20190225:0906

20190224:1822 P51

20190223:1322 P37

20190222:1404 P33

:

# GIT起步

## Git初次配置

每台计算机上只需要配置一次，程序升级时会保留配置信息。你可以在任何时候再次通过运行命令来修改它们。Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。这些变量存储在三个不同的位置：

1. /etc/gitconfig文件: 包含系统上每一个用户及他们仓库的通用配置。如果使用带有 --system 选项的 git config 时，它会从此文件读写配置变量。

**git config --system**

1. ~/.gitconfig或 ~/.config/git/config 文件：只针对当前用户。可以传递 --global选项让 Git 读写此文件。

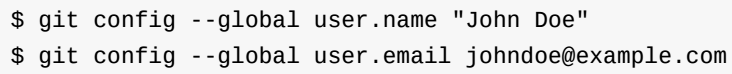
**git config -- global**

1. 当前使用仓库的 Git 目录中的 config 文件（就是 .git/config ）：针对该仓库。**git config**

每一个级别覆盖上一级别的配置，所以 .git/config 的配置变量会覆盖 /etc/gitconfig中的配置变量。即级别低的配置，优先级高。

### 用户信息配置

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：



### 文本编辑器

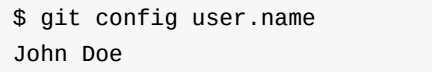
既然用户信息已经设置完毕，你可以配置默认文本编辑器了，



### 检查配置信息

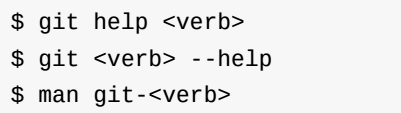
如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置。

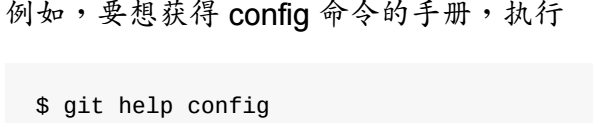


你可能会看到重复的变量名，因为 Git 会从不同的文件中读取同一个配置（例如： /etc/gitconfig与 ~/.gitconfig）。这种情况下，Git 会使用它找到的每一个变量的最后一个配置。你可以通过输入 git config <key>：来检查 Git 的某一项配置

### 获取帮助

**若你使用 Git 时需要获取帮助，有三种方法可以找到 Git 命令的使用手册：**





常用方法有二：

**$git help config**

**$git config --help**

# GIT基础

## 创建GIT仓库（Repository）

有两种取得 Git 项目仓库的方法。第一种是在现有项目或目录下导入所有文件到 Git 中；第二种是从一个服务器克隆一个现有的 Git 仓库。

### 在现有目录中初始化仓库

如果你打算使用 Git 来对现有的项目进行管理，你只需要进入该项目目录并输入：

**git init**

该命令将创建一个名为 .git 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。但是，在这个时候，我们仅仅是做了一个初始化的操作，你的项目里的文件还没有被跟踪。

### 克隆现有的仓库

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，你想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令。如果你对其它的 VCS 系统（比如说Subversion）很熟悉，请留心一下你所使用的命令是"clone"而不是"checkout"。这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。事实上，如果你的服务器的磁盘坏掉了，你通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库

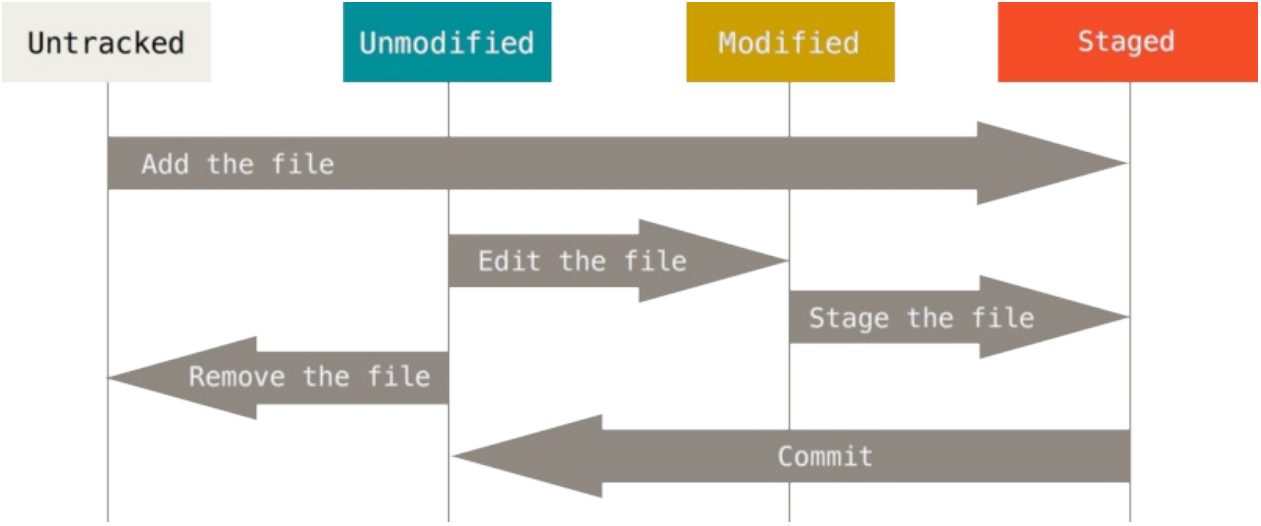
**git clone [url]**



这会在当前目录下创建一个名为 `libgit2'' 的目录，并在这个目录下初始化一个 `.git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。如果你进入到这个新建的 libgit2 文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以使用如下命令：

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为mylibgit。

## 记录每次更新到仓库

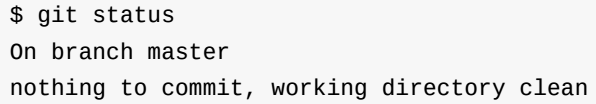
工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：

### 检查当前文件状态

要查看哪些文件处于什么状态，可以用 git status 命令。仅针对当前分支。

**git status**

如果在克隆仓库后立即使用此命令，会看到类似这样的输出：



所有已跟踪文件在上次提交后都未被更改过。此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git会在这里列出来。最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上记录每次更新到仓库32对应的分支没有偏离。现在，分支名是 ``master'',这是默认的分支名。

### 跟踪新文件

使用命令 git add 开始跟踪一个文件。所以，要跟踪 README 文件，运行：

git add 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。**通过该命令可将目标加入缓冲区。**

使用git rm --cached <file>可删除缓冲器（stagedArea）中的指定内容。

### 暂存已修改文件

利用下述命令，可以实现对暂存区的操作：

$ git add file // 将文件加入暂存区

$ git status // 获取暂存区的状态

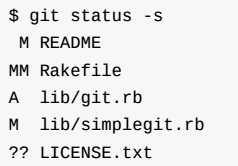
$ git rm –cached file // 将暂存区中的制定文件删除

$ git checkout file // 从暂存区中获取指定文件。

### 状态简览

关于git status 命令。

Git status -s / --short // 获取状态简述信息。



未跟踪文件前面有 ?? 标记，

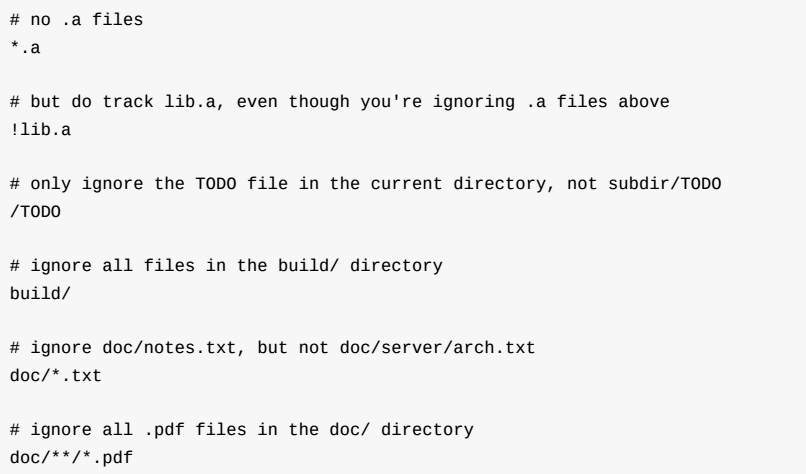
新添加到暂存区中的文件前面有 A 标记

修改过的文件前面有 M 标记。

### 忽略文件(.gitignore)

要养成一开始就设置好.gitignore文件的习惯，以免将来误提交这类无用的文件。文件 .gitignore的格式规范如下：

1. 所有空行或者以＃开头的行都会被 Git 忽略。
2. 可以使用标准的 glob 模式匹配。
3. 匹配模式可以以（ / ）开头防止递归。
4. 匹配模式可以以（ / ）结尾指定目录。
5. 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。



说明如下：

1. \*.a // 忽略工程目录中的所有.a文件
2. ！lib.a // 忽略将lib.a除外
3. /TODO // 忽略当前目录下的TODO文件
4. build/ // 忽略build目录下的所有文件
5. doc/\*.txt // 忽略doc目录下的所有txt文件
6. doc/**\*\***/\*.pdf // 忽略doc目录下的所有pdf文件，含子目录

GitHub 有一个十分详细的针对数十种项目及语言的 .gitignore文件列表，你可以在https://github.com/github/gitignore找到它.

* glob 模式

所谓的glob 模式是指shell 所使用的简化了的正则表达式。

1. 星号（\*）匹配零个或多个任意字符；
2. [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）；
3. 问号（ ? ）只匹配一个任意字符；
4. 如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（比如 [0-9] 表示匹配所有 0 到 9 的数字）;
5. 使用两个星号（\*\*) 表示匹配任意中间目录，比如 a/\*\*/z 可以匹配 a/z , a/b/z 或a/b/c/z 等。

### 查看已暂存和未暂存的修改

你想知道具体修改了什么地方，可以用git diff 命令

**git diff**

查看尚未暂存的文件更新了哪些部分,此命令比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

* git diff

工作目录中的改动与暂存（高优先）/提交的内容做比较。

只显示尚未暂存的改动。

已跟踪内容与已暂存内容做比较。

* git diff --cached/--staged

暂存内容与提交内容做比较。

查看已暂存的将要添加到下次提交里的内容。

已暂存内容与已提交内容做比较。

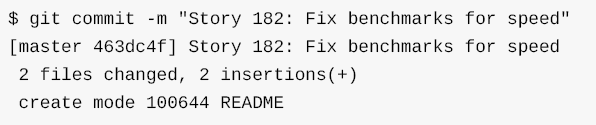
### 提交更新

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



这种方式会启动文本编辑器以便输入本次提交的说明。

你也可以在commit 命令后添加-m 选项，将提交信息与命令放在同一行



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。

* Git commit

如此提交，将会自动调出VIM，让你输入提交说明

* Git commit -m “本次提交说明”

如此将会直接提交。

### 跳过使用暂存区域

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

* Git commit -a -m “Depiction about this commit”

对于首次提交的文件，不能如此操作。

首次提交，必须先暂存（add）

### 移除文件

Git rm xxx.xx//从库中删除指定文件，也从当前目录中删除指定文件。

Gitrm --cached xxx.xx//从暂存区/库中删除，不影响当前目录文件，即停止对指定文件的跟踪。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。

Gitrm -fxxx.xx //从暂存区中强制删除。

$ 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。此命令删除log/ 目录下扩展名为.log 的所有文件，包括库中和缓冲中的，以及当前目录中的。

$ git rm \\*~

该命令为删除以~ 结尾的所有文件。包括当前目录中的，库中的，缓冲中的。

### 移动文件

实际是将库中的指定文件重命名。

$ git mv file\_fromfile\_to

此操作同样影响三个地方：当前目录/暂存区/库。

其实，运行git mv 就相当于运行了下面三条命令：

$ mv README.md README

$ git rm README.md

$ git add README

### 查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史. 完成这个任务最简单而又有效的工具是git log 命令

git log

默认不用任何参数的话，git log 会按提交时间列出所有的更新，最近的更新排在最上面。正如你所看到的，这个命令会列出每个提交的SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

Git log //显示每次提交的提交信息

Git log -p //用来显示每次提交的内容差异

Git log -p -n //用来显示最近n次提交的内容差异

Gitlog –stat //显示每次提交的提交信息，并显示每次提交的统计信息

列出额所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。在每次提交的最后还有一个总结。

Git log --pretty = //格式换log输出信息。

Git log --pretty=oneline // 在一行显示提交信息

Git log --pretty=short //

Git log --pretty=full //

Git log --pretty=fuller //

Git log --pretty=format:”%h %s” --graph

* 限制输出长度

在git中，当看到END时，输入q，即可推出到git中。

### 撤消操作

可以更改提交说明

可以再次覆盖提交。

在任何一个阶段，你都有可能想要撤消某些操作。

git commit --amend // 更改提交信息。

这个命令会将暂存区中的文件提交。如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。编辑后保存会覆盖原来的提交信息。

提交后发现忘记了暂存某些需要的修改，可以像下面这样操作

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

最终你只会有一个提交- 第二次提交将代替第一次提交的结果。

### 取消暂存的文件

Gitreset filename // 取消指定文件的暂存。

### 撤消对文件的修改

Git checkout filename// 即用暂存区（优先使用）/提交区中的文件来覆盖替换当前已修改文件。

如果你仍然想保留对那个文件做出的修改，但是现在仍然需要撤消，我们将会在介绍保存进度与分支；这些通常是更好的做法。

## 远程仓库的使用

### 查看远程仓库

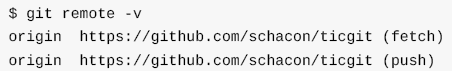
* git remote

查看你已经配置的远程仓库服务器,它会列出你指定的每一个远程服务器的简写。

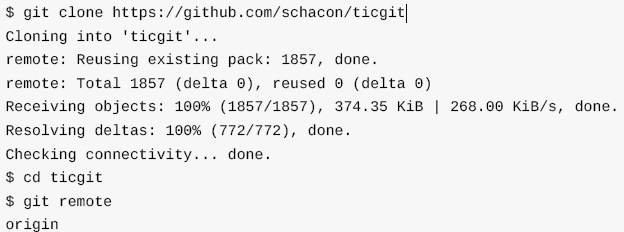


* git remote -v

会显示需要读写远程仓库使用的Git 保存的简写与其对应的URL



* Git clone URL



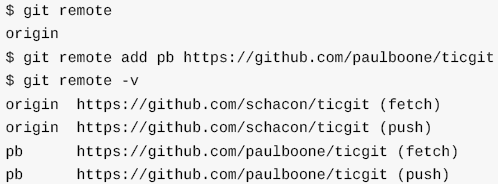
如果你已经克隆了自己的仓库，那么至少应该能看到origin，这是Git 给你克隆的仓库服务器仓库的默认名字：

### 添加远程仓库

* git remote add <shortname><url>

运行git remote add <shortname><url>添加一个新的远程Git 仓库，同时指定一个你可以轻松引用的简写：

为REPO提供一个别名。



现在你可以在命令行中使用字符串pb 来代替整个URL。例如，如果你想拉取Paul 的仓库中有但你没有的信息，可以运行git fetch pb ：

* git fetch <shortname>/<url> == git clone <shortname>/<url>

获取远程仓库

拉取仓库中有但你没有的信息，可以运行git fetch pb ：

### 从远程仓库中抓取与拉取

* git fetch [remote-name]

抓取克隆（或上一次抓取）后新推送的所有工作。

这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

如果你使用clone 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以`origin'' 为简写。所以，`git fetch origin 会抓取克隆（或上一次抓取）后新推送的所有工作。必须注意git fetch 命令会将数据拉取到你的本地仓库- 它并不会自动合并或修改你当前的工作。当准备好时你必须手动将其合并入你的工作。

如果你有一个分支设置为跟踪一个远程分支可以使用git pull 命令来自动的抓取然后合并远程分支到当前分支

默认情况下，git clone 命令会自动设置本地master分支跟踪克隆的远程仓库的master 分支。运行git pull通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

### 推送到远程仓库

* git push [remote-name][branch-name]

当你想要将master 分支推送到origin 服务器时（再次说明，克隆时通常会自动帮你设置好那两个名字），

Origin为被克隆的库，

Master为当前局部分支库。

那么运行这个命令就可以将你所做的备份到服务器：

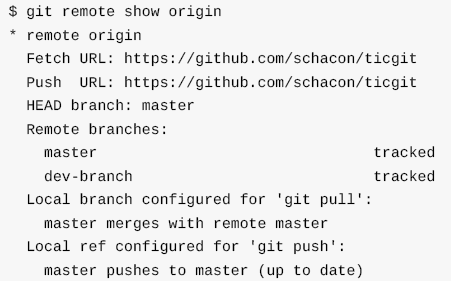
$ git push origin master

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。以此办证上推的有效性。

### 查看远程仓库

git remote show [remote-name]

如果想要查看某一个远程仓库的更多信息，以一个特定的缩写名运行这个命令，例如origin ，会得到像下面类似的信息



它同样会列出远程仓库的URL 与跟踪分支的信息。这些信息非常有用，它告诉你正处于master 分支，并且如果运行git pull，就会抓取所有的远程引用，然后将远程master 分支合并到本地master 分支。它也会列出拉取到的所有远程引用。

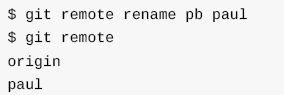
这个命令列出了当你在特定的分支上执行git push 会自动地推送到哪一个远程分支。它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当你执行git pull 时哪些分支会自动合并。

### 远程仓库的移除与重命名

* git remote rename

如果想要重命名引用的名字可以运行

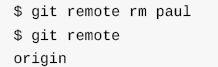
想要将pb 重命名为paul，可以用git remote rename 这样做：



这同样也会修改你的远程分支名字。那些过去引用pb/master 的现在会引用paul/master

* git remote rm

如果因为一些原因想要移除一个远程仓库- 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了- 可以使用



实际上是切断与指定库的联系。

## 打标签

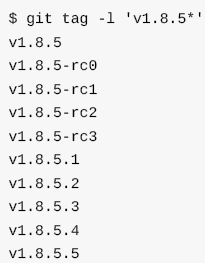
Git 可以给历史中的某一个提交打上标签，以示重要。 比较有代表性的是人们会使用这个功能来标记发布结点

### 列出标签

* git tag



这个命令以字母顺序列出标签；但是它们出现的顺序并不重要。



### 创建标签

Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）

一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。

附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU PrivacyGuard （GPG）签名与验证。

通常建议创建附注标签，这样你可以拥有以上所有信息；但是如果你只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。

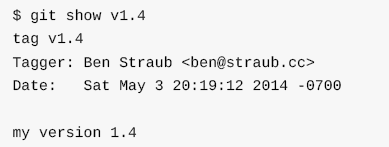
* 附注标签

在 Git 中创建一个附注标签是很简单的。 最简单的方式是当你在运行 tag 命令时指定 -a选项：



标签为后续引用的索引。

通过使用 git show 命令可以看到标签信息与对应的提交信息：



* 轻量标签

另一种给提交打标签的方式是使用轻量标签。 轻量标签本质上是将提交校验和存储到一个文件中 - 没有保存任何其他信息。 创建轻量标签，不需要使用 -a 、 -s 或 -m 选项，只需要提供标签名字：



* 后期打标签

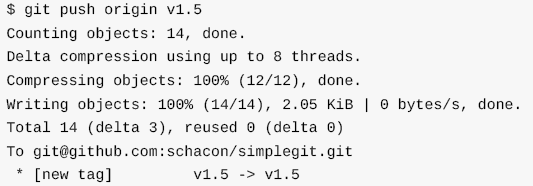
你可以在之后补上标签。 要在那个提交上打标签，你需要在命令的末尾指定提交的校验和（或部分校验和,**必须从头开始**）,:

git tag -a v1.2 –a ‘message’ 9fceb02

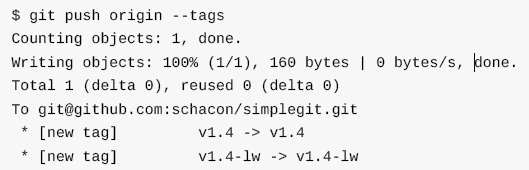
git tag v1.6 9gceb02

* 共享标签

默认情况下， git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上。 这个过程就像共享远程分支一样 - 你可以运行 git push origin [tagname]



如果想要一次性推送很多标签，也可以使用带有 --tags 选项的 git push 命令。 这将会把所有不在远程仓库服务器上的标签全部传送到那里。



现在，当其他人从仓库中克隆或拉取，他们也能得到你的那些标签。

* 检出标签

在 Git 中你并不能真的检出一个标签，因为它们并不能像分支一样来回移动。 如果你想要工作目录与仓库中特定的标签版本完全一样，可以使用 git checkout -b [branchname][tagname] 在特定的标签上创建一个新分支：

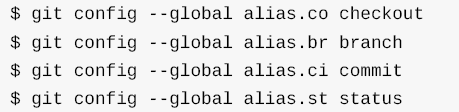


当然，如果在这之后又进行了一次提交， version2 分支会因为改动向前移动了，那么version2 分支就会和 v2.0.0 标签稍微有些不同，这时就应该当心了。

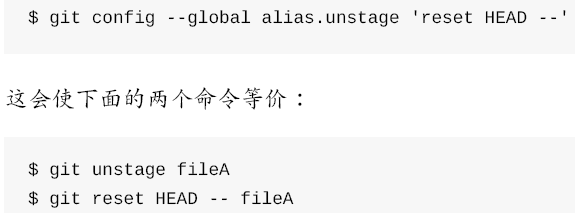
## 别名—ALIAS

### 创建别名

Git config --global alias.xx XXXXXX



Git config --global alias.xx ‘XXX XXX YYY’



这样看起来更清楚一些。 通常也会添加一个 last 命令，像这样：



### 创建外部命令别名

如果是那样的话，可以在命令前面加入 ! 符号。 如果你自己要写一些与 Git 仓库协作的工具的话，那会很有用。 我们现在演示将 git visual 定义为 gitk 的别名：



### 删除别名

# GIT分支

与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。理解和精通这一特性，你便会意识到 git 是如此的强大而又独特，并且从此真正改变你的开发方式。

## 分支简介

在进行提交操作时，git 会保存一个提交对象（commit object）--暂存内容快照。

在进行提交操作时，git会保存一个提交对象（commit object）。知道了git保存数据的方式，可以很自然的想到:该提交对象会包含一个指向暂存内容快照的指针。但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象，后续普通提交操作产生的提交对象含有一个父对象，而由多个分支合并产生的提交对象包含多个父对象。

假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。

暂存操作（git add）动作描述如下：

1. 为每一个文件计算校验和（SHA-1哈希算法），以作为后续访问的索引；
2. 把当前版本的文件快照保存到 Git 仓库中（Git使用 blob 对象来保存它们）；
3. 最终将校验和加入到暂存区域等待提交。



提交操作动作如下：

1. 先计算每一个子目录（本例中只有项目根目录）的校验和，
2. 在 Git 仓库中将这些校验和保存为树对象。
3. 创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。

一个目录一个树对象，目录中的子目录们，可象文件一样处理。

如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引），以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）

即如下：

库对象（文件快照，即文件备份）；

树对象（库对象索引及目录结构）；

提交对象（树对象及提交信息）；

具体结构如下所述：

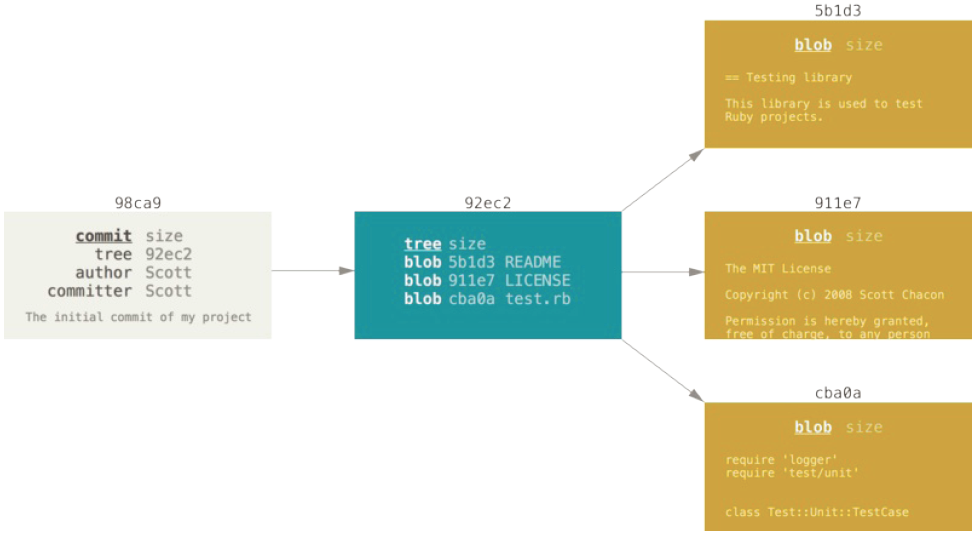
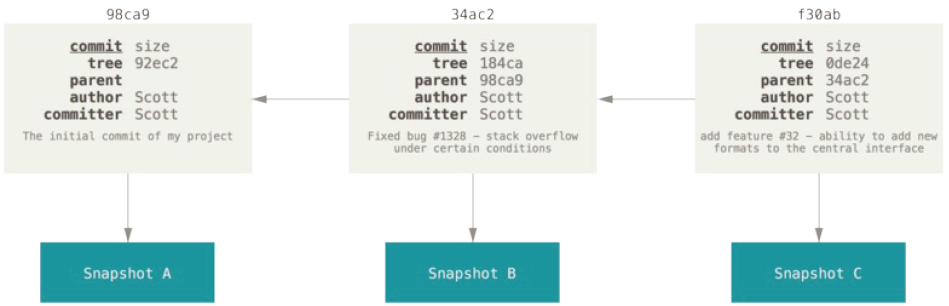


图1：首次提交对象及其树结构

即每次提交，都有一个“提交对象（Commit object）”与之对应。

可见，提交对象中索引，使用的都是相应对象的哈希值。

首次提交对象没有父对象，后续提交皆有父对象，通过提交对象链，即可掌握所有对象的过程状态。



一个提交对象（🡪树对象🡪库对象），即对应一次版本快照。

git的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master 。在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支（master始终自动指向最近的那个提交对象）。 它会在每次的提交操作中自动向前移动。

Git 的 `master'' 分支并不是一个特殊分支。 它就跟其它分支完全没有区别。 之所以几乎每一个仓库都有 master 分支，是因为 `git init 命令默认创建它，并且大多数人都懒得去改动它。

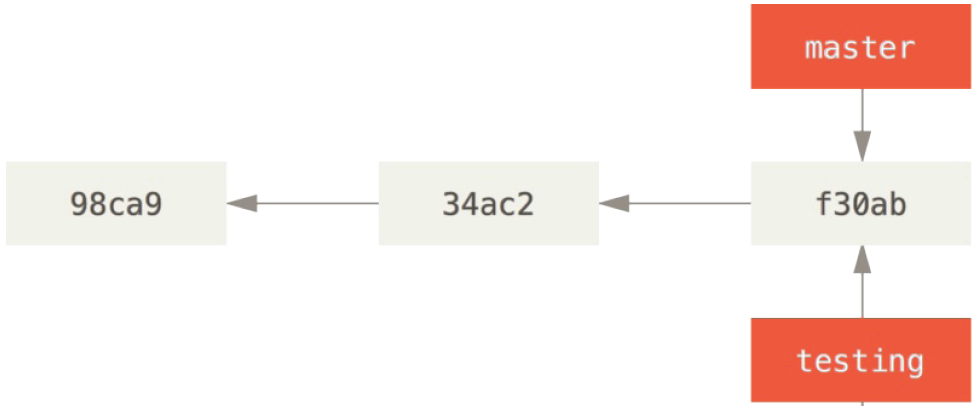
## 分支创建

**Git 的分支，其实本质上仅仅是指向提交对象的可变指针。**

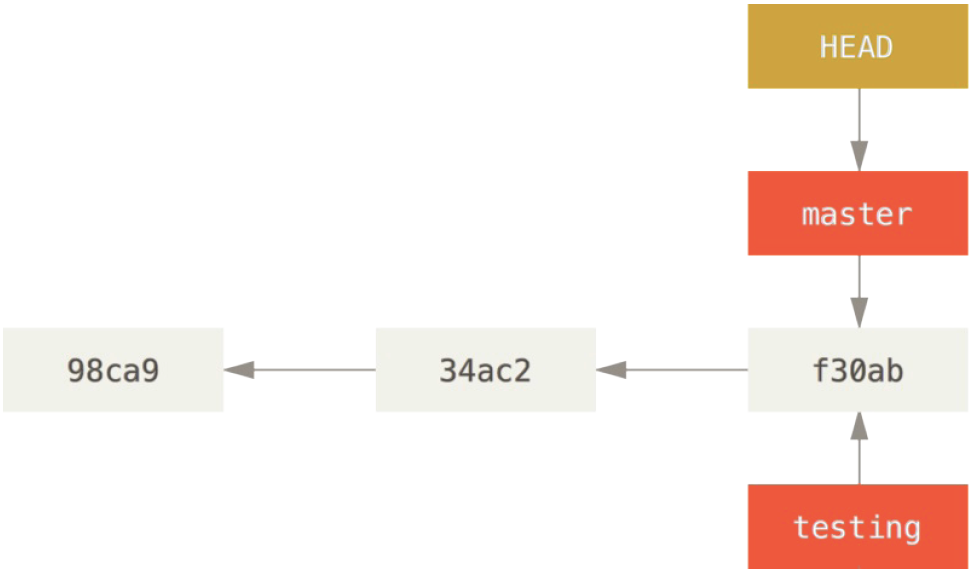
Git 是怎么创建新分支的呢？ 很简单，它只是为你创建了一个可以移动的新的指针。 比如，创建一个 testing 分支， 你需要使用 git branch 命令：



这会创建一个指向当前所在的提交对象指针。



有一个名为 HEAD 的特殊指针，她会知道Git 当前是在哪一个分支上。在 Git 中，它是一个指针，指向当前所在的本地分支，**将 HEAD 想象为当前分支的别名**）



因为 git branch 命令仅仅 创建 一个新分支，并不会自动切换到新分支中去。

你可以简单地使用 git log 命令查看各个分支当前所指的对象。

正如你所见，当前 master'' 和 testing'' 分支均指向校验和以 f30ab 开头的提交对象。

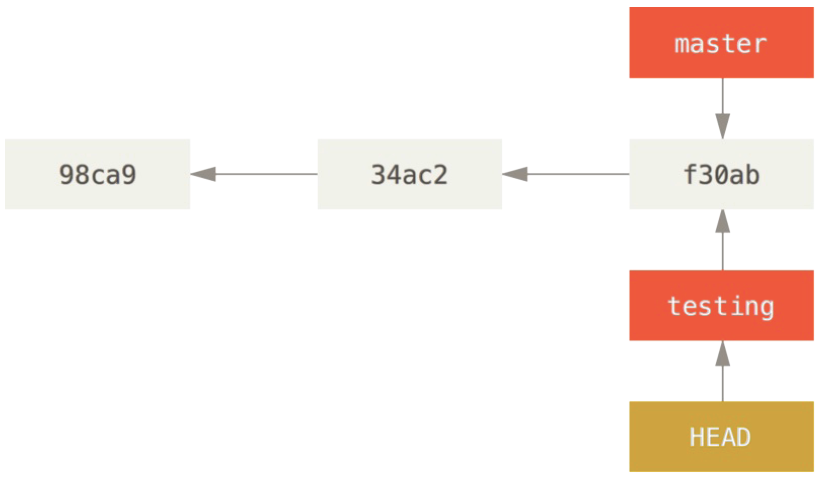
## 分支切换

要切换到一个已存在的分支，你需要使用 git checkout 命令

我们现在切换到新创建的testing 分支去：



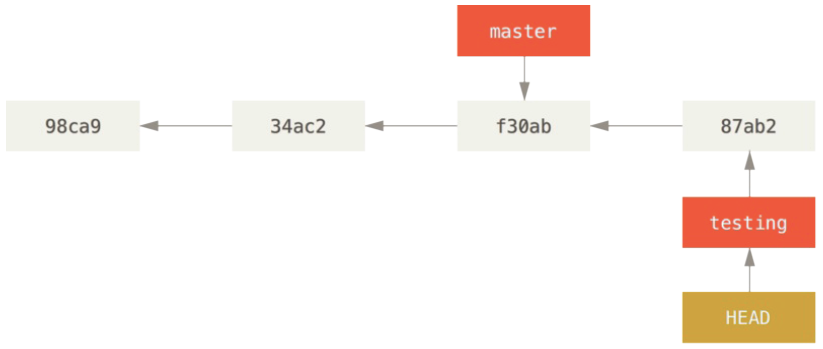
这样 HEAD 就指向 testing 分支了。



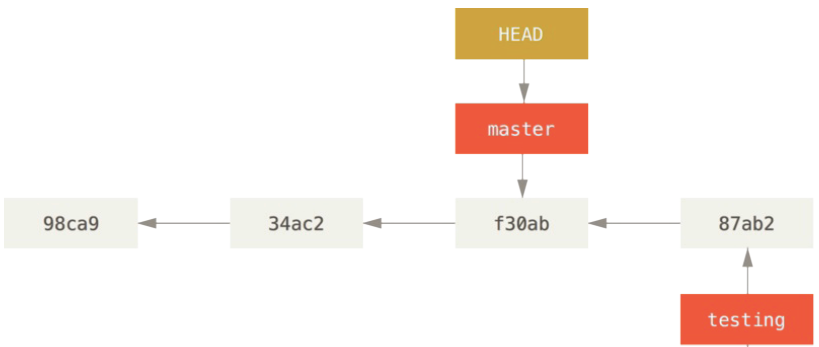
即

Master指向最新的分支；HEAD指向当前分支。

那么，这样的实现方式会给我们带来什么好处呢？ 现在不妨再提交一次：



如图所示，你的 testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。 这就有意思了，现在我们切换回 master 分支看看：

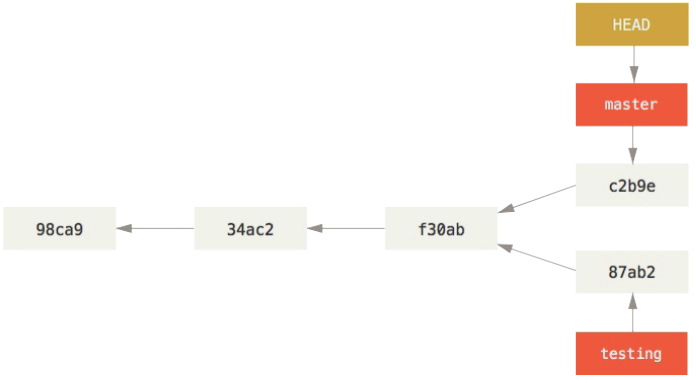


这条命令做了两件事。 一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。 也就是说，你现在做修改的话，项目将始于一个较旧的版本。 本质上来讲，这就是忽略 testing 分支所做的修改，以便于向另一个方向进行开发。

我们不妨再稍微做些修改并提交：

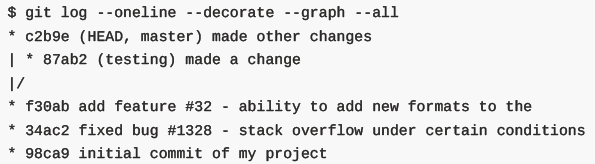


现在，这个项目的提交历史已经产生了分叉（参见 项目分叉历史）。 因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。 上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，**并在时机成熟时将它们合并起来**。 而所有这些工作，你需要的命令只有 branch 、 checkout 和 commit 。



项目分叉历史

你可以简单地使用 git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph--all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。



由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。

在 Git 中，任何规模的项目都能在瞬间创建新分支。 同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（译注：即共同祖先）也是同样的简单和高效。 这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支。

## 分支的新建与合并

让我们来看一个简单的分支创建与合并的例子，实际工作中，你会用到类似的工作流。

假设下述经历：

1. 开发某个网站。
2. 为实现某个新的需求，创建一个分支。
3. 在这个分支上开展工作。

正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。 你将按照如下方式来处理：

1. 切换到你的线上分支（production branch：线上分支，目前正在使用的那个）。
2. 为这个紧急任务新建一个分支（基于线上分支），并在其中修复它。
3. 在测试通过之后，切换回线上分支（production branch），然后合并这个修补分支，最后将改动推送到线上分支。
4. 切换回你最初工作的分支上，继续工作。

**是这样的！！！**

### 新建分支

$git branch iss53

$git checkout iss53

🡸🡺

$git checkout –b iss53

新建分支iss53，并切换过去。

HEAD为当前工作分支，

MASTER为最新分支。

在你这么做之前，要留意你的工作目录和暂存区里那些还没有被提交的修改，它可能会和你即将检出的分支产生冲突从而阻止 Git 切换到该分支。 最好的方法是，在你切换分支之前，保持好一个干净的状态。 有一些方法可以绕过这个问题

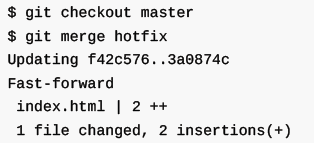
* 保存进度（stashing）
* 修补提交（commit amending）

当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。 Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样。

### 分支合并

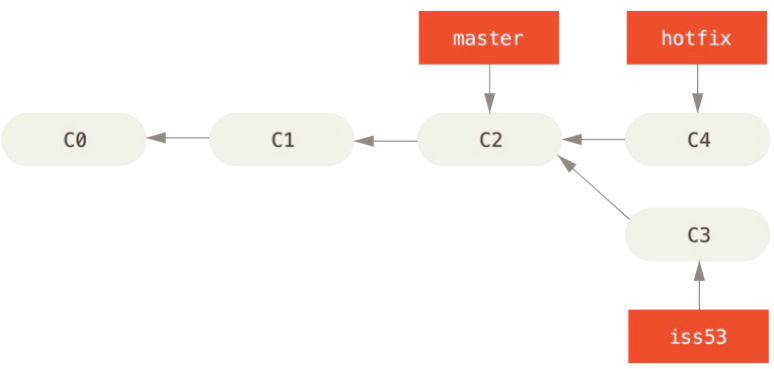
首先，使用checkout，切换至想要操作的分支，

然后，使用merge，将目标分支合并到当前分支中。

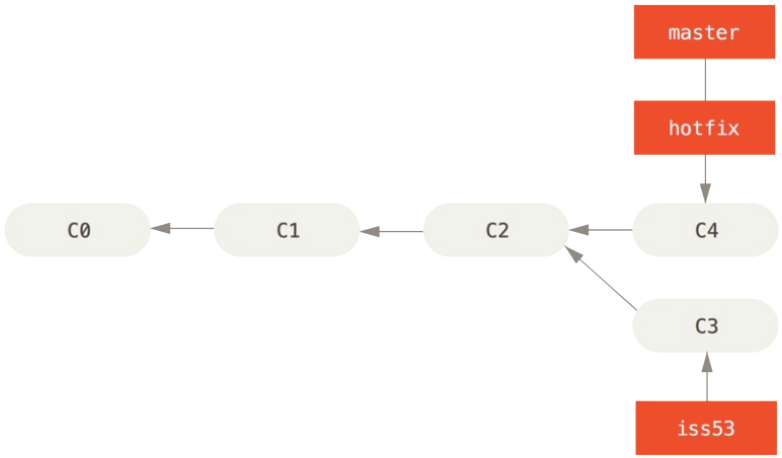


在合并的时候，你应该注意到了"快进（fast-forward）"这个词。 由于当前 master 分支所指向的提交是你当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动。 换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧 —— 这就叫做 “快进（fast-forward）”。

现在，最新的修改已经在 master 分支所指向的提交快照中，你可以着手发布该修复了。

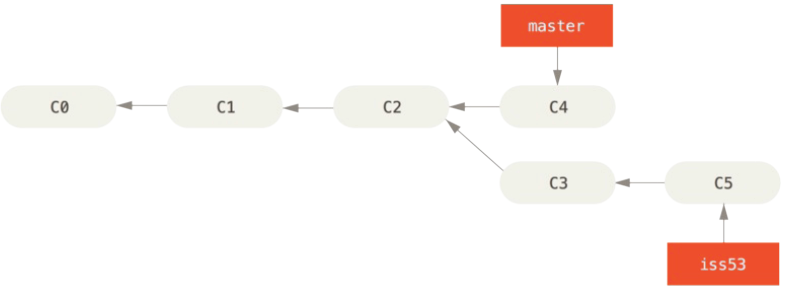


|  |  |  |
| --- | --- | --- |
| $ git checkout master  $ git merge hotfix | | |
| C:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.png | C:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.png | C:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.pngC:\Users\xxb\AppData\Local\Temp\SGPicFaceTpBq\1816\00612C20.png |

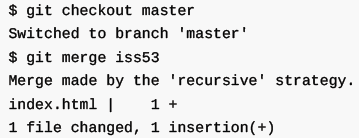


一般情况下，令master指向发布版本。

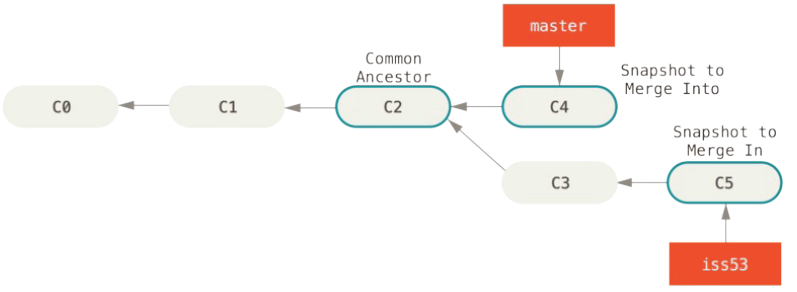
一般来讲，分支是为了解决问题而存在的，当问题解决，并进行合并后，即可删除相应分支。



只需要检出到你想合并入的分支，然后运行 git merge 命令：

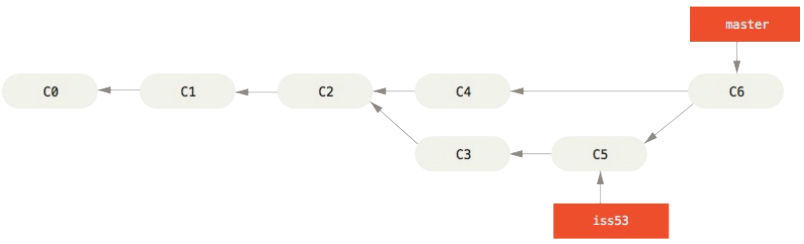


出现这种情况的时候，Git 会使用两个分支的末端所指的快照（ C4 和 C5 ）以及这两个分支的共同祖先（ C2 ），做一个简单的三方合并。



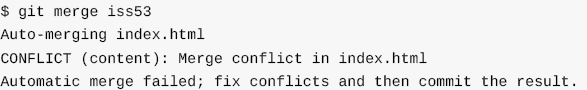
和之间将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自

动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。



### 遇到冲突时的分支合并

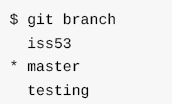
如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。





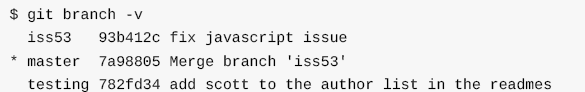
这表示 HEAD 所指示的版本在这个区段的上半部分（ ======= 的上半部分），而iss53 分支所指示的版本在 ======= 的下半部分。 为了解决冲突，你必须选择使用由======= 分割的两部分中的一个，或者你也可以自行合并这些内容。

## 分支管理

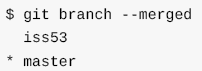


注意 master 分支前的 \* 字符：它代表现在检出的那一个分支（也就是说，当前 HEAD 指针所指向的分支）。 这意味着如果在这时候提交， master 分支将会随着新的工作向前移动。

如果需要查看每一个分支的最后一次提交，可以运行 git branch -v 命令



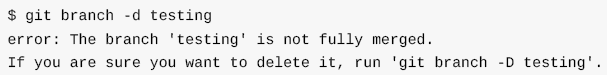
--merged 与 --no-merged 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。



在这个列表中分支名字前没有 \* 号的分支通常可以使用 git branch -d 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。



这里显示了其他分支。 因为它包含了还未合并的工作，尝试使用 git branch -d 命令删除它时会失败：



如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 -D 选项强制删除它。

Git branch --merged

可显示已经合并的分支，对于已经合并的分支，可以删除之（git branch -d）

Git branch --no-merged

可显示尚未合并的分支，对于没有合并的分支，可以强制删除之（git branch -D）

## 分支开发工作流

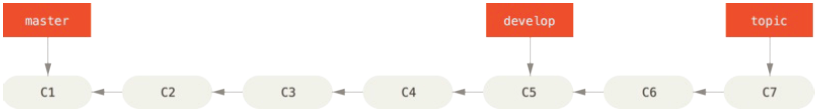
### 长期分支

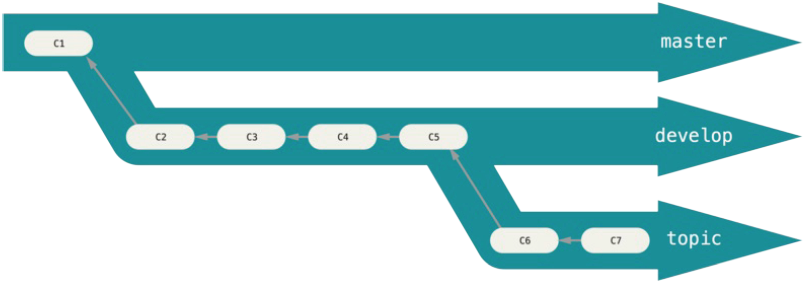
只在 master 分支上保留完全稳定的代码——有可能仅仅是已经发布或即将发布的代码

他们还有一些名为 develop 或者next 的平行分支，被用来做后续开发或者测试稳定性——这些分支不必保持绝对稳定，但是一旦达到稳定状态，它们就可以被合并入 master 分支了。

这样，在确保这些已完成的特性分支（短期分支，比如之前的 iss53 分支）能够通过所有测试，并且不会引入更多 bug 之后，就可以合并入主干分支中，等待下一次的发布。

事实上我们刚才讨论的，是随着你的提交而不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支的指针往往比较靠前。





你可以用这种方法维护不同层次的稳定性。一些大型项目还有一个 proposed （建议） 或pu: proposed updates （建议更新）分支，它可能因包含一些不成熟的内容而不能进入 next或者 master 分支。

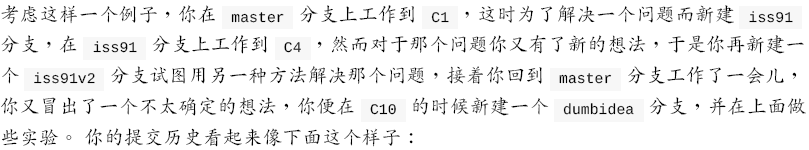
这么做的目的是使你的分支具有不同级别的稳定性。

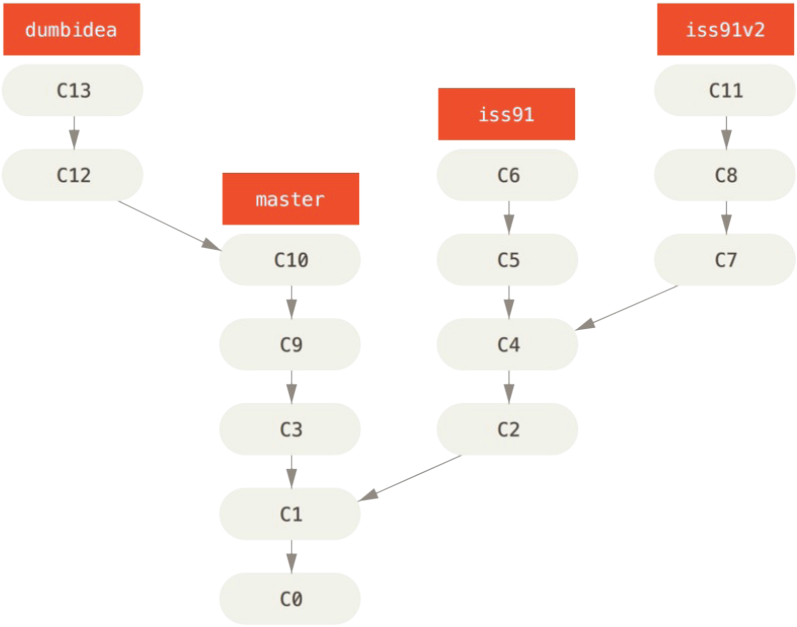
当它们具有一定程度的稳定性后，再把它们合并入具有更高级别稳定性的分支中。 再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

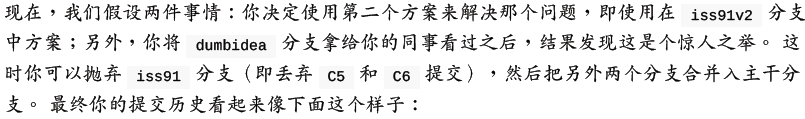
### 特性分支

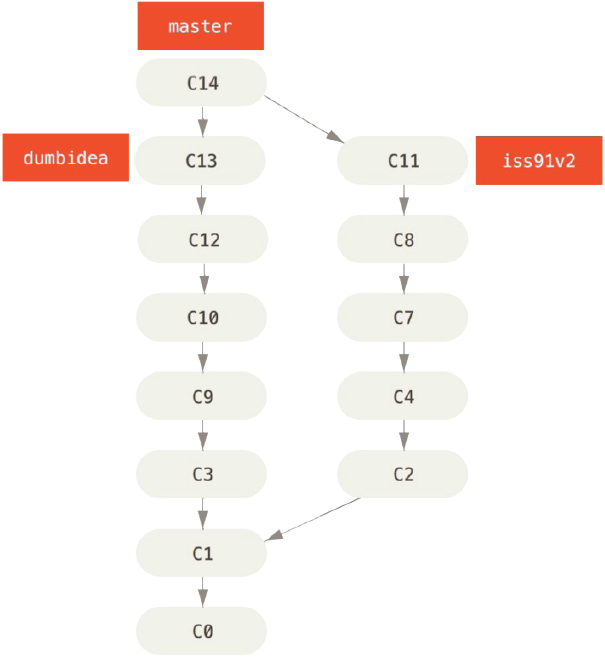
特性分支是一种短期分支，它被用来实现单一特性或其相关工作。

这项技术能使你快速并且完整地进行上下文切换（context-switch）——因为你的工作被分散到不同的流水线中，在不同的流水线中每个分支都仅与其目标特性相关，因此，在做代码审查之类的工作的时候就能更加容易地看出你做了哪些改动。你可以把做出的改动在特性分支中保留几分钟、几天甚至几个月，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度。









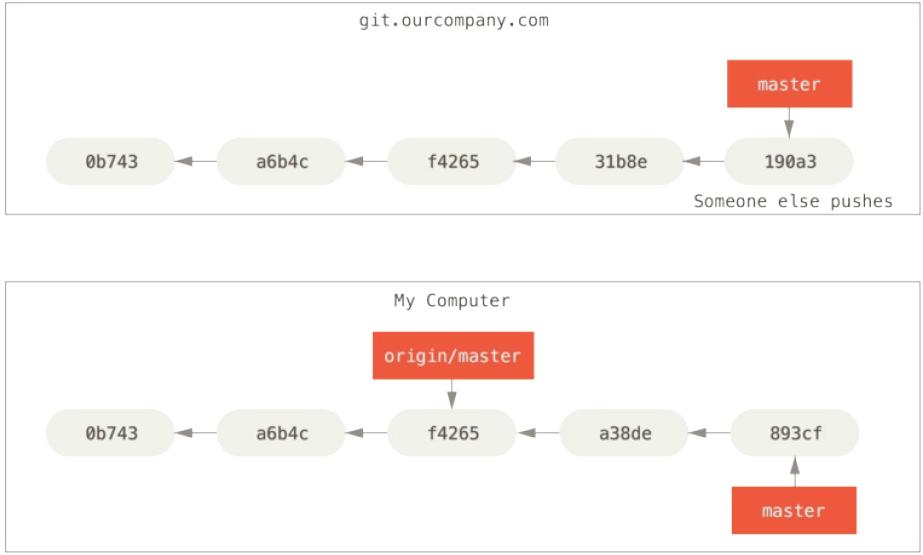
## 远程分支

### 单个远程仓库

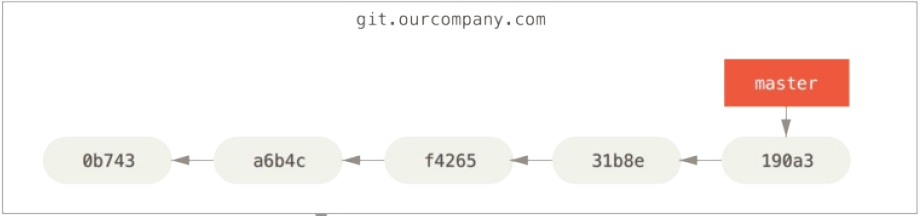
你的网络里有一个在git.ourcompany.com 的 Git 服务器。 如果你从这里克隆，Git 的 clone 命令会为你自动将其命名为 origin ，拉取它的所有数据，创建一个指向它的 master 分支的指针，并且在本地将其命名为 origin/master （默认的远程分支名称）。 Git 也会给你一个与 origin 的 master 分支在指向同一个地方的本地 master 分支，这样你就有工作的基础。

|  |
| --- |
|  |
|  |
|  |

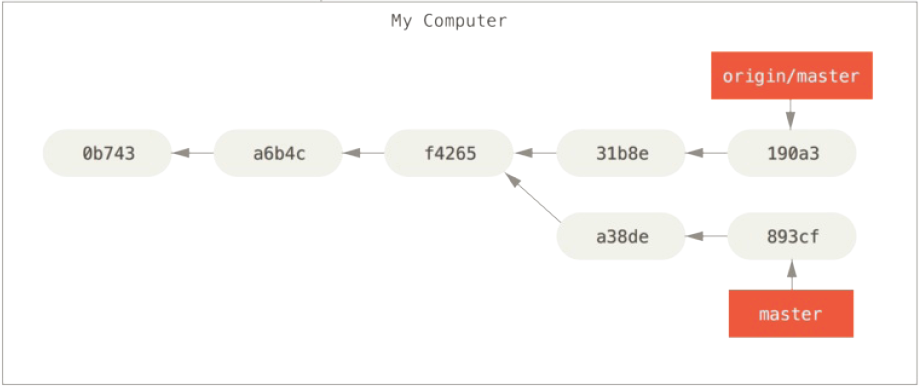
如果你在本地的 master 分支做了一些工作，然而在同一时间，其他人推送提交到git.ourcompany.com 并更新了它的 master 分支，那么你的提交历史将向不同的方向前进。也许，只要你不与 origin 服务器连接，你的 origin/master 指针就不会移动。



如果要同步你的工作，运行 git fetch origin 命令。 这个命令查找 `origin'' 是哪一个服务器（在本例中，它是 `git.ourcompany.com ），从中抓取本地没有的数据，并且更新本地数据库，移动 origin/master 指针指向新的、更新后的位置。



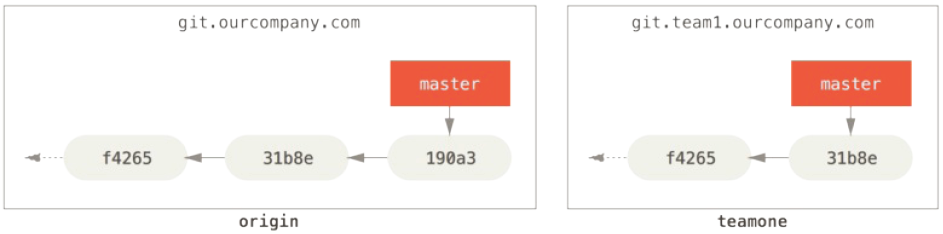




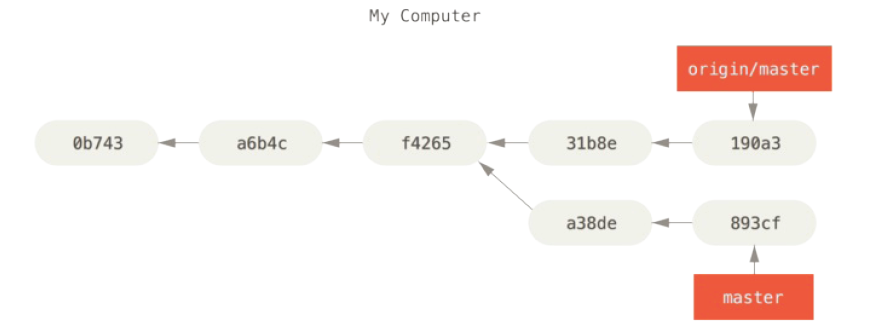
### 多个远程仓库

有多个远程仓库与远程分支的情况：

假定你有另一个内部 Git 服务器，仅用于你的 sprint 小组的开发工作。 这个服务器位于 git.team1.ourcompany.com 。 你可以运行 git remote add 命令添加一个新的远程仓库引用到当前的项目，将这个远程仓库命名为 teamone ，将其作为整个 URL 的缩写。

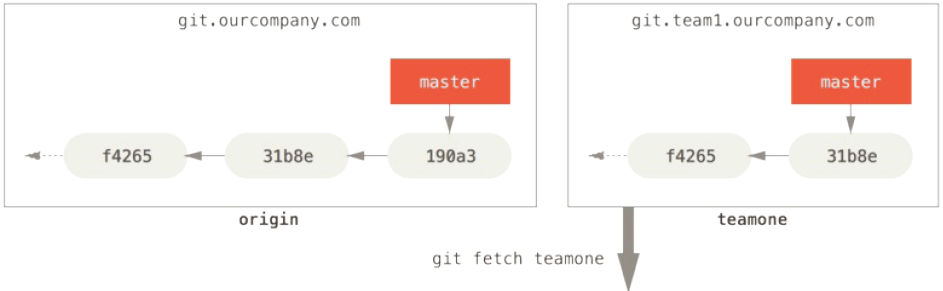


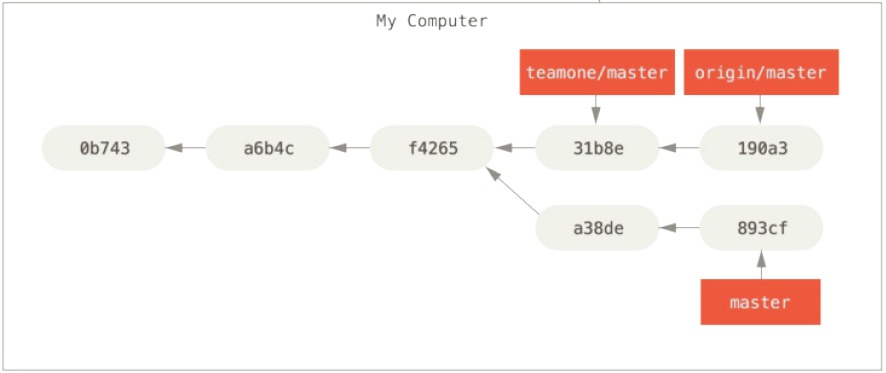




图：添加另一个远程仓库

现在，可以运行 git fetch teamone 来抓取远程仓库 teamone 有而本地没有的数据。 **因为那台服务器上现有的数据是 origin 服务器上的一个子集**，所以 Git 并不会抓取数据而是会设置远程跟踪分支 teamone/master 指向 teamone 的 master 分支。

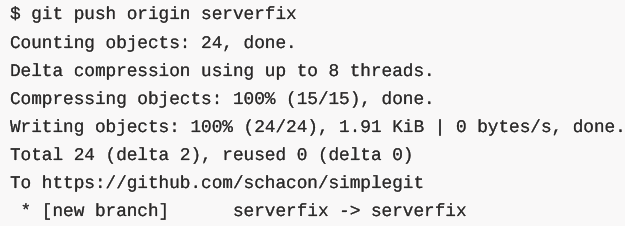




### 推送

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。 本地的分支并不会自动与远程仓库同步 - 你必须显式地推送想要分享的分支。 这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为 serverfix 的分支上工作，你可以像推送第一个分支那样推送它。 运行 git push (remote) (branch) :

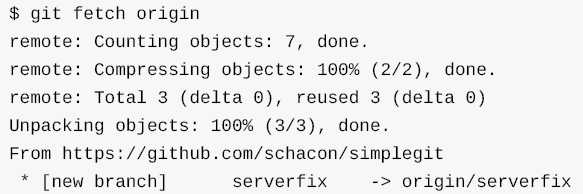


git push origin serverfix:serverfix == git push origin serverfix

git push origin serverfix:anotherName

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支

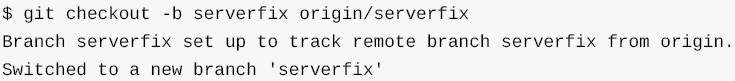
origin/serverfix ，指向服务器的 serverfix 分支的引用：



**要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 serverfix 分支 - 只有一个不可以修改的 origin/serverfix 指针。**

可以运行 git merge origin/serverfix 将这些工作合并到当前所在的本地分支。

另一种方法：如果想要在自己的 serverfix 分支上工作，可以将其建立在远程跟踪分支之上：



这会给你一个用于工作的本地分支，并且起点位于 origin/serverfix

创建一个本地分支，并让其跟踪指定远程分支，且切换至该分支。

### 跟踪分支

从一个远程跟踪分支检出一个本地分支会自动创建一个叫做 跟踪分支。跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull ，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

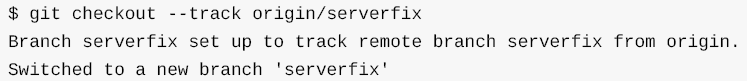
当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。

跟踪远程分支，最简单的方法如下：

$ git checkout -b branchName origin/branchName

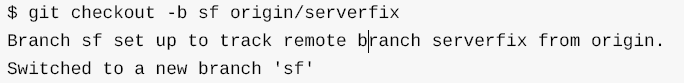
git checkout -b [branch] [remotename]/[branch]

也可用更简单的方法，如下：



跟踪指定分支后，执行pull操作，即可拉去所有的相关内容数据。

如果想要将本地分支与远程分支设置为不同名字，你可以轻松地增加一个不同名字的本地分支的上一个命令：

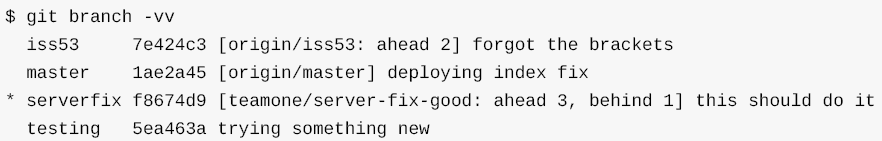


其实，设置跟踪分支后，即自动从被跟踪分支中拉取（pull）数据。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。



如果想要查看设置的所有跟踪分支，可以使用 git branch 的 -vv 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。



1. 这里可以看到 iss53 分支正在跟踪 origin/iss53 并且 `ahead'' 是 2，意味着本地有两个提交还没有推送到服务器上。
2. 也能看到 `master 分支正在跟踪 origin/master 分支并且是最新的
3. 可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 3落后 1，意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。
4. 最后看到 testing 分支并没有跟踪任何远程分支。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做： $ git fetch --all;git branch -vv

### 拉取

git pull 在大多数情况下它的含义是一个 git fetch 紧接着一个 git merge 命令。

git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。

跟踪分支可以用clone或checkout的方式来建立。gitpull 都会查找当前分支所跟踪的服务器与分支，从服务器上抓取数据然后尝试合并入那个远程分支。

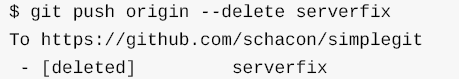
由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

分支跟踪、跟踪分支。

clone和checkout命令用以实现对远程分支的跟踪。

### 删除远程分支

假设你已经通过远程分支做完所有的工作了 - 也就是说你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 serverfix分支，运行下面的命令：

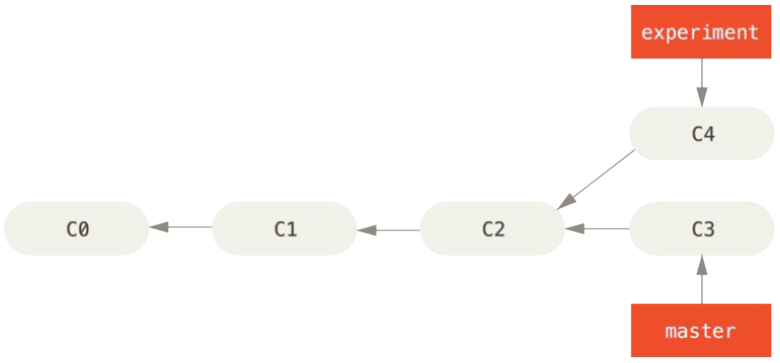


基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

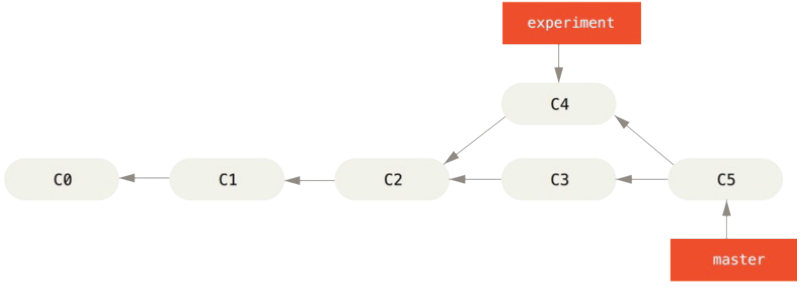
## 变基

### 变基的基本操作

在 Git 中整合来自不同分支的修改主要有两种方法：merge 以及 rebase。



整合分支最容易的方法是 merge 命令。 它会把两个分支的最新快照（ C3 和C4 ）以及二者最近的共同祖先（ C2 ）进行三方合并，合并的结果是生成一个新的快照（并提交）



### 更有趣的变基例子

### 变基的风险

### 用变基解决变基

### 变基 vs. 合并

## 总结

# 其他

***$git checkout V0.5***

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 37b2bb3 forth commit