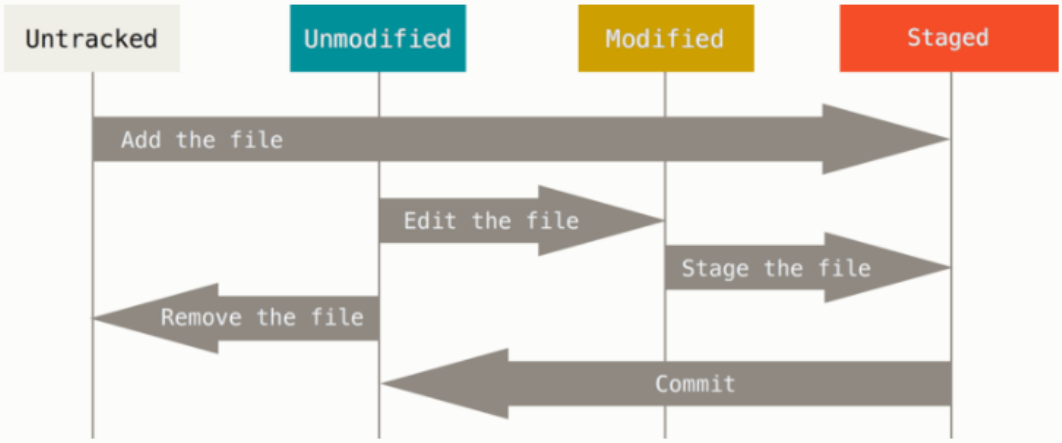
记录每次更新到仓库

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

请记住，你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：



1. 跳过暂存，直接提交

```
git commit -a -m 'xxxx'
```

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

2. 移除文件

2.1 从暂存区域移除,并连带从工作目录中删除指定的文件

```
git rm
```

如果只是简单的从工作空间里面删除，运行git status 会出现在Changes not staged for commit” 部分（也就是 未暂存清单）

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        deleted:    PROJECTS.md

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

2.2 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。

```
$ git rm --cached README
```

当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。

```
git rm  --cached 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

3. 查看提交历史

按`q`键退出历史查看

```
git log
参数说明:
-p           :显示每次提交的内容差异
-number      :number是一个变量，如-2表示显示最近的2次提交
--stat       :每次提交的简略的统计信息
--pretty     :使用不同于默认格式的方式展示提交历史 --pretty=oneline  --
pretty=format:"%h - %an, %ar : %s"
----graph    :添加了一些ASCII字符串来形象地展示你的分支、合并历史 git log --
pretty=format:"%h %s" --graph
```

4.覆盖上一次commit

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令尝试重新提交：

```
git commit --amend
```

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。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。

5.撤销暂存add

某个文件被add了，现在想要撤销暂存

```
git reset HEAD CONTRIBUTING.md
```

6.撤销对文件的修改

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

7.查看自己的远程仓库

```
git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

8.添加远程仓库

```
git remote add <shortname> <url>
$git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

9.从远程分支中拉取 git fetch

```
$ git fetch [remote-name]
```

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。

10.推送到远程分支

```
git push [remote-name] [branch-name]
$ git push origin master
```

11.别名

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

相当于

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

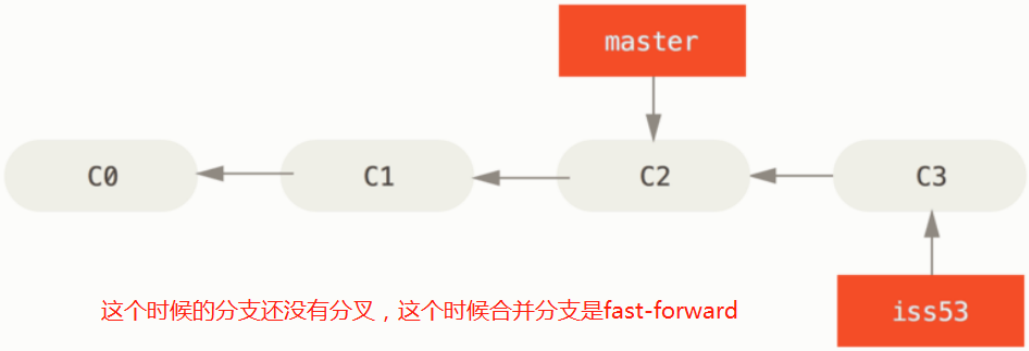
12.分支

git的分支其实就是指向提交的指针

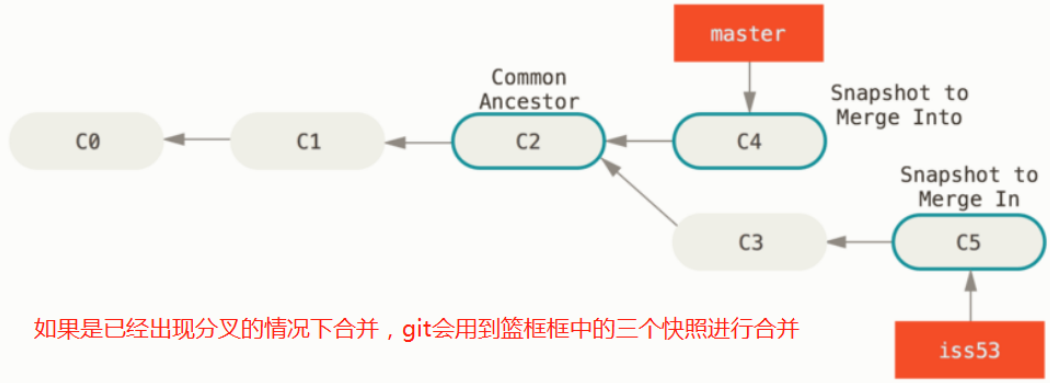
Figure 19. 创建一个新分支指针

你继续在 #53 问题上工作，并且做了一些提交。 在此过程中，`iss53` 分支在不断的向前推进，因为你已经检出到该分支（也就是说，你的 `HEAD` 指针指向了 `iss53` 分支）

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



这个时候的分支还没有分叉，这个时候合并分支是fast-forward

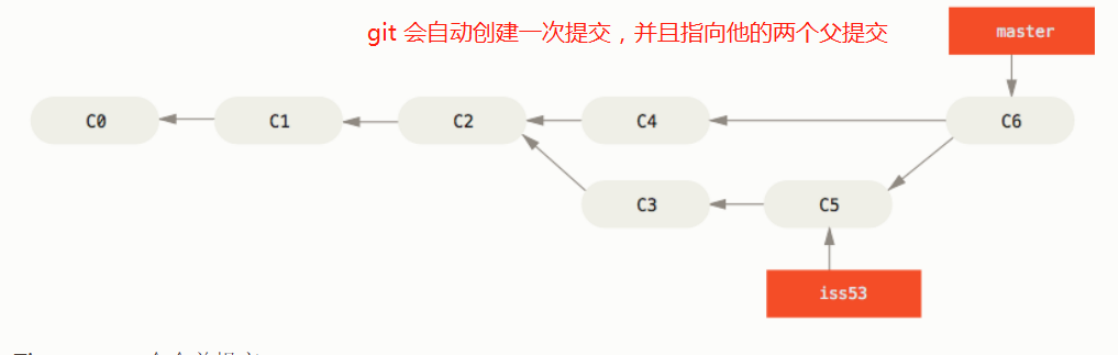


如果是已经出现分叉的情况下合并，git会用到篮框框中的三个快照进行合并

Figure 24. 一次典型合并中所用到的三个快照

Figure 24. 一次典型合并中所用到的三个快照

和之前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。



除了merge这种合并分支的做法。还有一种合并分支的操作是rebase（变基）。请看17小点

13.推送到远程分支push

```
git push [remote name 远程仓库名] [local branch 本地分支]:[remote branch 远程分支]
git push origin serverfix:serverfix
```

14.关联本地分支和远程分支

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。

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

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：

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

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。

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from
origin.
```

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。

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1]
this should do it
  testing   5ea463a trying something new
```

15.删除远程分支

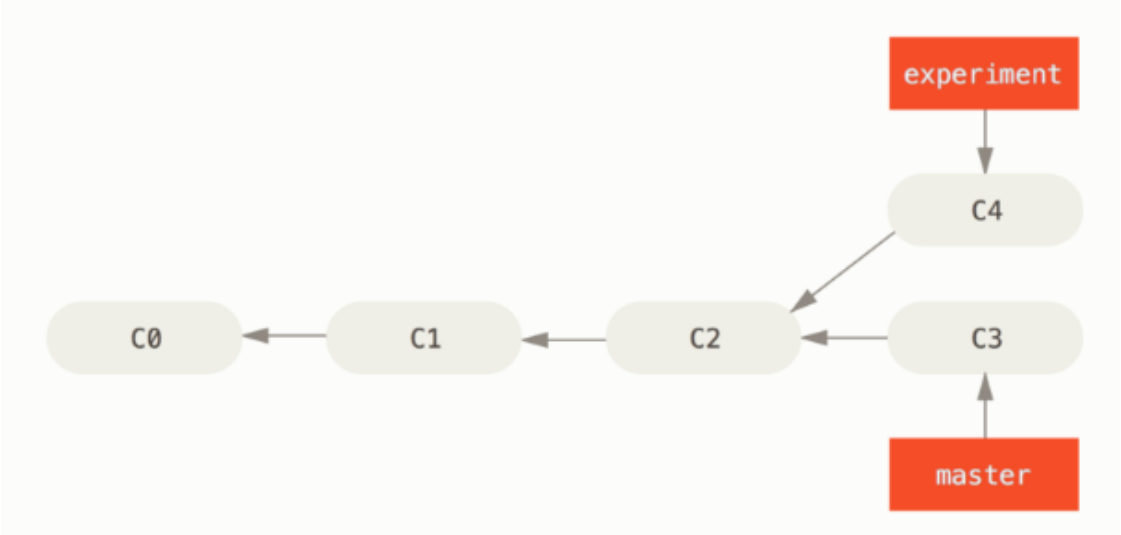
```
git push origin --delete serverfix
```

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。

16.删除本地分支

```
git branch -d <BranchName>
```

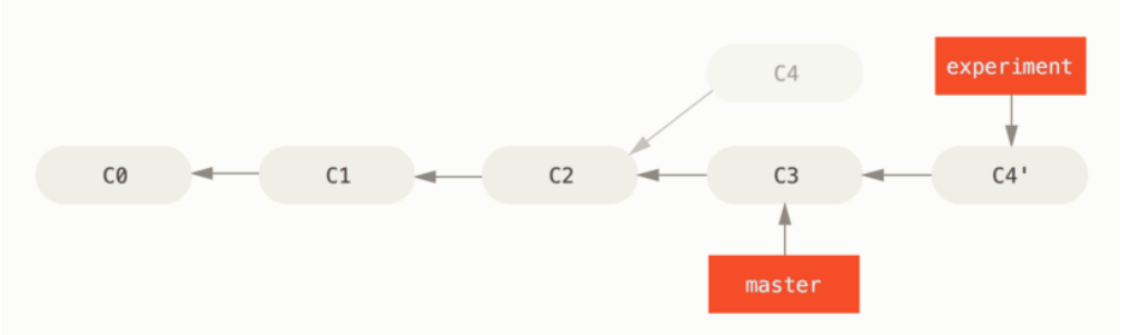
17.rebase 变基



现在有如上图的这种情况，你可以提取在 C4 中引入的补丁和修改，然后在 C3 的基础上应用一次。 在 Git 中，这种操作就叫做 变基。 你可以使用 rebase 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

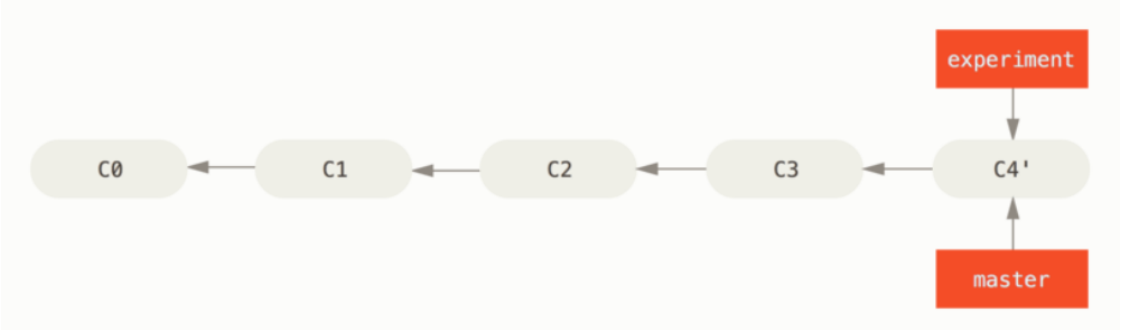
```
$ git checkout experiment //切换到要重演的分支上
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

它的原理是首先找到这两个分支（即当前分支 experiment、变基操作的目标基底分支 master）的最近共同祖先 C2，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用



现在回到 master 分支，进行一次快进合并。

```
$ git checkout master
$ git merge experiment
```



18.stash 存储于清理

```
// 暂存
$ git stash / git stash save

// 查看暂存的列表
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log

//如果想要应用其中一个更旧的储藏，可以通过名字指定它
$ git stash apply stash@{2}
git stash apply //如果不指定就是最近的版本
```

stash apply应用选项只会尝试应用暂存的工作 - 在堆栈上还有它。 如果想要在堆栈上删除他可以使用pop 和drop

```
$git stash drop stash@{0} // 删除暂存，不应用
$git stash pop //应用暂存，然后立即从暂存中删除
```

19.git grep 搜索功能

Git 提供了一个 grep 命令，你可以很方便地从提交历史或者工作目录中查找一个字符串或者正则表达式。 我们用 Git 本身源代码的查找作为例子。

默认情况下 Git 会查找你工作目录的文件。 你可以传入 -n 参数来输出 Git 所找到的匹配行行号。

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep,
struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct
tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct
tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *,
struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

git 的日志搜索能力， 或许你不想知道某一项在 哪里 ， 而是想知道是什么 时候 存在或者引入的。 git log 命令有许多强大的工具可以通过提交信息甚至是 diff 的内容来找到某个特定的提交。

例如，如果我们想找到 ZLIB_BUF_MAX 常量是什么时候引入的，我们可以使用 -S 选项来显示新增和删除该字符串的提交。

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

20.filter-branch

有另一个历史改写的选项，如果想要通过脚本的方式改写大量提交的话可以使用它 - 例如，全局修改你的邮箱地址或从每一个提交中移除一个文件。 这个命令是 filter-branch，它可以改写历史中大量的提交，除非你的项目还没有公开并且其他人没有基于要改写的工作的提交做的工作，你不应当使用它。 然而，它可以很有用。 你将会学习到几个常用的用途，这样就得到了它适合使用地方的想法。

从每一个提交移除一个文件

这经常发生。 有人粗心地通过 git add . 提交了一个巨大的二进制文件，你想要从所有地方删除它。 可能偶然地提交了一个包括一个密码的文件，然而你想要开源项目。 filter-branch 是一个可能会用来擦洗整个提交历史的工具。 为了从整个提交历史中移除一个叫做 passwords.txt 的文件，可以使用 --tree-filter 选项给 filter-branch

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

使一个子目录做为新的根目录 假设已经从另一个源代码控制系统中导入，并且有几个没意义的子目录（trunk、tags 等等）。 如果想要让 trunk 子目录作为每一个提交的新的项目根目录，filter-branch 也可以帮助你那么做：

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

全局修改邮箱地址

另一个常见的情形是在你开始工作时忘记运行 git config 来设置你的名字与邮箱地址，或者你想要开源一个项目并且修改所有你的工作邮箱地址为你的个人邮箱地址。 任何情形下，你也可以通过 filter-branch 来一次性修改多个提交中的邮箱地址。 需要小心的是只修改你自己的邮箱地址，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
```

21.中断一次合并

有时候在合并的时候发生了冲突，就不想合并了，这个时候可以用git merge --abort命令来终止合并

```
git merge --abort
```