。相应的 Git 是分布式版本控制系统，客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜 像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份。

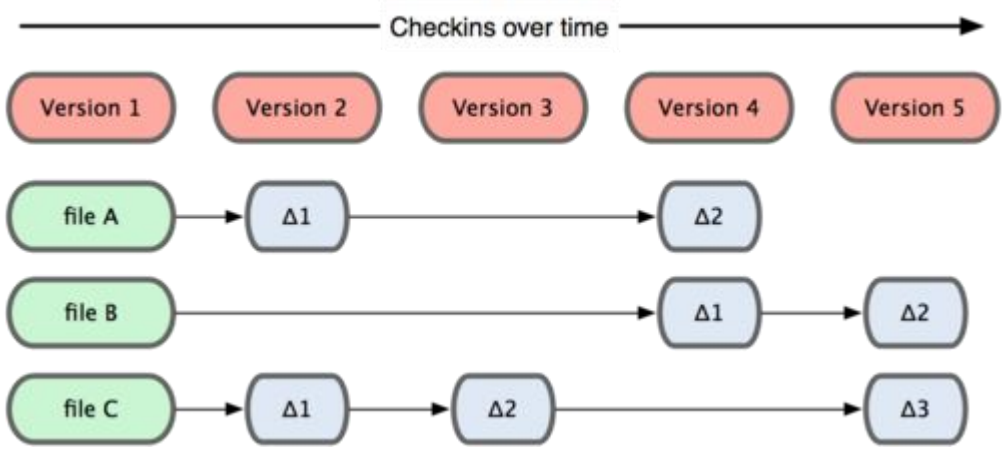
更进一步，git 这类分布式系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程，比如层次模型式的工作流，下图是 git 的工作流



2. Git 思想及基本工作原理

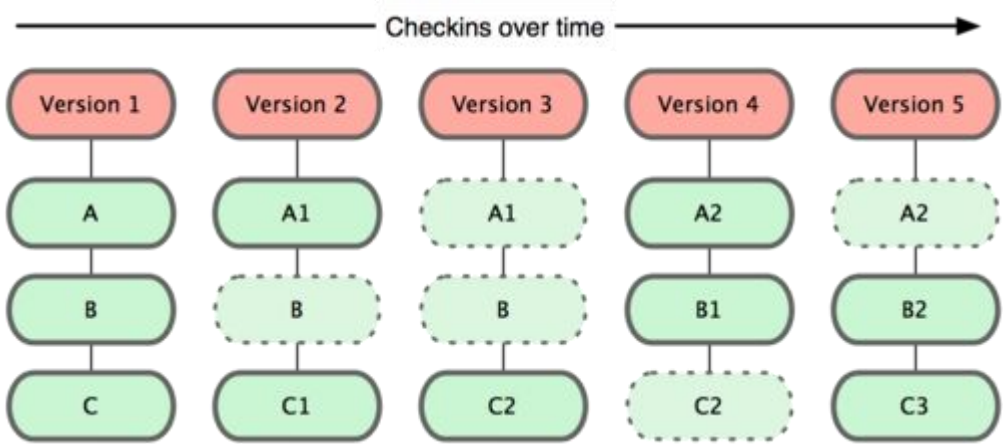
2.1 直接记录快照，而非差异比较

Git 和其他版本控制系统的主要差别在于，Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。比如 Subversion 每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容



可以看到，subversion 在每个版本中保存着每个文件的差异

Git 并不保存这些前后变化的差异数据。实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照 的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一链接



由图可以看到 git 保存每次更新时候的文件快照

2.2 近乎所有操作都是本地执行

在 Git 中的绝大多数操作都只需要访问本地文件和资源，不用连网。但如果用 CVCS 的话，差不多所有操作都需要连接网络。因为 Git 在本地磁盘上就保存着所有当前项目的历史更新，所以处理起来速度飞快。

举个例子，如果要浏览项目的历史更新摘要，Git 不用跑到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。如果想要看当前版本的文件和一个月 前的版本之间有何差异，Git 会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。

用 CVCS 的话，没有网络或者断开 VPN 你就无法做任何事情。但用 Git 的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程仓库

2.3 时刻保持数据完整性

在保存到 Git 之前，所有数据都要进行内容的校验和（checksum）计算，并将此结果作为数据的唯一标识和索引。换句话说，不可能在你修改了文件或目录之后，Git 一无所知。这项特性作为 Git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git 都能立即察觉。

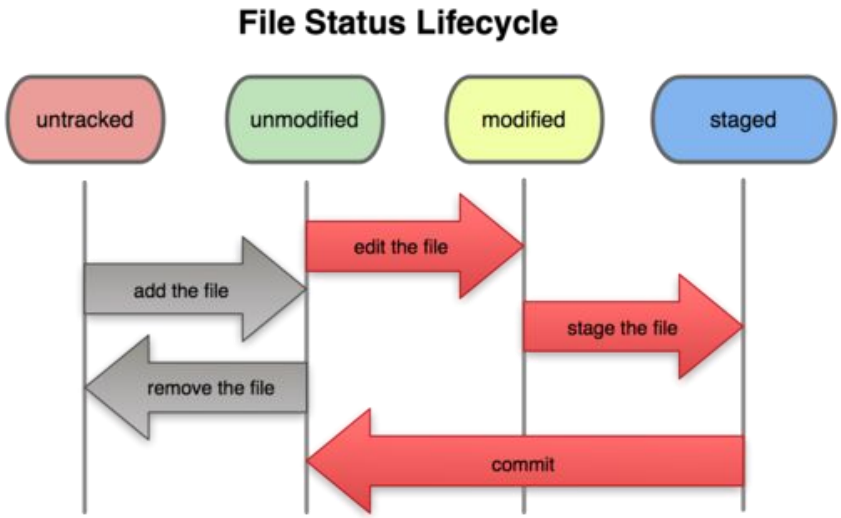
Git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值，作为指纹字符串。该字串由 40 个十六进制字符（0-9 及 a-f）组成，看起来就像是：

24b9da6552252987aa493b52f8696cd6d3b00373

Git 的工作完全依赖于这类指纹字串，所以你会经常看到这样的哈希值。实际上，所有保存在 Git 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

2.4 文件的三种状态

对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。已提交表示该文件已经被安全地保存在本地数据库 中了；已修改表示修改了某个文件，但还没有提交保存；已暂存表示把已修改的文件放在下次提交时要保存的清单中。由此我们看到 Git 管理项目时，文件流转的三个工作区域：Git 的工作目录，暂存区域，以及本地仓库。



每个项目都有一个 Git 目录（译注：如果 git clone 出来的话，就是其中 .git 的目录；如果 git clone --bare 的话，新建的目录本身就是 Git 目录。），它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做工作目录。这些文件实际上都是从 Git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

所谓的暂存区域只不过是个简单的文件，一般都放在 Git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改某些文件。
2. 对修改后的文件进行快照，然后保存到暂存区域。
3. 提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中。

使用 git status 命令

文件在“Changes to be committed”这行下面的，就说明是已暂存状态

文件在“Changed but not staged”这行下面的，说明文件发生了变化，还没有放到暂存区，要暂存此次更新，需要运行 git add（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 ）命令

2.5git 配置

Git 提供了一个叫做 git config 的工具（译注：实际是 git-config 命令，只不过可以通过 git 加一个名字来呼叫此命令。），专门用来配置或读取相应的工作环境变量。而正是由这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

/etc/gitconfig 文件：系统中对所有用户都普遍适用的配置。若使用 git config 时用--system 选项，读写的就是这个文件。

~/.gitconfig 文件：用户目录下的配置文件只适用于该用户。若使用 git config 时用--global 选项，读写的就是这个文件。

当前项目的 git 目录中的配置文件（也就是工作目录中的 .git/config 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以.git/config 里的配置会覆盖/etc/gitconfig 中的同名变量。

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

```
git config --global user.name "robbin"
git config --global user.email "fankai@gmail.com"
git config --global color.ui true
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.br branch
```

Git 需要你输入一些额外消息的时候，会自动调用一个外部文本编辑器给你用。默认会使用操作系统指定的默认编辑器，一般可能会是 Vi 或者 Vim。如果你有其他偏好，比如 Emacs 的话，可以重新设置：

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

差异分析工具 Git 可以理解 kdiff3，tkdiff，meld，xxdiff，emerge，vimdiff，gvimdiff，ecmerge，和 opendiff 等合并工具的输出信息

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

2.6 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。
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'
在提交文件之前首先要添加文件到分支中，很多人只知道：
git add .
如果有文件删除，会发现这些删除的文件并没有被附加进去，肿么办？
#方式一
git add --all .
#方式二
git add -A .

- --all 参数，顾名思义，添加所有文件（change|delete|add）
- -A 参数，添加修改过和删除过的文件（change|delete）

- ✧ 不加 参数，添加修改过和添加的文件（change|add）

通过 git add 之后再运行 git status 命令，会看到文件已被跟踪，并处于暂存状态（在“Changes to be committed”这行下面的，就说明是已暂存状态），也就是说 git add 的潜台词就是把目标文件快照放入暂存区域

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，比如：
$ 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。

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
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；

提交更新

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。
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

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 “Changed but not updated” 部分（也就是_未暂存_清单）看到
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

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
那 git mv 命令是干什么的呢？要在 git 中对文件改名可以如下：

$ git mv file_from file_to
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

查看提交历史

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。
-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
--stat，仅显示简要的增改行数统计
--shortstat 只显示--stat 中最后的行数修改添加移除统计
--name-only 尽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
-n 仅显示最近的若干条提交
--since 按时间做限制的选项，比如 git log --since=2.weeks 表示列出所有最近两周内的提交，可以指定具体的时间：“2008-01-15” 或者“2 years 1 day 3

minutes ago”
--author 显示指定作者的提交
--grep 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。）
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。

--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
git log --pretty=oneline
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
git log --pretty=format:"%h - %an, %ar : %s"
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
%ae	作者的电子邮件地址
%an	作者（author）的名字
%ad	作者修订日期（可以用 -date= 选项定制格式）
%ar	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
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。

命令行输入 gitk 命令会调出 git log 的图形界面，更直观，更方便

撤销操作

当你提交完了之后发现漏掉了几个文件没加或者提交信息写错，可以使用 git commit --amend 命令修改这最后一次的提交。此命令将使用当前的暂存区域快照提交，编辑提交说明确认没问题保存后就会使用新的提交说明覆盖刚才失误的提交。也就是--amend 命令修正了最近一次的提交内容。Cool！

添加远程仓库

git remote add [shoutname] [url] ：添加远程仓库
git remote -v :列出所有远程仓库的信息
git remote show [remote-name] 查看某个远程仓库的详细信息
重命名远程仓库： git remote rename [oldname] [newname]; ---对远程仓库的重命名也会使对应的分支名称发生变化
删除远程仓库命令： git remote rm [reposity-name]

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。

git tag: 列出已有标签

git tag -l 'v1.4.2.*' : 列出 1.4.2 版本的 tag

git tag -a v1.4 -m 'my version 1.4': 使用-a 指定标签名字来创建一个含附注类型的标签；-m 制定了对应的标签说明。

git show tagname: 产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 tag -s v1.5 -m 'my signed 1.5 tag'： 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 tagname: 只需给出标签名的轻量标签

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

可以在后期对早先的某次提交加注标签，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

git push origin [tagname]： 分享标签，也就是将标签推送到远端服务器上

Git push origin --tags: 一次推送所有本地新增的标签上去

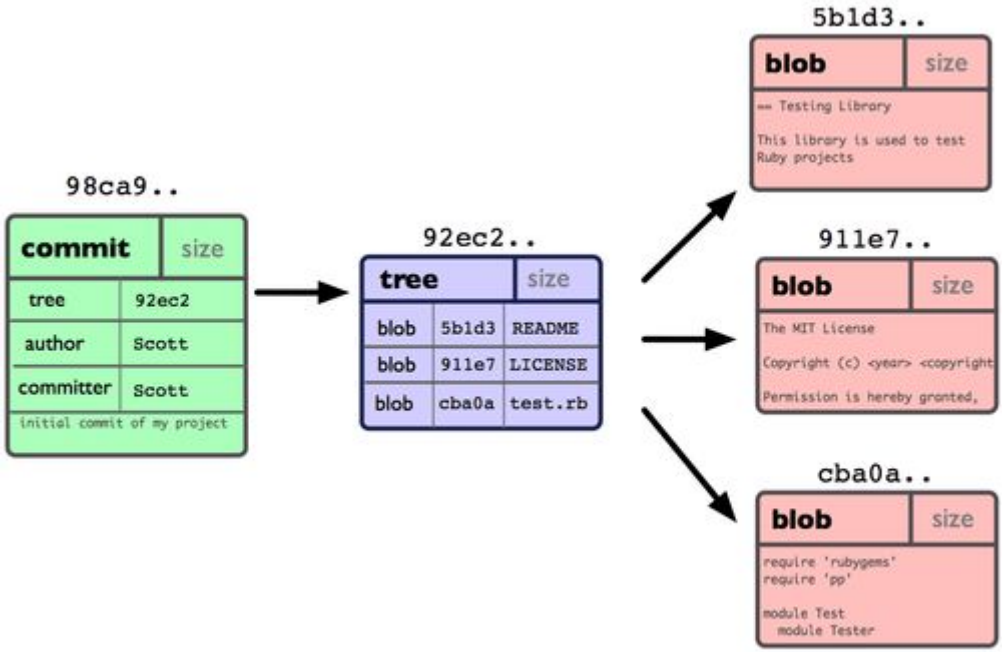
删除 tag：

git push origin --delete tag <tagname>

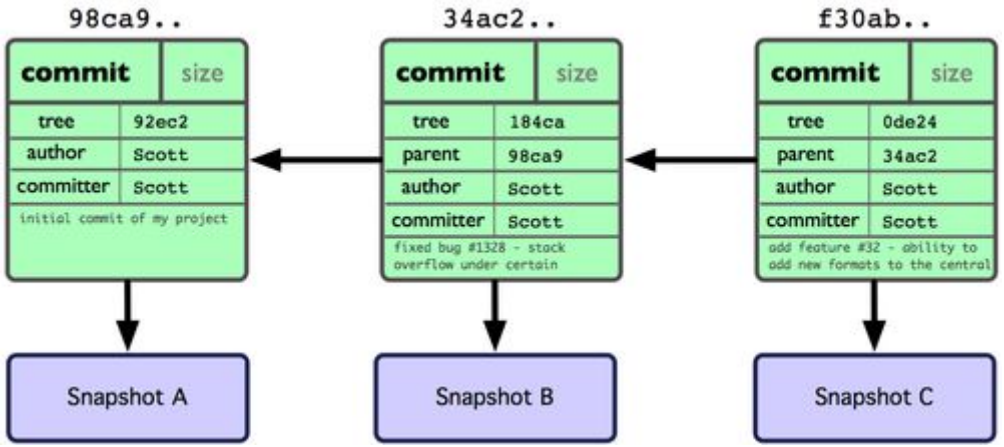
分支

当使用 git commit 新建一个提交对象前，Git 会先计算每一个子目录（本例中就是项目根目录）的校验和，然后在 Git 仓库中将这些目录保存为树（tree）对象。之后 Git 创建的提交对象，除了包含相关提交信息以外，还包含着指向这个树对象（项目根目录）的指针，如此它就可以在将来需要的时候，重现此次快照。

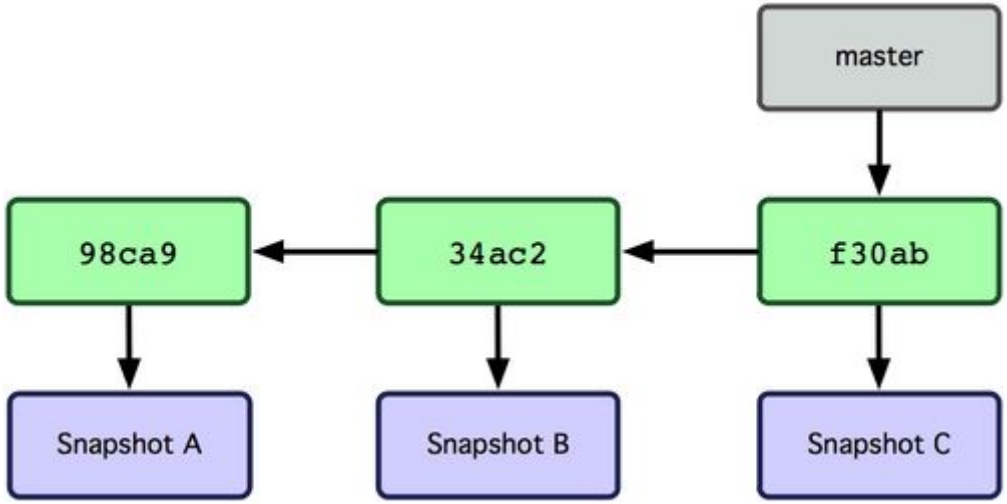
加入我们有三个文件：add README test.rb LICENSE 将它们提交后，Git 仓库中有五个对象：三个表示文件快照内容的 blob 对象；一个记录着目录树内容及其中各个文件对应 blob 对象索引的 tree 对象；以及一个包含指向 tree 对象（根目录）的索引和其他提交信息元数据的 commit 对象。概念上来说，仓库中的各个对象保存的数据和相互关系看起来如图：



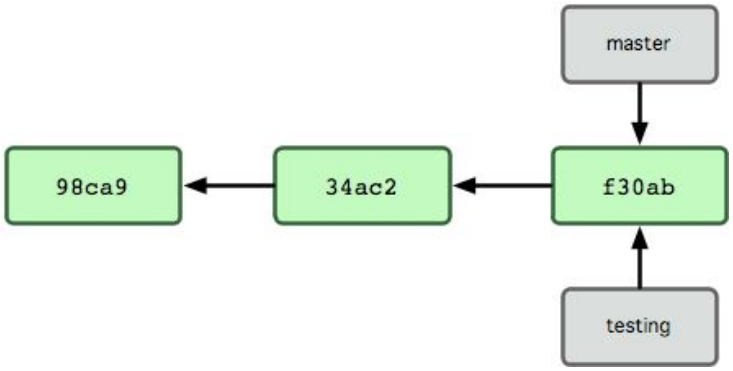
作些修改后再次提交，那么这次的提交对象会包含一个指向上次提交对象的指针（译注：即下图中的 parent 对象）。两次提交后，仓库历史会变成图：



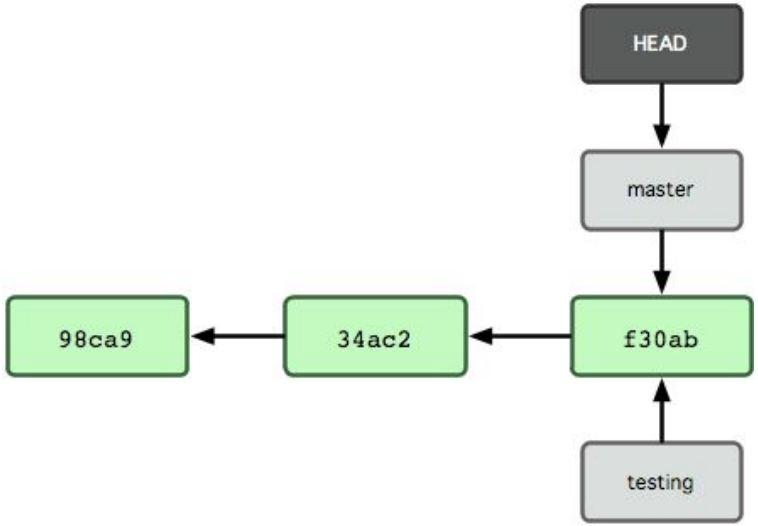
所以说 Git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。Git 会使用 master 作为分支的默认名字。在若干次提交后，你其实已经有了一个指向最后一次提交对象的 master 分支，它在每次提交的时候都会自动向前移动。



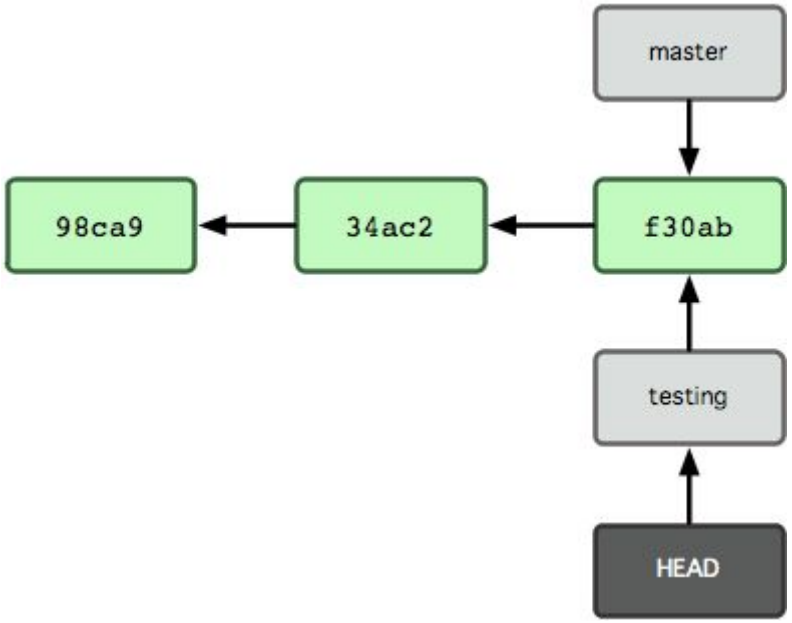
当通过 git branch testing 新建 testing 分支时，git 其实是在当前 commit 对象上创建了一个新的分支指针 testing



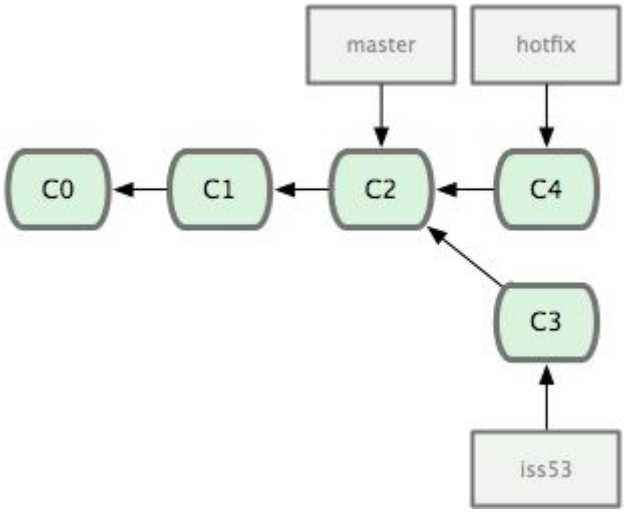
我们知道 git 中有个 HEAD 标志，它其实是一个指向你正在工作中的本地分支的指针，也由此知道你当前是在哪个分支上工作。我们可以将 HEAD 想象为当前分支的别名。注意：git branch branchName 仅仅是建立一个新的分支，不会自动切换到这个分支。如图：HEAD 指向当前所在分支



git checkout testing: 切换到其他分支(testing).
如图：HEAD 在转换分支时指向新的分支



git checkout -b branchName: 创建分支 branchName 并切换到此分支
假如我们在开发网站时，开了个 iss53 分支新增功能，突然收到消息需要紧急修复线上的 bug，我们需要开个 hotfix 分支修复 bug，如图所示：

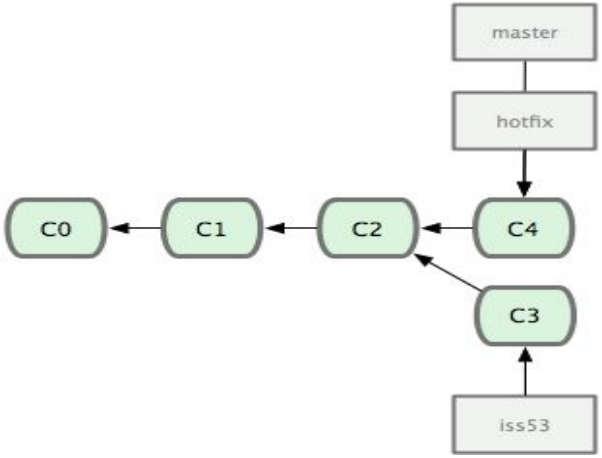


修复完成后切换到 master 分支，并将 hotfix 分支合并进来：

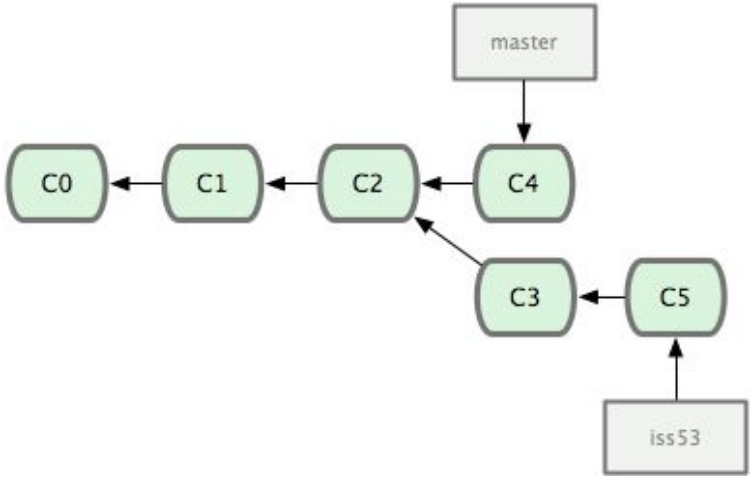
```
$ git checkout master
$ git merge hotfix
   Updating f42c576..3a0874c
   Fast-forward
     README | 1 -
     1 file changed, 1 deletion(-)
```

可以看到合并时出现了“Fast forward”的提示。由于当前 master 分支所在的提交对象是要并入的 hotfix 分支的直接上游，Git 只需把 master 分支指针直接右移。换句话说，如果顺着一个分支走下去可以到达另一个分支的话，那么 Git 在合并两者时，只会简单地把指针右移，因为这种单线的历史分支不存在任何需要解决的分歧，所以这种合并过程可以称为快进（Fast forward）。

现在最新的修改已经在当前 master 分支所指向的提交对象中了，可以部署到生产服务器上去了



修补完 bug 后使用 git branch -d hotfix 删掉 hotfix 分支，然后 git checkout iss53 到 iss53 分支下继续新功能的开发

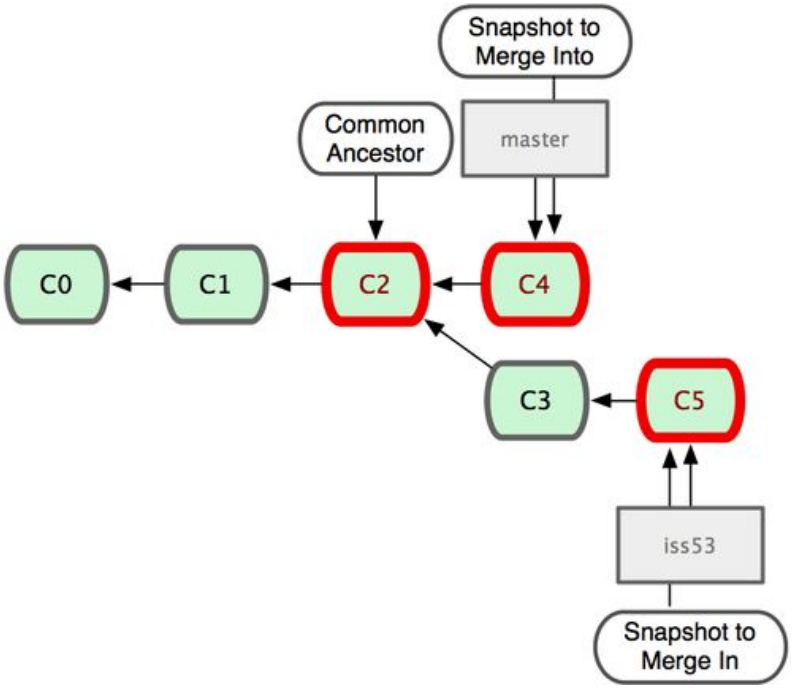


在 iss53 工作完成后可以合并会 master 分支，只需回到 master 分支 git merge iss53 即可将 iss53 分支合并进 master 分支，但是看

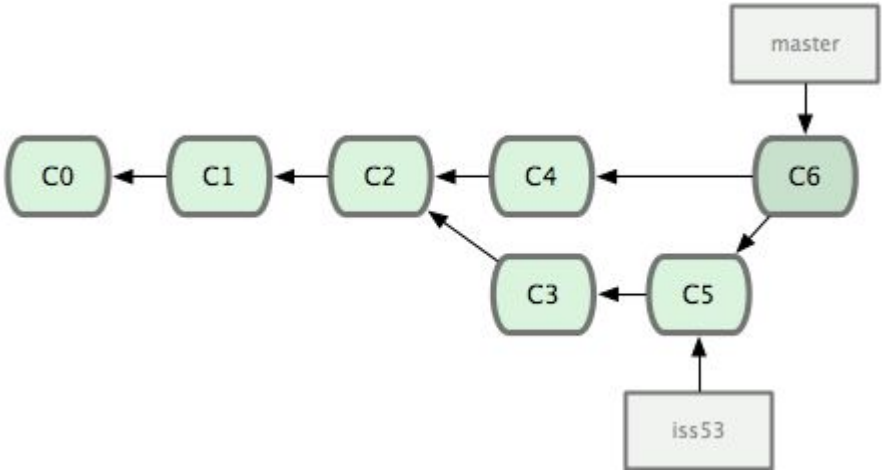
下执行 merge 的 git 提示：

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

这次合并操作的底层实现，并不同于之前 hotfix 的并入方式。因为这次你的开发历史是从更早的地方开始分叉的。由于当前 master 分支所指向的提交对象（C4）并不是 iss53 分支的直接祖先，Git 不得不进行一些额外处理。就此例而言，Git 会用两个分支的末端（C4 和 C5）以及它们的共同祖先（C2）进行一次简单的三方合并计算。如图用红框标出了 Git 用于合并的三个提交对象：



这次，Git 没有简单地把分支指针右移，而是对三方合并后的结果重新做一个新的快照，并自动创建一个指向它的提交对象（C6）。这个提交对象比较特殊，它有两个祖先（C4 和 C5）
值得一提的是 Git 可以自己裁决哪个共同祖先才是最佳合并基础



如果在不同的分支中都修改了同一个文件的同一部分，Git 就无法干净地把两者合到一起，也就是我们常碰到的合并冲突。当发生冲突时，Git 只是 作了合并，但没有提交，它会停下来等你解决冲突。要看看哪些文件在合并时发生冲突，可以用 git status 查阅

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

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

        both modified: index.html
```

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

任何包含未解决冲突的文件都会以未合并（unmerged）的状态列出。Git 会在有冲突的文件里加入标准的冲突解决标记，可以通过它们来手工定位并解决这些冲突。可以看到此文件包含类似下面这样的部分：

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

可以看到 ======= 隔开的上半部分，是 HEAD（即 master 分支，在运行 merge 命令时所切换到的分支）中的内容，下半部分是在 iss53 分支中的内容。在解决了所有文件里的所有冲突后，运行 git add 将把它们标记为已解决状态（译注：实际上就是来一次快照保存到暂存区域。）。因为一旦暂存，就表示冲突已经解决。如果你想用一个有图形界面的工具来解决这些问题，不妨运行 git mergetool，它会调用一个可视化的合并工具并引导你解决所有冲突

git branch： 不加参数会列出当前所有分支的清单

git branch -v: 查看各个分支最后一个提交对象的信息

git branch --merged:查看哪些分支已被并入当前分支

git branch --no-merged: 查看尚未合并的工作

git branch -d branchName: 删除已经合并了的分支，如果分支尚未合并进来删除的时候会提示错误，因为那样会丢失数据，如果确定要删除，可以使用-D来强制删除

远程分支

远程分支（remote branch）是对远程仓库中的分支的索引。它们是一些无法移动的本地分支；只有在 Git 进行网络交互时才会更新。远程分支就像是书

签，提醒着你上次连接远程仓库时上面各分支的位置。
git push (远程仓库名) (branchName)：取出我在本地的 branchName 分支，推送到远程仓库的 branchName 分支中去”
你可以把本地分支推送到某个命名不同的远程分支：若想把远程分支叫作 awesomebranch，可以用 git push origin serverfix:awesomebranch 来推送数据。
在 fetch 操作下载好新的远程分支之后，你仍然无法在本地编辑该远程仓库中的分支。换句话说，执行 git fetch serverfix，你不会有一个新的 serverfix 分支，有的只是一个你无法移动的 origin/serverfix 指针。
如果要把该远程分支的内容合并到当前分支，可以运行 git merge origin/serverfix。如果想要一份自己的 serverfix 来开发，可以在远程分支的基础上分化出一个新的分支来：

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

这会切换到新建的 serverfix 本地分支，其内容同远程分支 origin/serverfix 一致，这样你就可以在里面继续开发了。

跟踪远程分支

如果合并需要采用 rebase 模式，可以使用--rebase 选项。
$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>
如果当前分支与多个主机存在追踪关系，则可以使用-u 选项指定一个默认主机，这样后面就可以不加任何参数使用 git push。
$ git push -u origin master
不带任何参数的 git push，默认只推送当前分支，这叫做 simple 方式。此外，还有一种 matching 方式，会推送所有有对应的远程分支的本地分支。Git 2.0 版本之前，默认采用 matching 方法，现在改为默认采用 simple 方式。如果要修改这个设置，可以采用 git config 命令。
$ git config --global push.default matching
# 或者
$ git config --global push.default simple

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用--all 选项。

$ git push --all origin

上面命令表示，将所有本地分支都推送到 origin 主机。

如果远程主机的版本比本地版本更新，推送时 Git 会报错，要求先在本地做 git pull 合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用--force 选项。

$ git push --force origin

上面命令使用--force 选项，结果导致在远程主机产生一个"非直进式"的合并（non-fast-forward merge）。除非你很确定要这样做，否则应该尽量避免使用--force 选项。

从远程分支 checkout 出来的本地分支，称为 跟踪分支 (tracking branch)。跟踪分支是一种和某个远程分支有直接联系的本地分支。在跟踪分支里输入 git push，Git 会自行推断应该向哪个服务器的哪个分支推送数据。同样，在这些分支里运行 git pull 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

在克隆仓库时，Git 通常会自动创建一个名为 master 的分支来跟踪 origin/master。这正是 git push 和 git pull 一开始就能正常工作的原因。

git checkout -b [localBranch] [远程名]/[remoteBranch]： 用来创建本地分支 localBranch 并跟踪远程分支 remoteBranch，当然本地也会自动切换到此 localBranch

git checkout --track [远程名]/[remoteBranch]： 此结果和上面命令的结果一致，也是创建本地的 remoteBranch 并跟踪远程分支 remoteBranch，且自动切换本地分支到 remoteBranch，显然上面的命令可以自定义本地分支名或者同名，而此命令使得本地分支跟远程分支同名

git push [远程名] : [分支名] ：删除远程的某个分支，比如删除服务器上的 serverfix 可以运行：git push orgin :serverfix

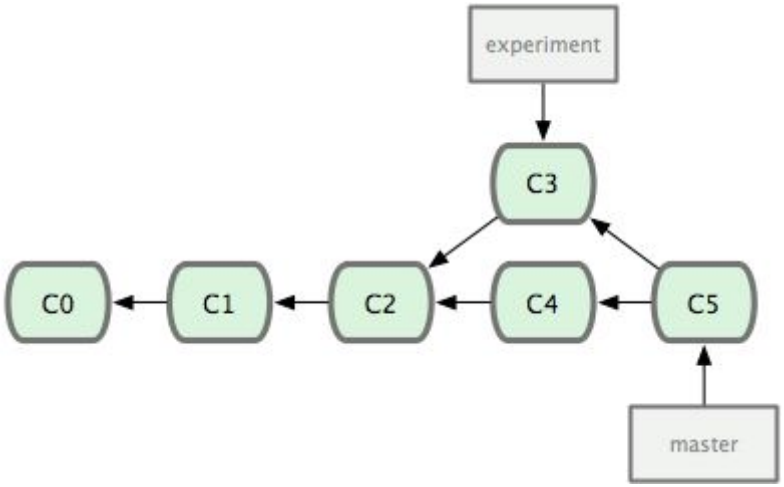
删除远程分支：

git push origin --delete <branch>

记忆此删除命的方式是记住 git push 的语法：git push [远程名] [本地分支]:[远程分支] 语法，如果省略 [本地分支]，那就等于是在说“在这里提取空白然后把它变成[远程分支]”

分支的衍合(rebase)

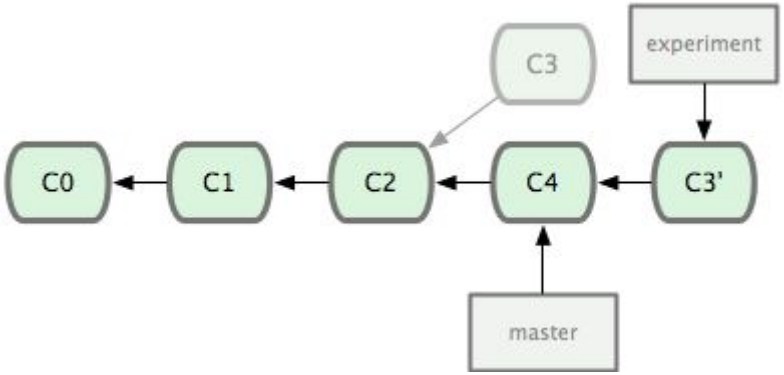
上面有介绍整合分支的 merge 命令，会把两个分支最新的快照（C3 和 C4） 及二者最新的共同祖先（C2） 进行三方合并，合并的结果是产生一个新的提交对象（C5） ，如图：



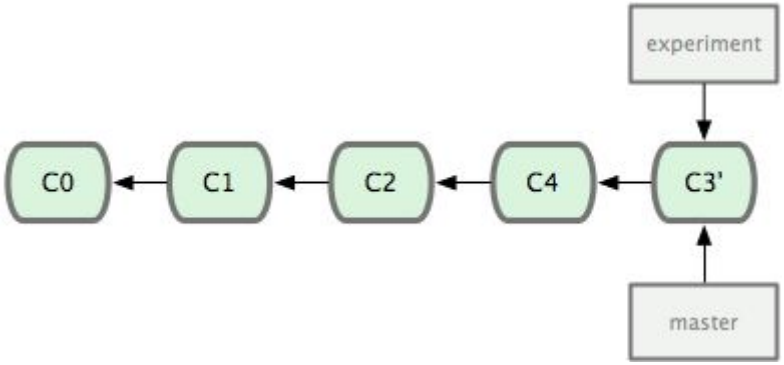
其实，还有另外一个选择：你可以把在 C3 里产生的变化补丁在 C4 的基础上重新打一遍。在 Git 里，这种操作叫做衍合（rebase）。有了 rebase 命令，就可以把在一个分支里提交的改变移到另一个分支里重放一遍。比如这里我们运行：

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

它的原理是回到两个分支最近的共同祖先，根据当前分支（也就是要进行衍合的分支 experiment）后续的历次提交对象（这里只有一个 C3），生成一系列文件补丁，然后以基底分支（也就是主干分支 master）最后一个提交对象（C4）为新的出发点，逐个应用之前准备好的补丁文件，最后会生成一个新的合并提交对象（C3'），从而改写 experiment 的提交历史，使它成为 master 分支的直接下游，如图



现在回到 master 分支，进行一次快进合并，如图：

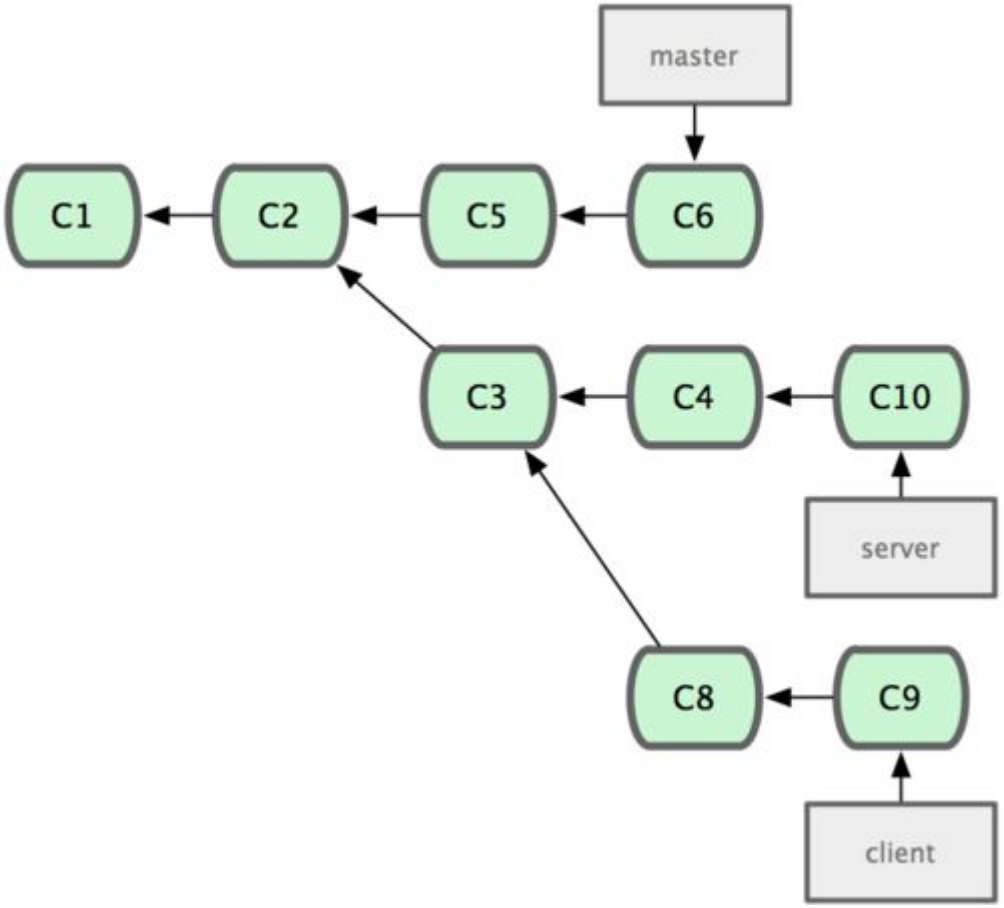


现在的 C3' 对应的快照，其实和普通的三方合并，即上个例子中的 C5 对应的快照内容一模一样了。虽然最后整合得到的结果没有任何区别，但衍合能产生一个更为整洁的提交历史。如果视察一个衍合过的分支的历史记录，看起来会更清楚：仿佛所有修改都是在一根线上先后进行的，尽管实际上它们原本是同时并行发生的。

一般我们使用衍合的目的，是想要得到一个能在远程分支上干净应用的补丁 — 比如某些项目你不是维护者，但想帮点忙的话，最好用衍合：先在自己的一个分支里进行开发，当准备向主项目提交补丁的时候，根据最新的 origin/master 进行一次衍合操作然后再提交，这样维护者就不需要做任何整合工作（译注：实际上是把解决分支补丁同最新主干代码之间冲突的责任，化转为由提交补丁的人来解决。），只需根据你提供的仓库地址作一次快进合并，或者直接采纳你提交的补丁。

请注意，合并结果中最后一次提交所指向的快照，无论是通过衍合，还是三方合并，都会得到相同的快照内容，只不过提交历史不同罢了。衍合是按照每行的修改次序重演一遍修改，而合并是把最终结果合在一起。

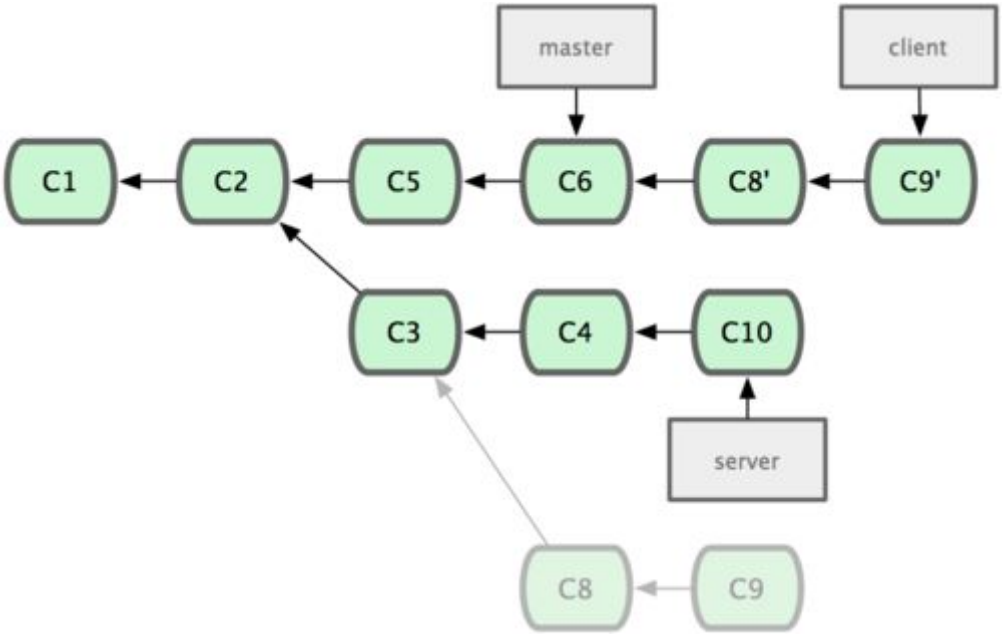
以下图为例看个使用衍合的有趣实例：



假设在接下来的一次软件发布中，我们决定先把客户端的修改并到主线中，而暂缓并入服务端软件的修改（因为还需要进一步测试）。这个时候，我们就可以把基于 client 分支而非 server 分支的改变（即 C8 和 C9），跳过 server 直接放到 master 分支中重演一遍，但这需要用 git rebase 的 --onto 选项指定新的基底分支 master：

$ git rebase --onto master server client

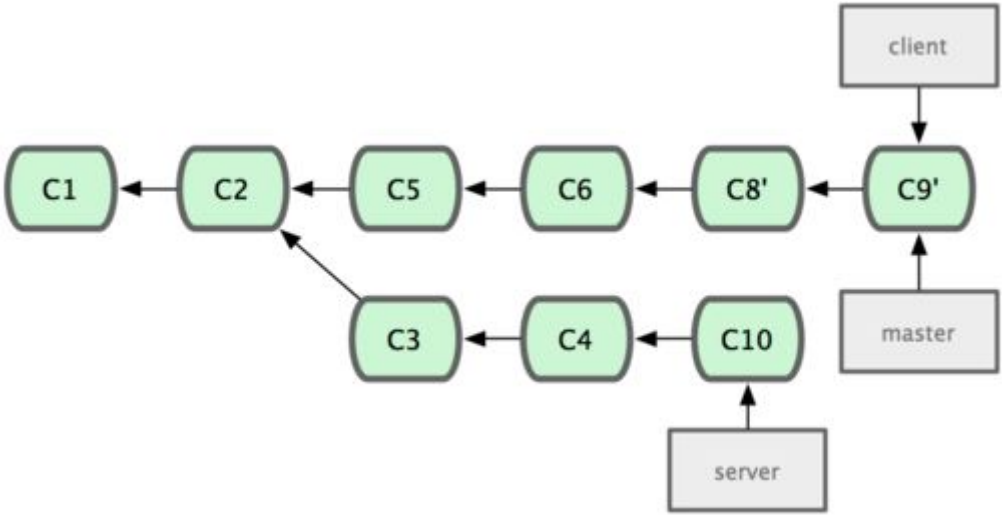
意思是说：取出 client 分支，找出 client 分支和 server 分支的共同祖先之后的变化，然后把它们在 master 上重演一遍，如图：



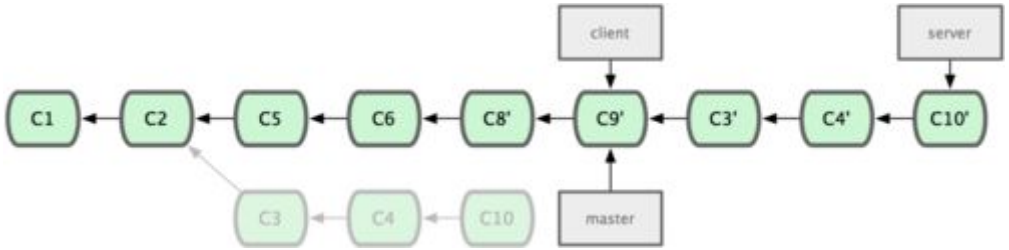
然后就可以快进 master 分支了：

$ git checkout master

$ git merge client



于是，server 的进度应用到 master 的基础上

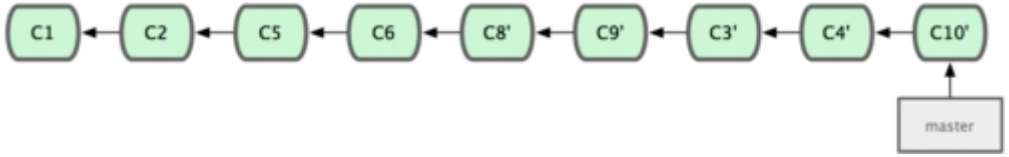


然后就可以快进主干分支 master 了：

$ git checkout master

$ git merge server

现在 client 和 server 分支的变化都已经集成到主干分支来了，可以删掉它们了。最终我们的提交历史会变成图

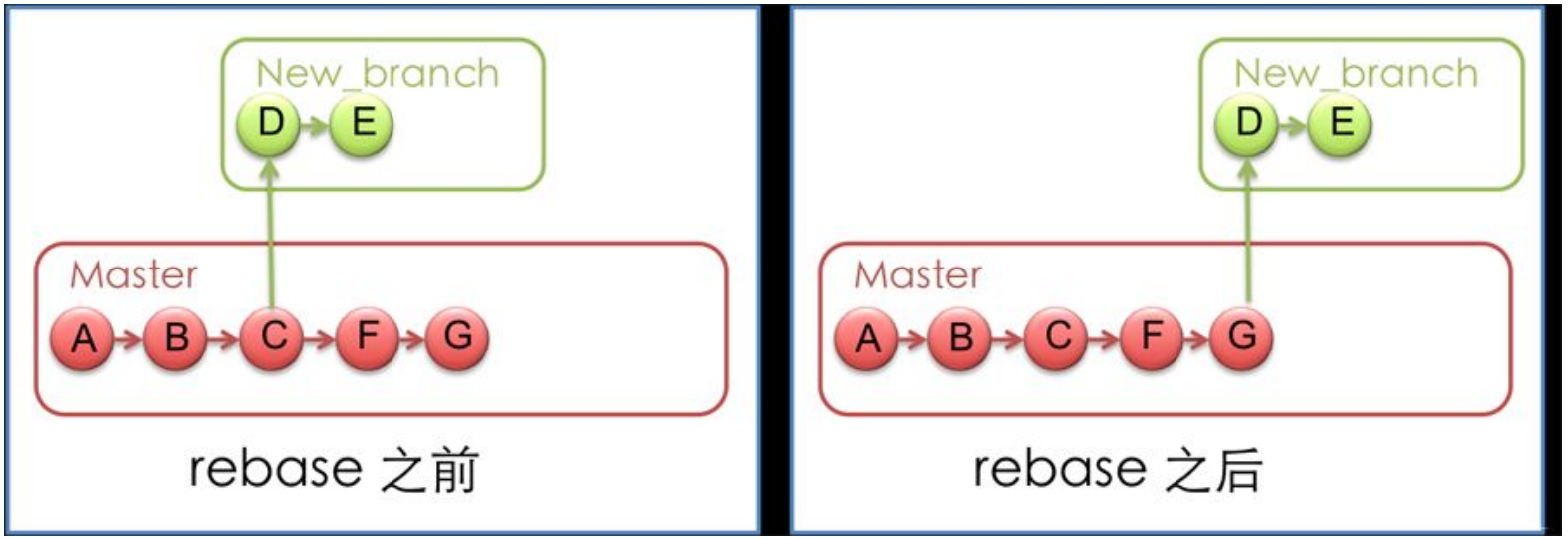


现在我们决定把 server 分支的变化也包含进来。我们可以直接把 server 分支衍合到 master，而不用手工切换到 server 分支后再执行衍合操作 — git rebase [主分支] [特性分支] 命令会先取出特性分支 server，然后在主分支 master 上重演：

$ git rebase master server

如果把衍合当成一种在推送之前清理提交历史的手段，而且仅仅衍合那些尚未公开的提交对象，就没问题。如果衍合那些已经公开的提交对象，并且已经有人基于这些提交对象开展了后续开发工作的话，就会出现叫人沮丧的麻烦。尽量避免！！

再用下图来说明下： rebase 命令执行后，实际上是将分支点从 C 移到了 G，这样分支也就具有了从 C 到 G 的功能。



要修改历史中更早的提交，可以使用 rebase 工具来衍合一系列的提交到他们原来所在的 HEAD 上而不是移到新的上。
依靠这个交互式的 rebase 工具，你就可以停留在每一次提交后，如果你想修改或改变说明、增加文件或任何其他事情。你可以通过给 git rebase 增加-i 选项来以交互方式地运行 rebase。你必须通过告诉命令衍合到哪次提交，来指明你需要重写的提交的回溯深度。
例如，你想修改最近三次的提交说明，或者其中任意一次，你必须给 git rebase -i 提供一个参数，指明你想要修改的提交的父提交，例如 HEAD~2 或者 HEAD~3。可能记住~3 更加容易，因为你想修改最近三次提交；但是请记住你事实上所指的是四次提交之前，即你想修改的提交的父提交。
$ git rebase -i HEAD~3
再次提醒这是一个衍合命令——HEAD~3..HEAD 范围内的每一次提交都会被重写，无论你是否修改说明。不要涵盖你已经推送到中心服务器的提交——这么做会使其他开发者产生混乱，因为你提供了同样变更的不同版本。
运行这个命令会为你的文本编辑器提供一个提交列表，看起来像下面这样
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

Rebase 710f0f8..a5f4a0d onto 710f0f8
#
Commands:
p, pick = use commit
e, edit = use commit, but stop for amending
s, squash = use commit, but meld into previous commit
#
If you remove a line here THAT COMMIT WILL BE LOST.
However, if you remove everything, the rebase will be aborted.
#
很重要的一点是你得注意这些提交的顺序与你通常通过 log 命令看到的是相反的。如果你运行 log，你会看到下面这样的结果：

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
```

```
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

请注意这里的倒序。交互式的 rebase 给了你一个即将运行的脚本。它会从你在命令行上指明的提交开始(HEAD~3)然后自上至下重播每次提交里引入的变更。它将最早的列在顶上而不是最近的，因为这是第一个需要重播的。
你需要修改这个脚本来让它停留在你想修改的变更上。要做到这一点，你只要将你想修改的每一次提交前面的 pick 改为 edit。例如，只想修改第三次提交说明的话，你就像下面这样修改文件：

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

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

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

      git commit --amend

Once you're satisfied with your changes, run

      git rebase --continue
```

这些指示很明确地告诉了你该干什么。输入
$ git commit --amend

修改提交说明，退出编辑器。然后，运行
$ git rebase --continue
这个命令会自动应用其他两次提交，你就完成任务了。如果你将更多行的 pick 改为 edit ，你就能对你想修改的提交重复这些步骤。Git 每次都会停下，让你修正提交，完成后继续运行。

重排提交

你也可以使用交互式的衍合来彻底重排或删除提交。如果你想删除"added cat-file"这个提交并且修改其他两次提交引入的顺序，你将 rebase 脚本从这个

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

改为这个：

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

当你保存并退出编辑器，Git 将分支倒回至这些提交的父提交，应用 310154e，然后 f7f3f6d，接着停止。你有效地修改了这些提交的顺序并且彻底删除了"added cat-file"这次提交。

压制(squashing)提交

交互式的衍合工具还可以将一系列提交压制为单一提交。脚本在 rebase 的信息里放了一些有用的指示：

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

如果不用"pick"或者"edit"，而是指定"squash"，Git 会同时应用那个变更和它之前的变更并将提交说明归并。因此，如果你想将这三个提交合并为单一提交，你可以将脚本修改成这样：

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

当你保存并退出编辑器，Git 会应用全部三次变更然后将你送回编辑器来归并三次提交说明。

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame
```

```
# This is the 3rd commit message:

added cat-file
```

当你保存之后，你就拥有了一个包含前三次提交的全部变更的单一提交。

拆分提交

拆分提交就是撤销一次提交，然后多次部分地暂存或提交直到结束。例如，假设你想将三次提交中的中间一次拆分。将"updated README formatting and added blame"拆分成两次提交：第一次为"updated README formatting"，第二次为"added blame"。你可以在 rebase -i 脚本中修改你想拆分的提交前的指令为"edit"：

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

然后，这个脚本就将你带入命令行，你重置那次提交，提取被重置的变更，从中创建多次提交。当你保存并退出编辑器，Git 倒回到列表中第一次提交的父提交，应用第一次提交（f7f3f6d），应用第二次提交（310154e），然后将你带到控制台。那里你可以用 git reset HEAD^对那次提交进行一次混合的重置，这将撤销那次提交并且将修改的文件撤回。此时你可以暂存并提交文件，直到你拥有多次提交，结束后，运行 git rebase --continue。

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
```

```
$ git commit -m 'added blame'
$ git rebase --continue
```

Git 在脚本中应用了最后一次提交（a5f4a0d），你的历史看起来就像这样了：

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

再次提醒，这会修改你列表中的提交的 SHA 值，所以请确保这个列表里不包含你已经推送到共享仓库的提交。

filter-branch 从所有提交中删除一个文件

比如要从整个历史中删除一个名叫 password.txt 的文件，可以再 filter-barnch 上使用--tree-filter 选项：

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

--tree-filter 选项会在每次检出项目时先执行指定的命令然后重新提交结果。在这个例子中，你会在所有快照中删除一个名叫 password.txt 的文件，无论它是否存在。如果你想删除所有不小心提交上去的编辑器备份文件，你可以运行类似 git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD 的命令。
你可以观察到 Git 重写目录树并且提交，然后将分支指针移到末尾。一个比较好的办法是在一个测试分支上做这些然后在你确定产物真的是你所要的之后，再 hard-reset 你的主分支。要在你所有的分支上运行 filter-branch 的话，你可以传递一个--all 给命令。

将一个子目录设置为新的根目录

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

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

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

全局性地更换电子邮件地址

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

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

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

Git 调试之文件标注

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

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

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

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
```

```
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

子模块 submodule 功能详解

项目的版本库在某些情况下需要引用其他版本库中的文件，例如有一套公用的代码库，可以被多个项目调用，这个公用代码库能直接放在某个项目的代码中，而且要独立为一个代码库，那么其他要调用公用的代码库该如何处理？分别把公用的代码库拷贝到各自的项目中会造成冗余，丢弃了公共代码库的维护历史，这些显然不是好的办法，现在要了解的 git 子模组(git submodule)就解决了这个问题。
Git 子模块功能允许你将一个 Git 仓库当作另外一个 Git 仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。
这里不做详细说明，需要用到的时候可以参见 http://git-scm.com/book/zh/v1/Git-%E5%B7%A5%E5%85%B7-%E5%AD%90%E6%A8%A1%E5%9D%97

回滚代码：

git revert HEAD
你也可以 revert 更早的 commit，例如：
git revert HEAD^
销毁自己的修改
git reset --hard

Git reset 命令解析

我们知道 git 在初始化时，会为我们默认创建一个 master 分支，那这个 master 到底是什么呢？其实它在.git 目录下对应了一个引用文件-----.git/refs/heads/master 文件，而该文件的内容便是该分支中最新的一次提交的 ID：

```
$ cat .git/refs/heads/master
22f8aae534916e1174711f138573acfbb47e489c

$ git cat-file -t 22f8aae
commit

$ git log --oneline
22f8aae git ignore file added.
4e79a0b Rename third to third.txt
46ae7e3 remove branch_first.txt file from master branch
55d1e70 branch first file
545a382 Merge commit '35bbd32'
```

可见，master 文件果然记录了最新一次的提交，那么 git reset 到底做了什么？
同样以上例为例，我们来执行一个 reset 命令，reset 到上一次提交

```
$ git reset --hard HEAD^
HEAD is now at 4e79a0b Rename third to third.txt

Quadrangle@QUADRANGLE-PC /d/GitRepo/GitOne (master)
$ git log --oneline
4e79a0b Rename third to third.txt
46ae7e3 remove branch_first.txt file from master branch
55d1e70 branch first file
545a382 Merge commit '35bbd32'
```

我们看到提交日志里，发现最新一次提交没了，那么我们还能找回最新的一次提交么？回答这个问题我们需要用到 master 的引用文件，看下这文件的内容发生了什么变化？

```
$ cat .git/refs/heads/master
4e79a0ba92f1ae63dc661e29343fa0c369ca480d
```

看到里面记录了 reset 后的最后一次提交，其实 reset 后，git 没有删除最新提交的相关信息，包括目录树，因此只要记住提交 ID，便可以重新 reset 回来。

```
$ git reset --hard 22f8aae
HEAD is now at 22f8aae git ignore file added.

$ git log --oneline
22f8aae git ignore file added.
4e79a0b Rename third to third.txt
46ae7e3 remove branch_first.txt file from master branch
55d1e70 branch first file
545a382 Merge commit '35bbd32'
```

查看 log 发现最新提交又回来了，如果我们忘了最新提交 ID 怎么办？Reflog 命令可以汇总引用变更的记录：

```
Quadrangle@QUADRANGLE-PC /d/GitRepo/GitOne (master)
$ git reflog
22f8aae HEAD@{0}: reset: moving to 22f8aae
4e79a0b HEAD@{1}: reset: moving to HEAD^
22f8aae HEAD@{2}: reset: moving to 22f8aae
4e79a0b HEAD@{3}: reset: moving to HEAD^
```

我们可以找到前面任意操作的记录，并且可以 reset 到任意提交

查看最新版本和上一个版本的差异(一个^表示向前推进一个版本)
git diff HEAD HEAD^
git 取消已经缓存的文件(慎用)：
git reset
git 恢复删除了的文件，git pull 从 git 服务器取出，并且和本地修改 merge， 类似于 SVN up，但是对删除的文件不管用，恢复删除文件用
git checkout -f

应用补丁

两种应用补丁的方法：git apply 或者 git am。
收到的补丁文件是用 git diff 或由其它 Unix 的 diff 命令生成，就该用 git apply 命令来应用补丁，比如：
$ git apply /tmp/patch-ruby-client.patch
这会修改当前工作目录下的文件，效果基本与运行 patch -p1 打补丁一样，但它更为严格，且不会出现混乱。如果是 git diff 格式描述的补丁，此命令还会相应地添加，删除，重命名文件。当然，普通的 patch 命令是不会这么做的。另外请注意，git apply 是一个事务性操作的命令，也就是说，要么所有补丁都打上去，要么全部放弃。所以不会出现 patch 命令那样，一部分文件打上了补丁而另一部分却没有，这样一种不上不下的修订状态。所以总的来说，git apply 要比 patch 严谨许多。因为仅仅是更新当前的文件，所以此命令不会自动生成提交对象，你得手工缓存相应文件的更新状态并执行提交命令。
在实际打补丁之前，可以先用 git apply --check 查看补丁是否能够干净顺利地应用到当前分支中：

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

如果没有任何输出，表示我们可以顺利采纳该补丁。如果有问题，除了报告错误信息之外，该命令还会返回一个非零的状态，所以在 shell 脚本里可用于检测状态。

使用 am 命令应用补丁

如果贡献者也用 Git，且擅于制作 format-patch 补丁，那你的合并工作将会非常轻松。因为这些补丁中除了文件内容差异外，还包含了作者信息和提交消息。所以请鼓励贡献者用 format-patch 生成补丁。对于传统的 diff 命令生成的补丁，则只能用 git apply 处理。
对于 format-patch 制作的新式补丁，应当使用 git am 命令。从技术上来说，git am 能够读取 mbox 格式的文件。这是种简单的纯文本文件，可以包含多封电邮，格式上用 From 加空格以及随便什么辅助信息所组成的行作为分隔行，以区分每封邮件，就像这样：

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

这是 format-patch 命令输出的开头几行，也是一个有效的 mbox 文件格式。如果有人用 git send-email 给你发了一个补丁，你可以将此邮件下载到本地，然后运行 git am 命令来应用这个补丁。如果你的邮件客户端能将多封电邮导出为 mbox 格式的文件，就可以用 git am 一次性应用所有导出的补丁。

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

你会看到它被干净地应用到本地分支，并自动创建了新的提交对象。作者信息取自邮件头 From 和 Date，提交消息则取自 Subject 以及正文中补丁之前的内容。来看具体实例，采纳之前展示的那个 mbox 电邮补丁后，最新的提交对象为：

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700
```

```
   add limit to log function

   Limit log functionality to the first 20
```

Commit 部分显示的是采纳补丁的人，以及采纳的时间。而 Author 部分则显示的是原作者，以及创建补丁的时间。
有时，我们也会遇到打不上补丁的情况。这多半是因为主干分支和补丁的基础分支相差太远，但也可能是因为某些依赖补丁还未应用。这种情况下，git am 会报错并询问该怎么做：

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Git 会在有冲突的文件里加入冲突解决标记，这同合并或衍合操作一样。解决的办法也一样，先编辑文件消除冲突，然后暂存文件，最后运行 git am --resolved 提交修正结果：

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

如果想让 Git 更智能地处理冲突，可以用 -3 选项进行三方合并。如果当前分支未包含该补丁的基础代码或其祖先，那么三方合并就会失败，所以该选项默认为关闭状态。一般来说，如果该补丁是基于某个公开的提交制作而成的话，你总是可以通过同步来获取这个共同祖先，所以用三方合并选项可以解决很多麻烦：

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

像上面的例子，对于打过的补丁我又再打一遍，自然会产生冲突，但因为加上了 -3 选项，所以它很聪明地告诉我，无需更新，原有的补丁已经应用。
对于一次应用多个补丁时所用的 mbox 格式文件，可以用 am 命令的交互模式选项 -i，这样就会在打每个补丁前停住，询问该如何操作：

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

在多个补丁要打的情况下，这是个非常好的办法，一方面可以预览下补丁内容，同时也可以有选择性的接纳或跳过某些补丁。
打完所有补丁后，如果测试下来新特性可以正常工作，那就可以安心地将当前特性分支合并到长期分支中去了。

检出远程分支

如果贡献者有自己的 Git 仓库，并将修改推送到此仓库中，那么当你拿到仓库的访问地址和对应分支的名称后，就可以加为远程分支，然后在本地进行合并。
比如，Jessica 发来一封邮件，说在她代码库中的 ruby-client 分支上已经实现了某个非常棒的新功能，希望我们能帮忙测试一下。我们可以先把她的仓库加为远程仓库，然后抓取数据，完了再将她所说的分支检出到本地来测试：

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
```

$ git checkout -b rubyclient jessica/ruby-client
若是不久她又发来邮件，说还有个很棒的功能实现在另一分支上，那我们只需重新抓取下最新数据，然后检出那个分支到本地就可以了，无需重复设置远程仓库。

一般我们会先看下，特性分支上都有哪些新增的提交。比如在 contrib 特性分支上打了两个补丁，仅查看这两个补丁的提交信息，可以用 --not 选项指定要屏蔽的分支 master，这样就会剔除重复的提交历史：

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

如果想看当前分支同其他分支合并时的完整内容差异，有个小窍门：
$ git diff master
虽然能得到差异内容，但请记住，结果有可能和我们的预期不同。一旦主干 master 在特性分支创建之后有所修改，那么通过 diff 命令来比较的，是最新主干上的提交快照。显然，这不是我们所要的。比方在 master 分支中某个文件里添了一行，然后运行上面的命令，简单的比较最新快照所得到的结论只能是，特性分支中删除了这一行。
这个很好理解：如果 master 是特性分支的直接祖先，不会产生任何问题；如果它们的提交历史在不同的分叉上，那么产生的内容差异，看起来就像是增加了特性分支上的新代码，同时删除了 master 分支上的新代码。
实际上我们真正想要看的，是新加入到特性分支的代码，也就是合并时会并入主干的代码。所以，准确地讲，我们应该比较特性分支和它同 master 分支的共同祖先之间的差异。
我们可以手工定位它们的共同祖先，然后与之比较：

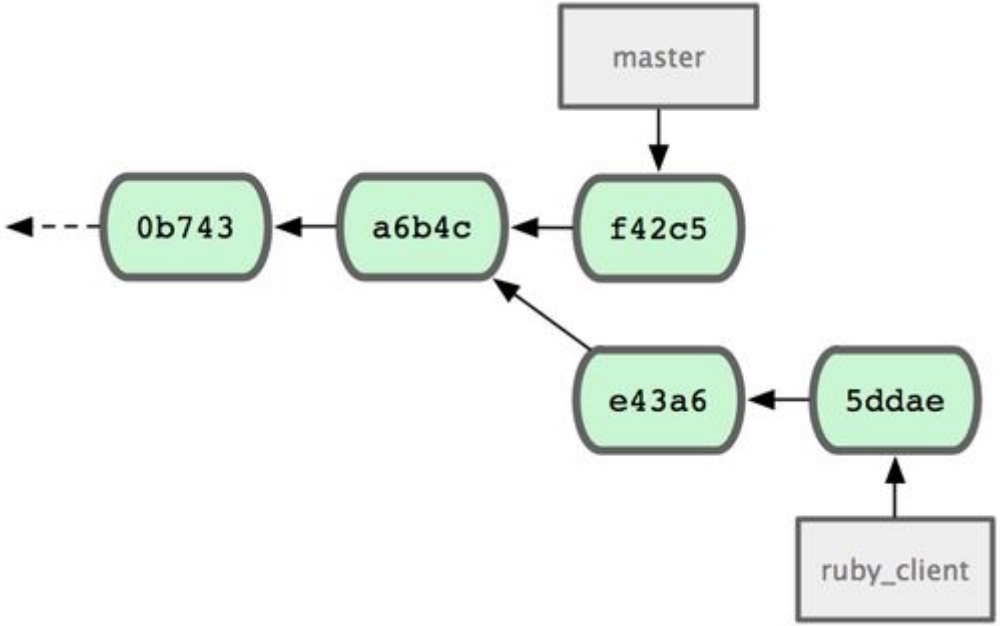
```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

但这么做很麻烦，所以 Git 提供了便捷的 ... 语法。对于 diff 命令，可以把 ... 加在原始分支（拥有共同祖先）和当前分支之间：
$ git diff master...contrib
现在看到的，就是实际将要引入的新代码。这是一个非常有用的命令，应该牢记。

简短的 SHA 描述：
Git 可以为你的 SHA-1 值生成出简短且唯一的缩写。如果你传递 --abbrev-commit 给 git log 命令，输出结果里就会使用简短且唯一的值；它默认使用七个字符来表示，不过必要时为了避免 SHA-1 的歧义，会增加字符数：
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit

衍合与挑拣(cherry-pick)的流程

为了保持线性的提交历史，除了使用衍合的方式还可以通过挑拣的方式，挑拣类似于针对某次特定提交的衍合，他首先提取某次提交的补丁，然后试着应用在当前分支上。如果某个特性分支上有多个 commits，但你只想引入其中之一就可以使用这种方法。以下图为例：

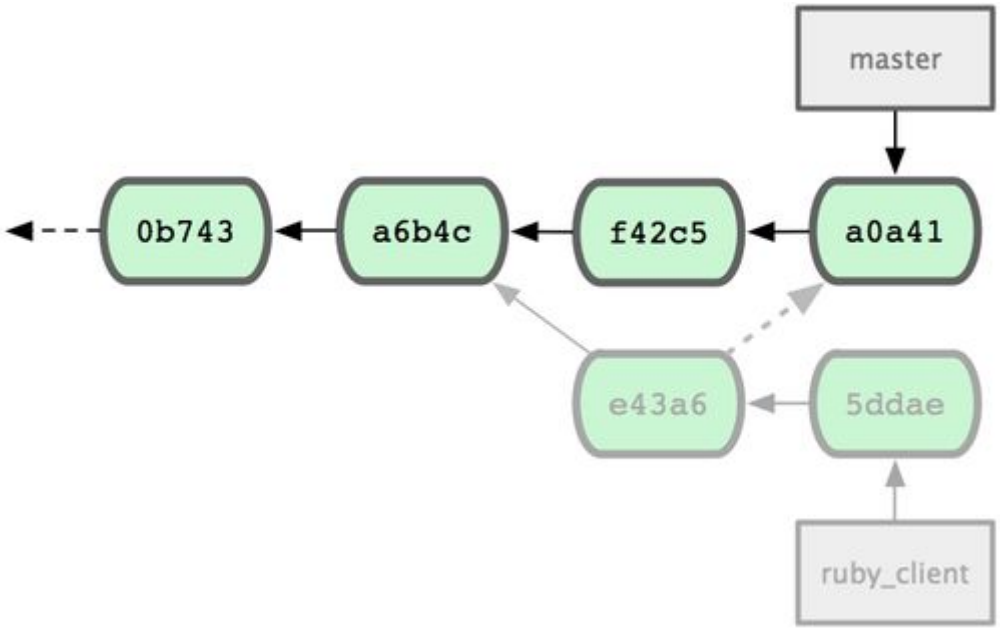


如果你希望拉取 e43a6 到你的主干分支，可以这样：

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
```

```
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

这将会引入 e43a6 的代码，但是会得到不同的 SHA-1 值，因为应用日期不同。现在你的历史看起来如下图所示：



现在，你可以删除这个特性分支并丢弃你不想引入的那些 commit。

制作简报

使用 git shortlog 命令可以方便快捷的制作一份修改日志（changelog），告诉大家上次发布之后又增加了哪些特性和修复了哪些 bug。实际上这个命令能够

统计给定范围内的所有提交;假如你上一次发布的版本是 v1.0.1，下面的命令将给出自从上次发布之后的所有提交的简介：

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

这就是自从 v1.0.1 版本以来的所有提交的简介，内容按照作者分组，以便你能快速的发 e-mail 给他们。

分支引用

指明一次提交的最直接的方法要求有一个指向它的分支引用。这样，你就可以在任何需要一个提交对象或者 SHA-1 值的 Git 命令中使用该分支名称了。如果你想要显示一个分支的最后一次提交的对象，例如假设 topic1 分支指向 ca82a6d，那么下面的命令是等价的：

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

如果你想知道某个分支指向哪个特定的 SHA，或者想看任何一个例子中被简写的 SHA-1，你可以使用一个叫做 rev-parse 的 Git 探测工具。有时你想看 Git 现在到底处于什么状态时，它可能会很有用。这里你可以对你的分支运执行 rev-parse。

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

引用日志里的简称

在你工作的同时，Git 在后台的工作之一就是保存一份引用日志——一份记录最近几个月你的 HEAD 和分支引用的日志。
你可以使用 git reflog 来查看引用日志：

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

每次你的分支顶端因为某些原因被修改时，Git 就会为你将信息保存在这个临时历史记录里面。你也可以使用这份数据来指明更早的分支。如果你想查看仓库中 HEAD 在五次前的值，你可以使用引用日志的输出中的 @{n} 引用：

$ git show HEAD@{5}
你也可以使用这个语法来查看某个分支在一定时间前的位置。例如，想看你的 master 分支昨天在哪，你可以输入
$ git show master@{yesterday}
它就会显示昨天分支的顶端在哪。这项技术只对还在你引用日志里的数据有用，所以不能用来查看比几个月前还早的提交。
需要注意的是，引用日志信息只存在于本地——这是一个记录你在你自己的仓库里做过什么的日志。其他人拷贝的仓库里的引用日志不会和你的相同；而你新克隆一个仓库的时候，引用日志是空的，因为你在仓库里还没有操作。git show HEAD@{2.months.ago} 这条命令只有在你克隆了一个项目至少两个月时才会有用——如果你是五分钟前克隆的仓库，那么它将不会有结果返回。

祖先引用

另一种指明某次提交的常用方法是通过它的祖先。如果你在引用最后加上一个 ^，Git 将其理解为此次提交的父提交。 假设你的工程历史是这样的：

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

那么，想看上一次提交，你可以使用 HEAD^，意思是“HEAD 的父提交
你也可以在 ^ 后添加一个数字——例如，d921970^2 意思是“d921970 的第二父提交”。这种语法只在合并提交时有用，因为合并提交可能有多个父提交。第一父提交是你合并时所在分支，而第二父提交是你所合并的分支：

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

另外一个指明祖先提交的方法是 ~。这也是指向第一父提交，所以 HEAD~ 和 HEAD^ 是等价的。当你指定数字的时候就明显不一样了。HEAD~2 是指“第一父提交的第一父提交”，也就是“祖父提交”——它会根据你指定的次数检索第一父提交。例如，在上面列出的历史记录里面，HEAD~3 会是

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

也可以写成 HEAD^^^，同样是第一父提交的第一父提交的第一父提交：

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

你也可以混合使用这些语法——你可以通过 HEAD~3^2 指明先前引用的第二父提交（假设它是一个合并提交）。

提交范围

现在你已经可以指明单次的提交，让我们来看看怎样指明一定范围的提交。这在你管理分支的时候尤显重要——如果你有很多分支，你可以指明范围来圈定一些问题的答案，比如：“这个分支上我有哪些工作还没合并到主分支的？”

双点

最常用的指明范围的方法是双点的语法。这种语法主要是让 Git 区分出可从一个分支中获得而不能从另一个分支中获得的提交。例如，假设你有类似于图 6-1 的提交历史。

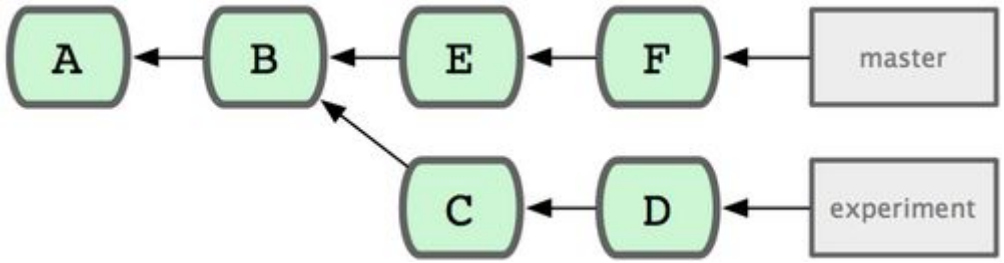


图 6-1. 范围选择的提交历史实例

你想要查看你的试验分支上哪些没有被提交到主分支，那么你就可以使用 master..experiment 来让 Git 显示这些提交的日志——这句话的意思是“所有可从 experiment 分支中获得而不能从 master 分支中获得的提交”。为了使例子简单明了，我使用了图标中提交对象的字母来代替真实日志的输出，所以会显示：

```
$ git log master..experiment
D
C
```

另一方面，如果你想看相反的——所有在 master 而不在 experiment 中的分支——你可以交换分支的名字。experiment..master 显示所有可在 master 获得而在 experiment 中不能的提交：

```
$ git log experiment..master
F
E
```

这在你想保持 experiment 分支最新和预览你将合并的提交的时候特别有用。这个语法的另一种常见用途是查看你将把什么推送到远程：

```
$ git log origin/master..HEAD
```

这条命令显示任何在你当前分支上而不在远程 origin 上的提交。如果你运行 git push 并且的你的当前分支正在跟踪 origin/master，被 git log

origin/master..HEAD 列出的提交就是将被传输到服务器上的提交。 你也可以留空语法中的一边来让 Git 来假定它是 HEAD。例如，输入 git log origin/master.. 将得到和上面的例子一样的结果—— Git 使用 HEAD 来代替不存在的一边。

多点

双点语法就像速记一样有用；但是你也许会想针对两个以上的分支来指明修订版本，比如查看哪些提交被包含在某些分支中的一个，但是不在你当前的分支上。Git 允许你在引用前使用^字符或者--not 指明你不希望提交被包含其中的分支。因此下面三个命令是等同的：
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
这样很好，因为它允许你在查询中指定多于两个的引用，而这是双点语法所做不到的。例如，如果你想查找所有从 refA 或 refB 包含的但是不被 refC 包含的提交，你可以输入下面中的一个
$ git log refA refB ^refC
$ git log refA refB --not refC
这建立了一个非常强大的修订版本查询系统，应该可以帮助你解决分支里包含了什么这个问题。

三点

最后一种主要的范围选择语法是三点语法，这个可以指定被两个引用中的一个包含但又不被两者同时包含的分支。回过头来看一下图 6-1 里所列的提交历史的例子。 如果你想查看 master 或者 experiment 中包含的但不是两者共有的引用，你可以运行

```
$ git log master...experiment
F
E
```

```
D
C
```

这个再次给出你普通的 log 输出但是只显示那四次提交的信息，按照传统的提交日期排列。
这种情形下，log 命令的一个常用参数是--left-right，它会显示每个提交到底处于哪一侧的分支。这使得数据更加有用。

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

交互式暂存

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

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

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

你会看到这个命令以一个完全不同的视图显示了你的暂存区——主要是你通过 git status 得到的那些信息但是稍微简洁但信息更加丰富一些。它在左侧列出了你暂存的变更，在右侧列出了未被暂存的变更。
在这之后是一个命令区。这里你可以做很多事情，包括暂存文件，撤回文件，暂存部分文件，加入未被追踪的文件，查看暂存文件的差别。

暂存补丁

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

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

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

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

此处你有很多选择。输入?可以显示列表：

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

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

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
```

```
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

simplegit.rb 的状态非常有意思。它显示有几行被暂存了，有几行没有。你部分地暂存了这个文件。在这时，你可以退出交互式脚本然后运行 git commit 来提交部分暂存的文件。
最后你也可以不通过交互式增加的模式来实现部分文件暂存——你可以在命令行下通过 git add -p 或者 git add --patch 来启动同样的脚本。

储藏（Stashing）

经常有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是 git stash 命令。
“‘储藏”“可以获取你工作目录的中间状态——也就是你修改过的被追踪的文件和暂存的变更——并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

储藏你的工作

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

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     modified:   index.html
#
# Changes not staged for commit:
```

```
#   (use "git add <file>..." to update what will be committed)
#
#     modified:   lib/simplegit.rb
#
```

现在你想切换分支，但是你还不想提交你正在进行中的工作；所以你储藏这些变更。为了往堆栈推送一个新的储藏，只要运行 git stash：

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

你的工作目录就干净了：

```
$ git status
# On branch master
nothing to commit, working directory clean
```

这时，你可以方便地切换到其他分支工作；你的变更都保存在栈上。要查看现有的储藏，你可以使用 git stash list：

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

在这个案例中，之前已经进行了两次储藏，所以你可以访问到三个不同的储藏。你可以重新应用你刚刚实施的储藏，所采用的命令就是之前在原始的 stash

命令的帮助输出里提示的：git stash apply。如果你想应用更早的储藏，你可以通过名字指定它，像这样：git stash apply stash@{2}。如果你不指明，Git 默认使用最近的储藏并尝试应用它：

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

你可以看到 Git 重新修改了你所储藏的那些当时尚未提交的文件。在这个案例里，你尝试应用储藏的工作目录是干净的，并且属于同一分支；但是一个干净的工作目录和应用到相同的分支上并不是应用储藏的必要条件。你可以在其中一个分支上保留一份储藏，随后切换到另外一个分支，再重新应用这些变更。在工作目录里包含已修改和未提交的文件时，你也可以应用储藏——Git 会给出归并冲突如果有任何变更无法干净地被应用。
对文件的变更被重新应用，但是被暂存的文件没有重新被暂存。想那样的话，你必须在运行 git stash apply 命令时带上一个 --index 的选项来告诉命令重新应用被暂存的变更。如果你是这么做的，你应该已经回到你原来的位置：

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
```

```
#
#      modified:   lib/simplegit.rb
#
```

apply 选项只尝试应用储藏的工作——储藏的内容仍然在栈上。要移除它，你可以运行 git stash drop，加上你希望移除的储藏的名字：

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

你也可以运行 git stash pop 来重新应用储藏，同时立刻将其从堆栈中移走。

取消储藏(Un-applying a Stash)

在某些情况下，你可能想应用储藏的修改，在进行了一些其他的修改后，又要取消之前所应用储藏的修改。Git 没有提供类似于 stash unapply 的命令，但是可以通过取消该储藏的补丁达到同样的效果：

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

同样的，如果你没有指定具体的某个储藏，Git 会选择最近的储藏：

$ git stash show -p | git apply -R

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

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
```

```
$ #... work work work
$ git stash-unapply
```

从储藏中创建分支

如果你储藏了一些工作，暂时不去理会，然后继续在你储藏工作的分支上工作，你在重新应用工作时可能会碰到一些问题。如果尝试应用的变更是针对一个你那之后修改过的文件，你会碰到一个归并冲突并且必须去化解它。如果你想用更方便的方法来重新检验你储藏的变更，你可以运行 git stash branch，这会创建一个新的分支，检出你储藏工作时的所处的提交，重新应用你的工作，如果成功，将会丢弃储藏。

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#     modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

这是一个很棒的捷径来恢复储藏的工作然后在新的分支上继续当时的工作。
Git 的内部原理参照这里来理解：

http://git-scm.com/book/zh/v1/Git-%E5%86%85%E9%83%A8%E5%8E%9F%E7%90%86-Git-%E5%AF%B9%E8%B1%A1

数据恢复

在使用 Git 的过程中，有时会不小心丢失 commit 信息。这一般出现在以下情况下：强制删除了一个分支而后又想重新使用这个分支，hard-reset 了一个分支从而丢弃了分支的部分 commit。如果这真的发生了，有什么办法把丢失的 commit 找回来呢？
下面的示例演示了对 test 仓库主分支进行 hard-reset 到一个老版本的 commit 的操作，然后恢复丢失的 commit 。首先查看一下当前的仓库状态：

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

接着将 master 分支移回至中间的一个 commit：

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

这样就丢弃了最新的两个 commit —— 包含这两个 commit 的分支不存在了。现在要做的是找出最新的那个 commit 的 SHA，然后添加一个指它它的分支。关键在于找出最新的 commit 的 SHA —— 你不大可能记住了这个 SHA，是吧？

通常最快捷的办法是使用 git reflog 工具。当你 (在一个仓库下) 工作时，Git 会在你每次修改了 HEAD 时悄悄地将改动记录下来。当你提交或修改分支时，reflog 就会更新。git update-ref 命令也可以更新 reflog，这是在本章前面的 "Git References" 部分我们使用该命令而不是手工将 SHA 值写入 ref 文件的理由。任何时间运行 git reflog 命令可以查看当前的状态：

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

可以看到我们签出的两个 commit ，但没有更多的相关信息。运行 git log -g 会输出 reflog 的正常日志，从而显示更多有用信息：

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    modified repo a bit
```

看起来弄丢了的 commit 是底下那个，这样在那个 commit 上创建一个新分支就能把它恢复过来。比方说，可以在那个 commit (ab1afef) 上创建一个名为 recover-branch 的分支：

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

酷！这样有了一个跟原来 master 一样的 recover-branch 分支，最新的两个 commit 又找回来了。接着，假设引起 commit 丢失的原因并没有记录在 reflog 中 —— 可以通过删除 recover-branch 和 reflog 来模拟这种情况。这样最新的两个 commit 不会被任何东西引用到：

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

因为 reflog 数据是保存在 .git/logs/ 目录下的，这样就没有 reflog 了。现在要怎样恢复 commit 呢？办法之一是使用 git fsck 工具，该工具会检查仓库的数据完整性。如果指定 --full 选项，该命令显示所有未被其他对象引用 (指向) 的所有对象：

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

本例中，可以从 dangling commit 找到丢失了的 commit。用相同的方法就可以恢复它，即创建一个指向该 SHA 的分支。

Git diff

不加参数的 git diff 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，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。
git diff --cached 命令查看已经暂存起来的文件和上次提交时的快照之间的差异，高级版本也可以叫 git diff --staged

查看 master 分支和 dev 分支差异：
git diff dev
$ git diff HEAD
上面的命令显示你工作目录与上次提交时之间的所有差别，这条命令所显示的 内容都会在执行"git commit -a"命令时被提交。
如果你要查看当前的工作目录与另外一个分支的差别，你可以用下面的命令执行:
$ git diff test
这会显示你当前工作目录与另外一个叫'test'分支的差别。你也以加上路径限定符，来只 比较某一个文件或目录。
$ git diff HEAD -- ./lib
上面这条命令会显示你当前工作目录下的 lib 目录与上次提交之间的差别(或者更准确的 说是在当前分支)。
如果不是查看每个文件的详细差别，而是统计一下有哪些文件被改动，有多少行被改 动，就可以使用'--stat' 参数。
$>git diff --stat

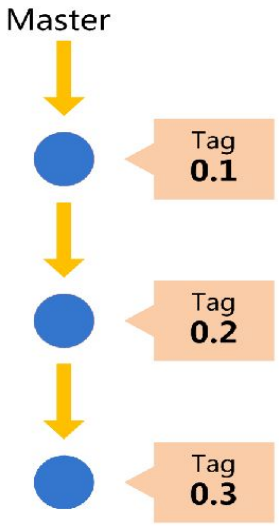
```
 layout/book_index_template.html                    |    8 ++-
 text/05_Installing_Git/0_Source.markdown           |   14 ++++++
 text/05_Installing_Git/1_Linux.markdown            |   17 +++++++
 text/05_Installing_Git/2_Mac_104.markdown          |   11 +++++
 text/05_Installing_Git/3_Mac_105.markdown          |    8 ++++
 text/05_Installing_Git/4_Windows.markdown          |    7 +++
 .../1_Getting_a_Git_Repo.markdown                  |    7 +++-
 .../0_ Comparing_Commits_Git_Diff.markdown         |   45 ++++++++++++++++++++-
```

.../0_ Hosting_Git_gitweb_repoorcz_github.markdown | 4 +-
9 files changed, 115 insertions(+), 6 deletions(-)

3. 分支管理

3.1 主分支 Master

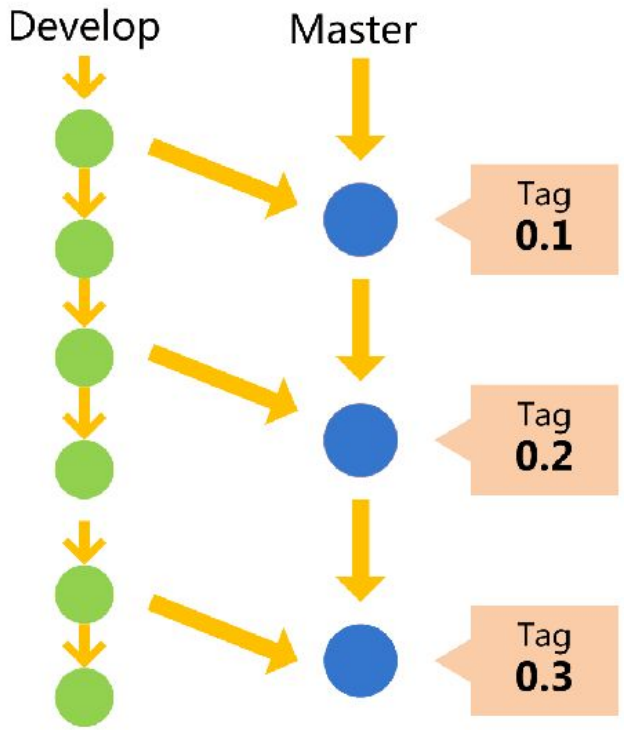
首先，代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。



Git 主分支的名字，默认叫做 Master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。

3.2 开发分支 Develop

主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做 Develop。



这个分支可以用来生成代码的最新隔夜版本（nightly）。如果想正式对外发布，就在 Master 分支上，对 Develop 分支进行"合并"（merge）。

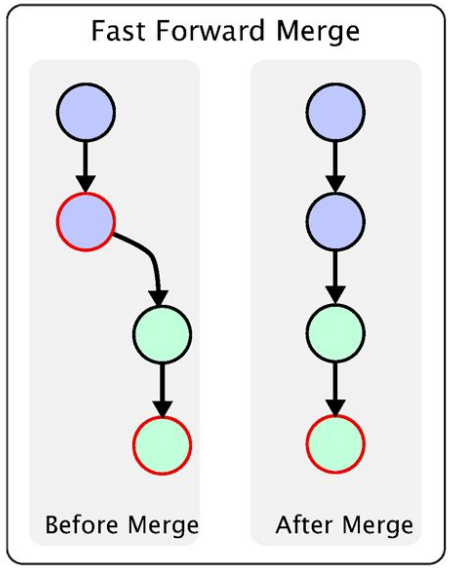
Git 创建 Develop 分支的命令：

```
git checkout -b develop master
```

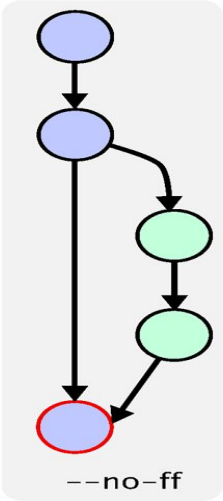
将 Develop 分支发布到 Master 分支的命令：

```
# 切换到 Master 分支
git checkout master
# 对 Develop 分支进行合并
git merge --no-ff develop
```

这里稍微解释一下，上一条命令的--no-ff 参数是什么意思。默认情况下，Git 执行"快进式合并"（fast-farward merge），会直接将 Master 分支指向 Develop 分支，你无法一眼从 Git 历史中看到哪些 commit 对象构成了一个特性——你需要阅读日志以获得该信息。在这种情况下，回退（revert）整个特性（一组 commit）就会比较麻烦



使用--no-ff 参数后，会执行正常合并，在 Master 分支上生成一个新节点，很好的避免丢失特性分支存在的历史信息，同时也能清晰的展现一组 commit 一起构成一个特性 。为了保证版本演进的清晰，我们希望采用这种做法。



3.3 临时性分支

前面讲到版本库的两条主要分支：Master 和 Develop。前者用于正式发布，后者用于日常开发。其实，常设分支只需要这两条就够了，不需要其他了。

但是，除了常设分支以外，还有一些临时性分支，用于应对一些特定目的的版本开发。临时性分支主要有三种：

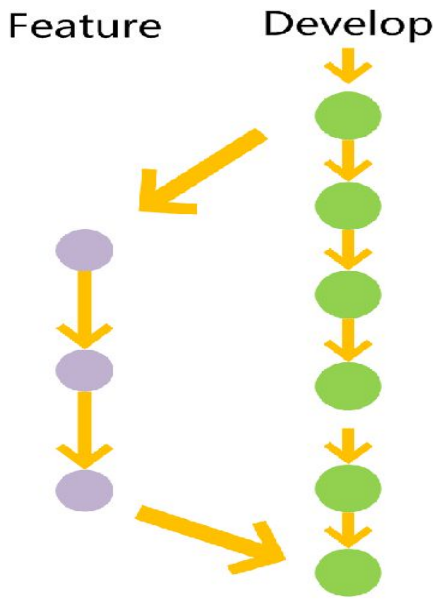
* 功能（feature）分支
* 预发布（release）分支
* 修补 bug（fixbug）分支

这三种分支都属于临时性需要，使用完以后，应该删除，使得代码库的常设分支始终只有 Master 和 Develop。

3.4 功能分支

接下来，一个个来看这三种"临时性分支"。

第一种是功能分支，它是为了开发某种特定功能，从 Develop 分支上面分出来的。开发完成后，要再并入 Develop。



功能分支的名字，可以采用 feature-*的形式命名。

创建一个功能分支：

```
git checkout -b feature-x develop
```

开发完成后，将功能分支合并到 develop 分支：

```
git checkout develop
```

```
git merge --no-ff feature-x
```

删除 feature 分支：

```
git branch -d feature-x
```

3.5 预发布分支

第二种是预发布分支，它是指发布正式版本之前（即合并到 Master 分支之前），我们可能需要有一个预发布的版本进行测试。

预发布分支是从 Develop 分支上面分出来的，预发布结束以后，必须合并进 Develop 和 Master 分支。它的命名，可以采用 release-*的形式。

创建一个预发布分支：

```
git checkout -b release-1.2 develop
```

确认没有问题后，合并到 master 分支：

```
git checkout master
git merge --no-ff release-1.2
# 对合并生成的新节点，做一个标签
git tag -a 1.2
```

再合并到 develop 分支：

```
git checkout develop
git merge --no-ff release-1.2
```

最后，删除预发布分支：

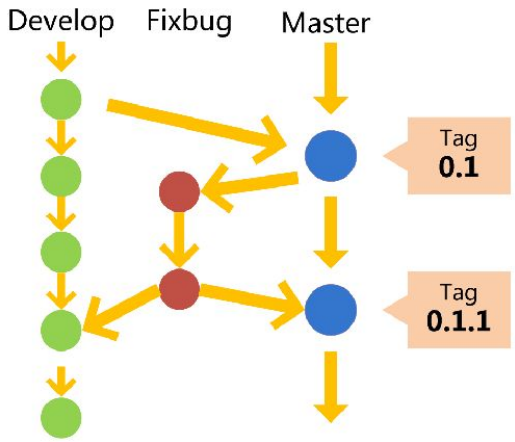
```
git branch -d release-1.2
```

3.6 修补 bug 分支

最后一种是修补 bug 分支。软件正式发布以后，难免会出现 bug。这时就需要创建一个分支，进行 bug 修补。

修补 bug 分支是从 Master 分支上面分出来的。修补结束以后，再合并进 Master 和 Develop 分支。这里还有个例外情况，如果这个时候有发布分支存在，

热补丁分支的变更则应该合并至发布分支，而不是 develop。将热补丁合并到发布分支，也意味着当发布分支结束的时候，变更最终会被合并到 develop。它的命名可以采用 fixbug-*的形式。



创建一个修补 bug 分支：

```
git checkout -b fixbug-0.1 master
```

修补结束后，合并到 master 分支：

```
git checkout master
git merge --no-ff fixbug-0.1
git tag -a 0.1.1
```

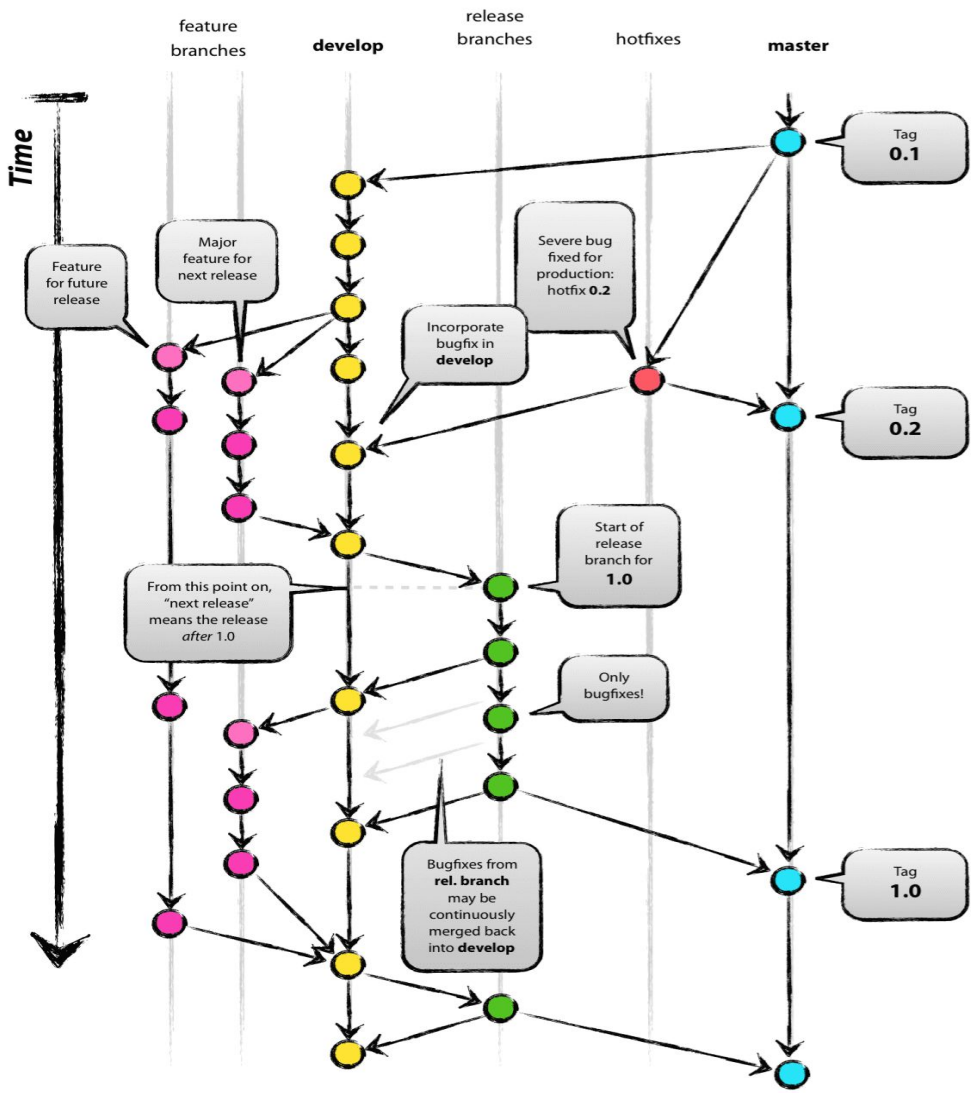
再合并到 develop 分支：

```
git checkout develop
git merge --no-ff fixbug-0.1
```

最后，删除"修补 bug 分支"：

```
git branch -d fixbug-0.1
```

最后上一张完整的分支模型图


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0