三：git 分支管理

什么是分支

我们知道Git保存的不是文件差异或者变化量，而是一系列文件快照。

在git中做一个提交时，一个提交(commit)对象包含：

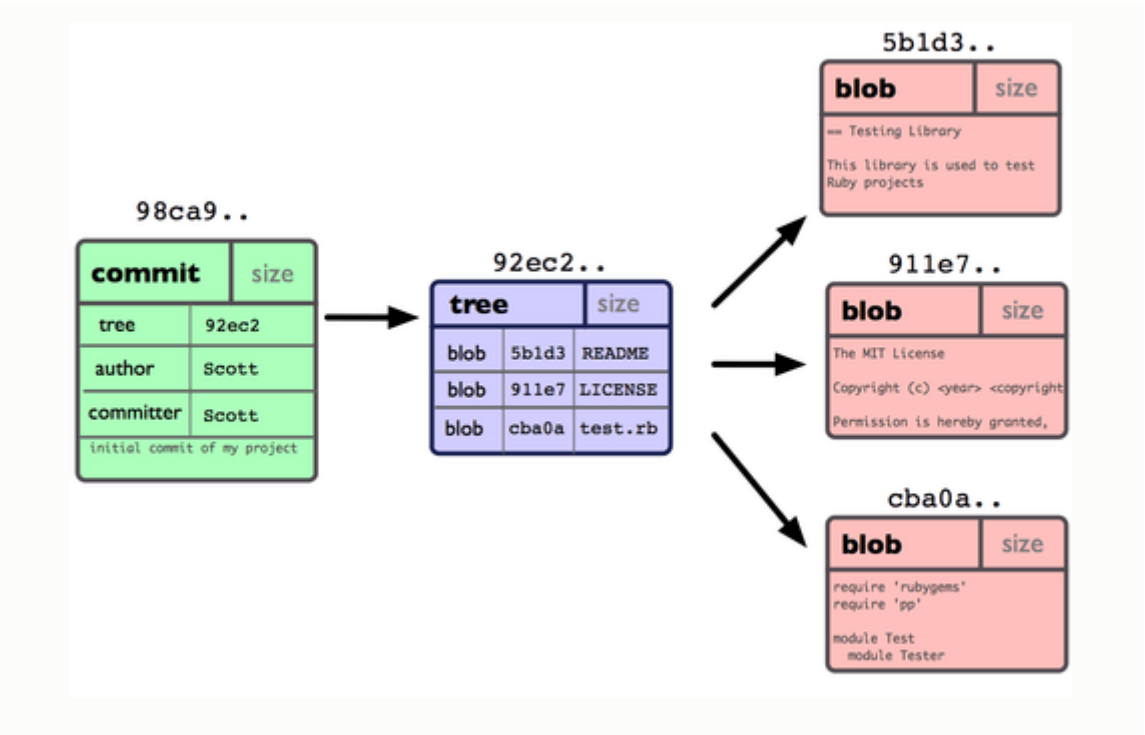
- 一个指向暂存内容快照的指针
- 本次提交的作者等相关附属信息
- 包含零个或多个指向该提交对象的父对象指针：

- 首次提交是没有直接祖先的
- 普通提交有一个祖先
- 由两个或多个分支合并产生的提交有多个祖先 例：假设我们工作目录中有三个文件，准备把它们暂存后提交。暂存操作会对每一个文件计算校验和（即SHA-1哈希字串），然后把当前版本的文件快照保存到git仓库中（git使用blob类型的对象存储这些快照），并将校验和加入暂存区域：

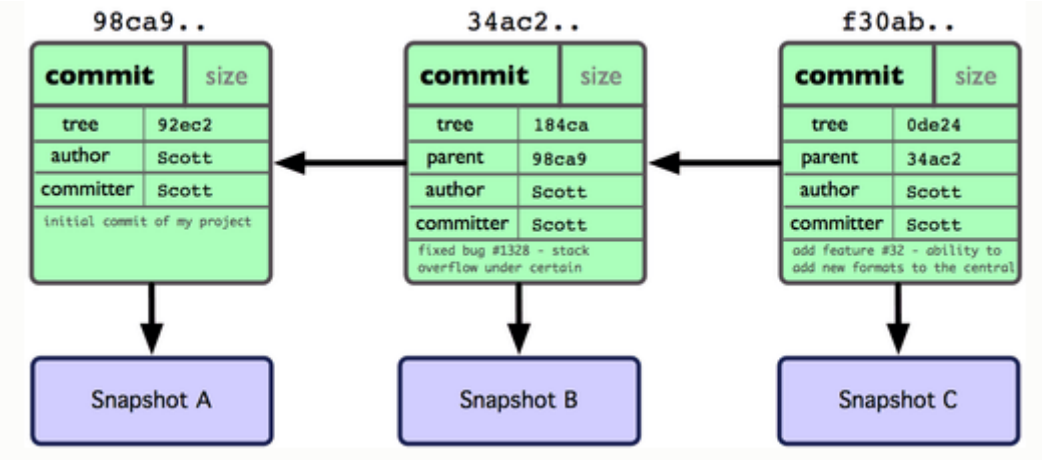
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

当使用`git commit`新建一个提交对象时，Git会计算每一个子目录（本例中就是根目录）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，重现此次内容的快照内容。

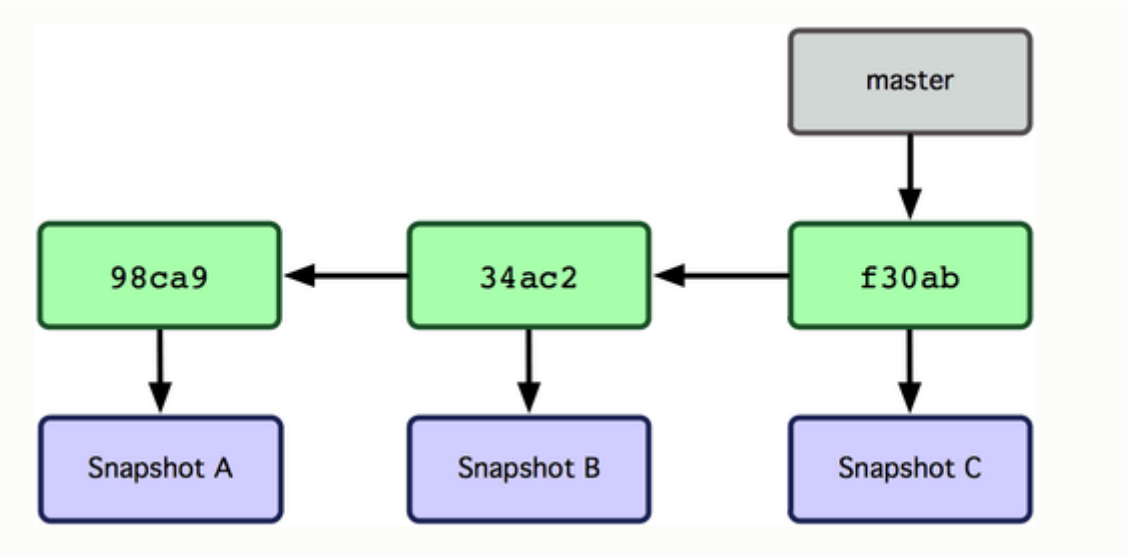
现在，Git仓库中有五个对象，三个表示文件快照内容的blob对象，一个记录目录树及其中各个文件对应blob对象索引的tree对象；以及一个包含指向tree对象（根目录）的索引和其他提交信息元数据的commit对象。仓库中各个对象保存的数据相互关系如下图：表示单个提交对象在仓库中的数据结构。



做些修改后再次提交，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（下图中的parent对象），经过两次提交，仓库会变成下图：



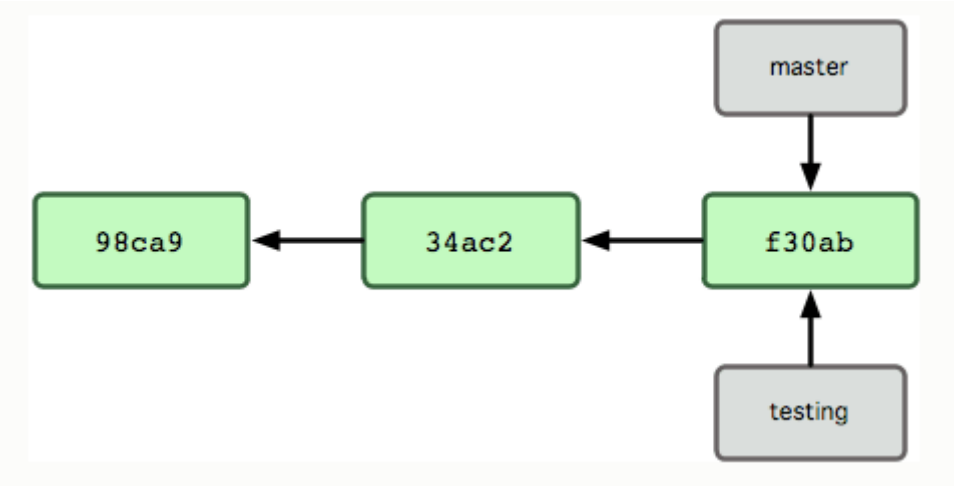
分支：Git中的分支，其实本质仅仅是个指向 commit 对象的可变指针，git会使用master作为分支的默认名字。在经过几次提交后，你已经有了一个指向最后一个提交对象的master分支，他在每次提交的时候都会自动向前移动。



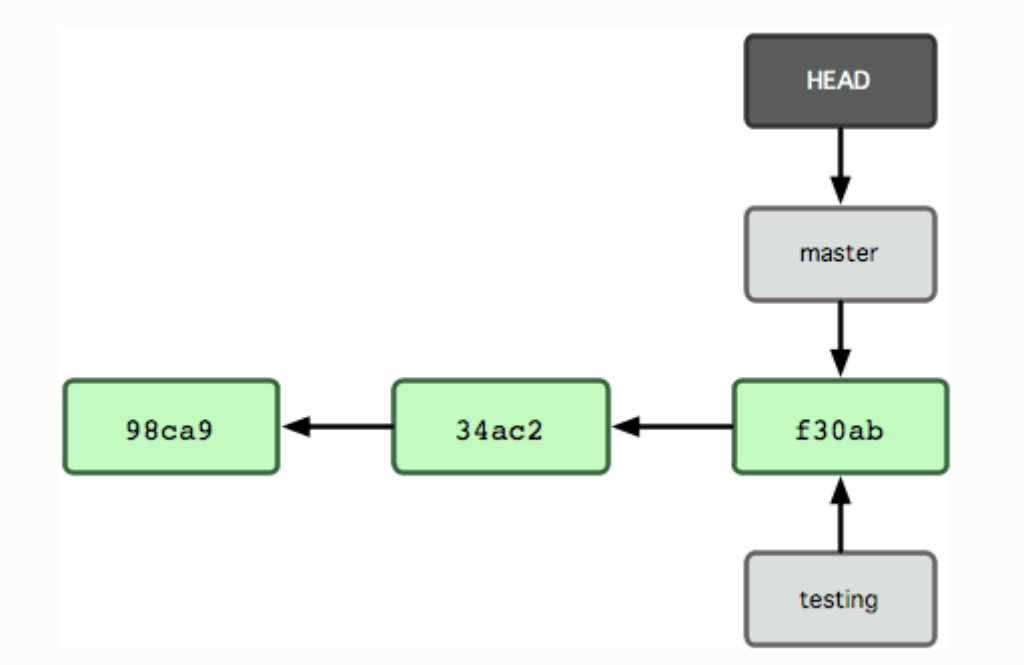
git 是如何新建分支的，如新建一个 testing 分支，可以使用如下命令。

```
$ git branch testing
```

这时会在当前commit对象上新建一个分支指针



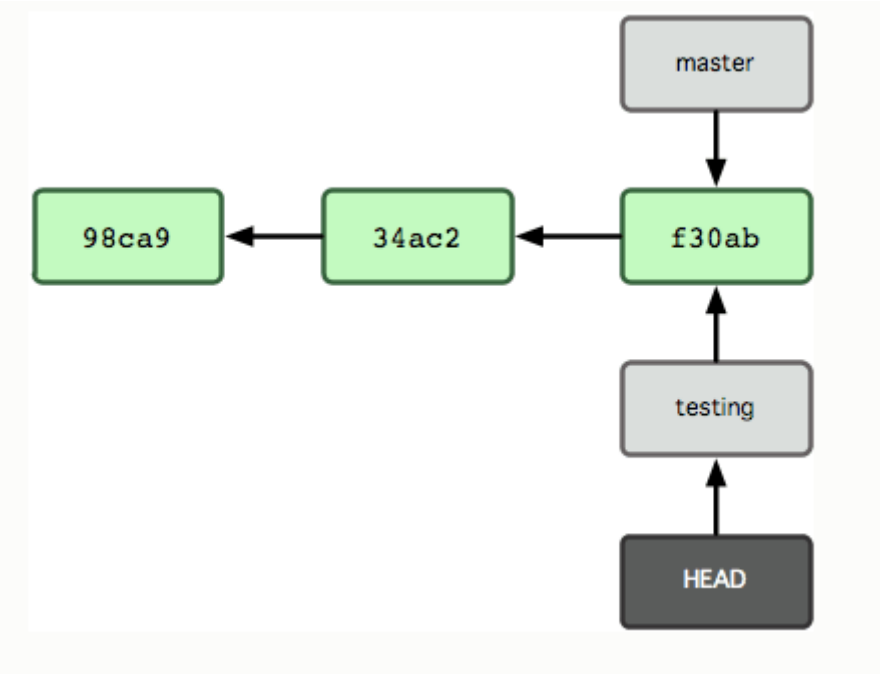
Git 是如何知道你当前是在那个分支上工作呢？其实很简单，它保存着一个名为HEAD的特别指针，它是一个指向你正在工作的本地分支的指针。运行 git branch 命令，仅仅是新建了一个分支，但不会自动切换到这个分支中去，所以本例中我们还在master分支，如下：



切换到其他分支，执行 git checkout 命令，如现在转换到新建的 testing 分支：

$ git checkout testing

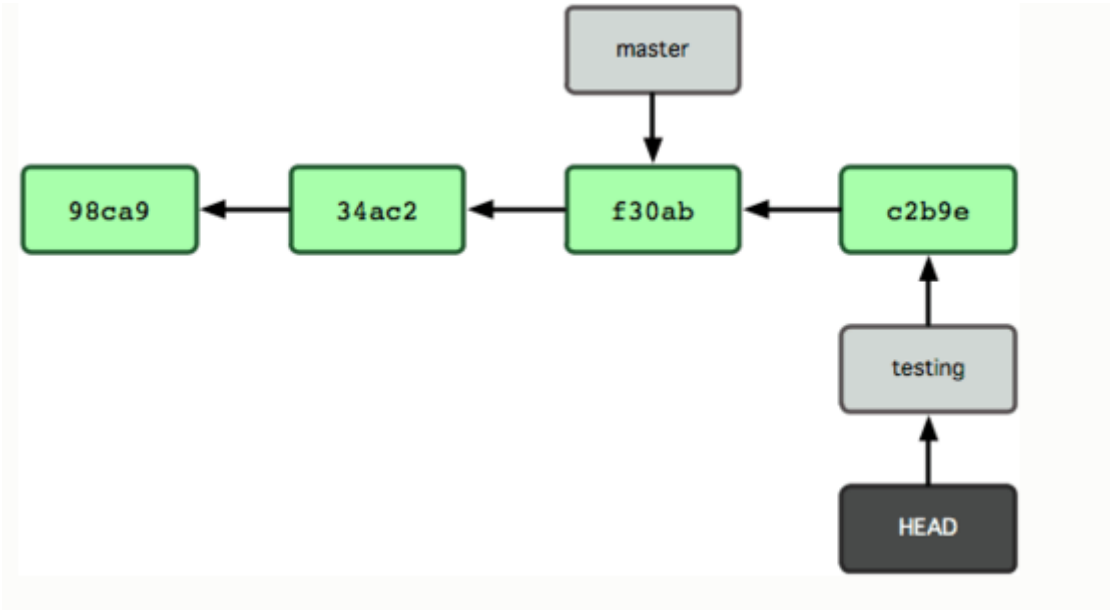
现在 HEAD 就指向了testing 分支



这种切换分支的方式有什么好处呢？现在我们再提交一次

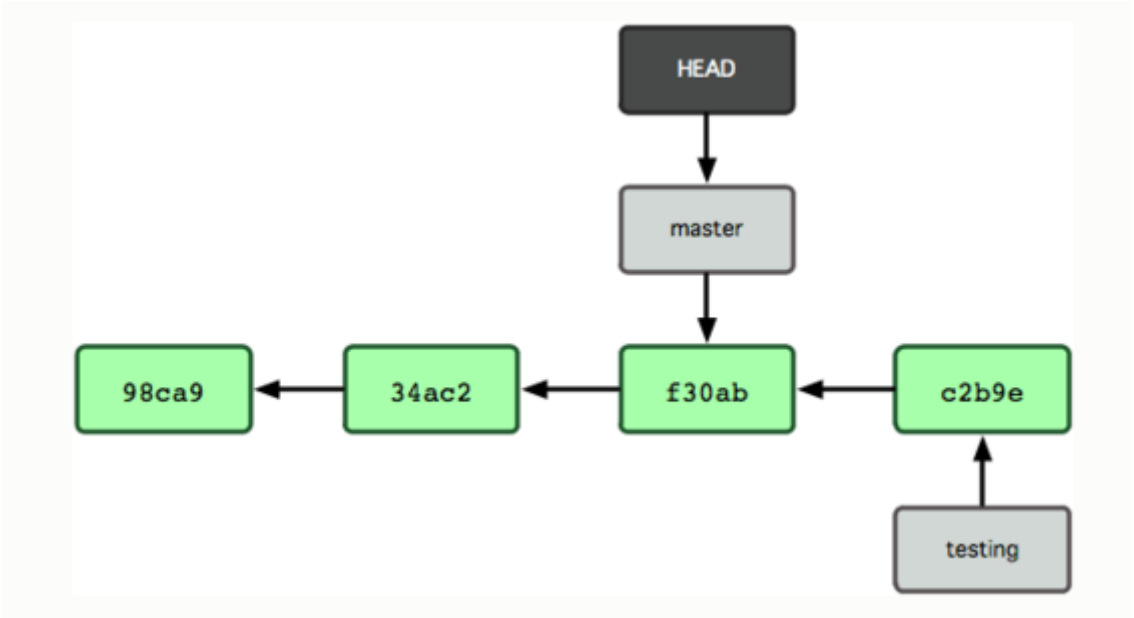
```
$ vim test.rb
$ git commit -a -m 'made a change'
```

下图是提交后的结果:



我们发现，现在 testing 分支向前移动了一格，而master分支仍然指向原先的 git checkout 切换分支时所指向的 commit 对象。现在我们回到master对象看：

```
$ git checkout master
```

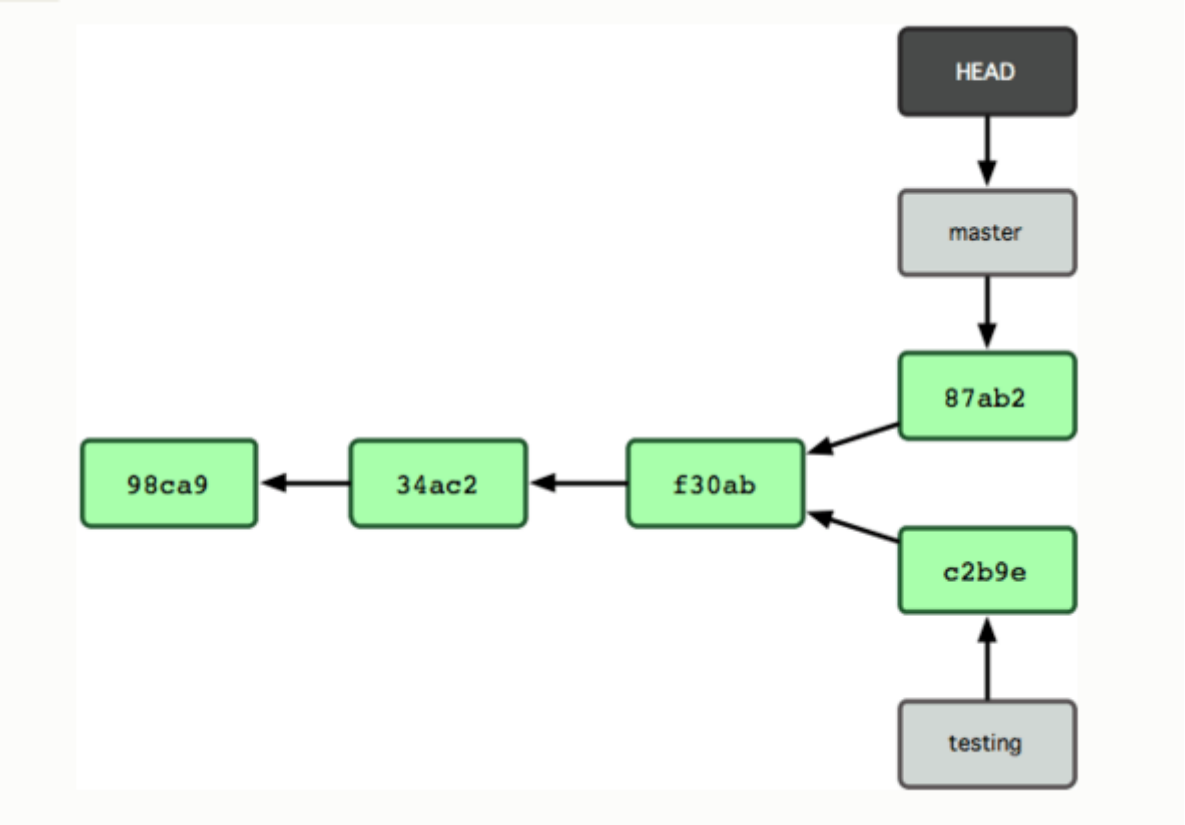


上面的命令做了两件事，它把 HEAD 指针移回到 master 分支，并把工作目录中的文件换成了 master 分支所指向的快照的内容，也就是说，现在开始的改动，将从本项目的较老的版本开始。它的主要作用是将 testing 分支里的修改暂时取消，现在你就可以向另一个方向进行开发：

我们再次做些修改后提交：

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

现在我们的项目的提交历史就产生了分叉，因为刚才我们新建了一个分支，转换到其中进行了一些工作，然后又回到原来的主分支进行了另外一些工作，这些改变孤立在不同的分支里，我们可以在这两个分支里重复切换，然后在时机成熟的时候把它们合并到一起。



git的分支实际上就是一个包含所指对象校验和（40个字符长度SHA-1字符）的文件，所以创建和销毁一个分支非常廉价，所以也就非常快。

由于git提交时每次都记录了祖先信息（parent对象），所以将来要合并分支时，只需要寻找它们的共同祖先就可以合并。

分支的新建与合并

我们来看一个简单的分支与合并的例子，实际工作中也会用到

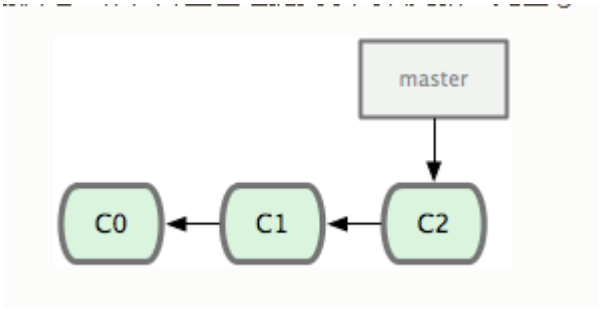
- 1.开发某个系统
- 2.为实现某个新的需求，创建一个分支
- 3.在这个分支上开展工作

假设此时，系统突然有个很严重的问题需要紧急修改，那么可以按照下面的方式处理：

- 1.返回到原先已经发布到生产服务器的分支
- 2.为这次紧急修改建立一个新分支，并在其中修改问题。
- 3.通过测试后，回到生产服务器所在分支，将修改分支合并进来，然后再推送到生产服务器上。
- 4.切换到之前实现新需求的分支，继续工作。

分支新建与切换

假设我们现在正在工作，并且已经提交了几次更新，如图：

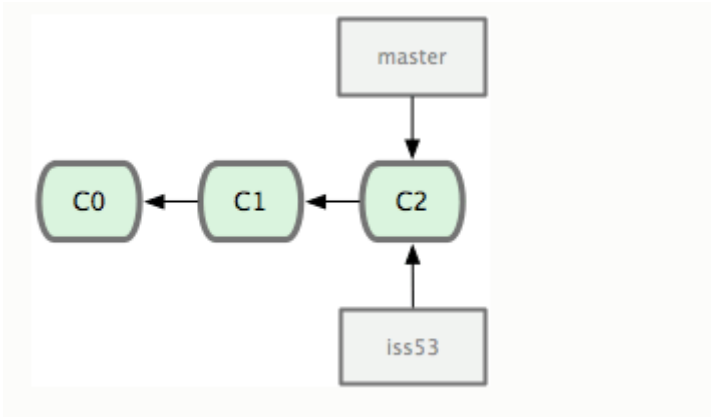


现在，你决定要修复bug系统上的 #53 问题，git不会和任何特定的问题追踪系统打交道，这里为了说明问题，我们把分支取名 iss53 ,新建并切换分支

```
$ git checkout -b iss53
Switched to a new branch 'iss53'
```

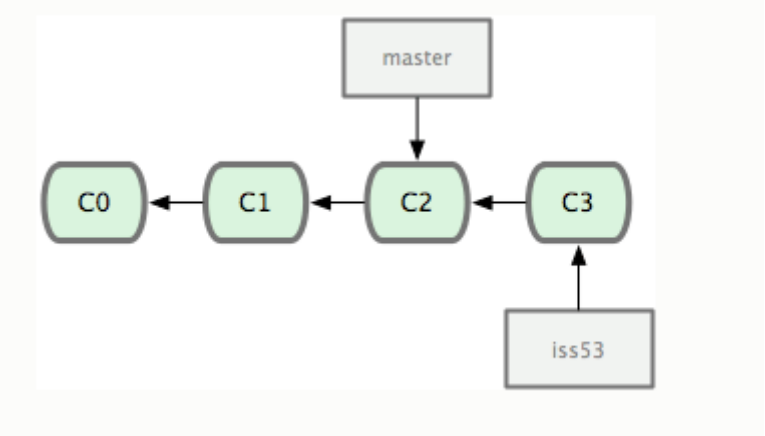
这一条命令相当于执行了两条命令

```
$ git branch iss53
$ git checkout iss53
```



接着我们修改问题，几次提交后，iss53 分支的指针会向前推进。

```
$ vim index.html
$ git commit -a -m 'modify issue53 email error'
```

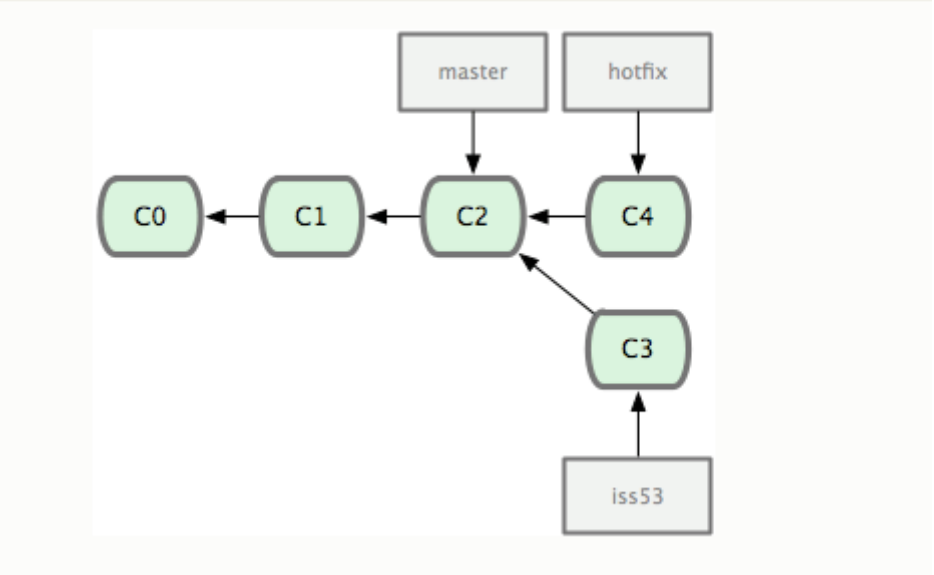


此时有个紧急要修复的问题，接下来我们要做的只是切换回master分支，不过在切换之前，要留心你的暂存区或者工作目录里，还有那些修改没有提交，它会和你即将检出的分支产生冲突而阻止 git为你切换分支。切换分支的时候最好保持一个清洁的工作区域，可以有几个绕过这种问题的方法（分别叫做 stashing 和 commit amending）,接下来切换到 master分支。

```
$ git checkout master
Switched to branch 'master'
```

此时工作目录和你在解决问题 #53 之前一模一样，接下来你得进行紧急修补。我们创建一个紧急修补分支 hotfix 来开展工作，直到修复，如下：

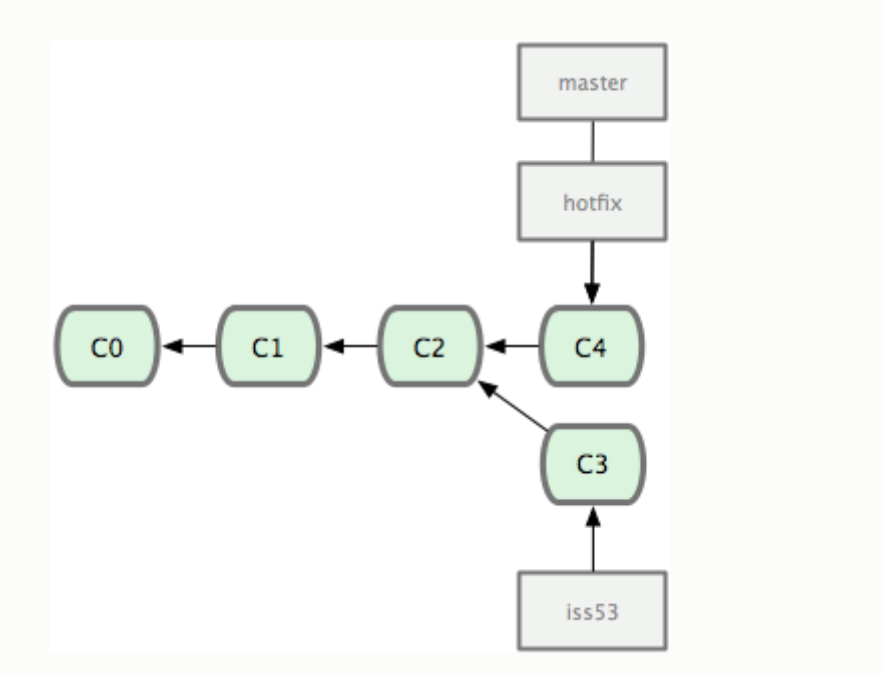
```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed bug'
[hotfix 3a0874c] fixed the broken email address
1 files changed, 1 deletion(-)
```



做必要的测试后，我们回到master分支把它合并进来，并推送到生产服务器。使用 git merge 来合并

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

合并时出现了 ‘Fast forward’ 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只需要把master分支直接右移，如果顺着一个分支走下去可以 到达另一个分支的话，git在合并时只会把指针简单右移，因为这种单线的历史分支不存在任何需要解决的分歧，所以这种合并称为快进(Fast forward)。

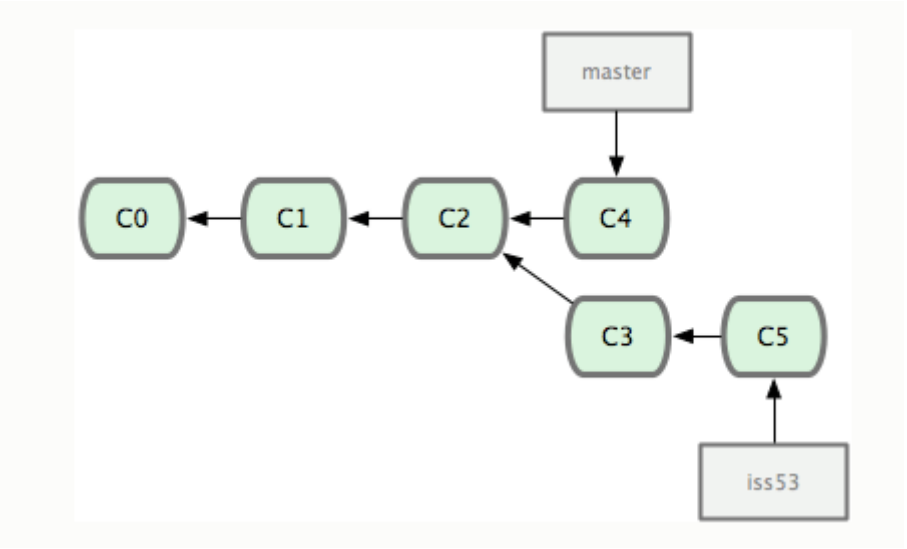


在这个紧急修复完成后，你想要回到 #53 的工作中去，由于当前 hotfix 分支 和master 都指向形同的提交对象，所以 hotfix 已经完成了历史使命，我们可以使用 git branch -d 删掉这个分支

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

现在回到之前的 #53上继续工作

```
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

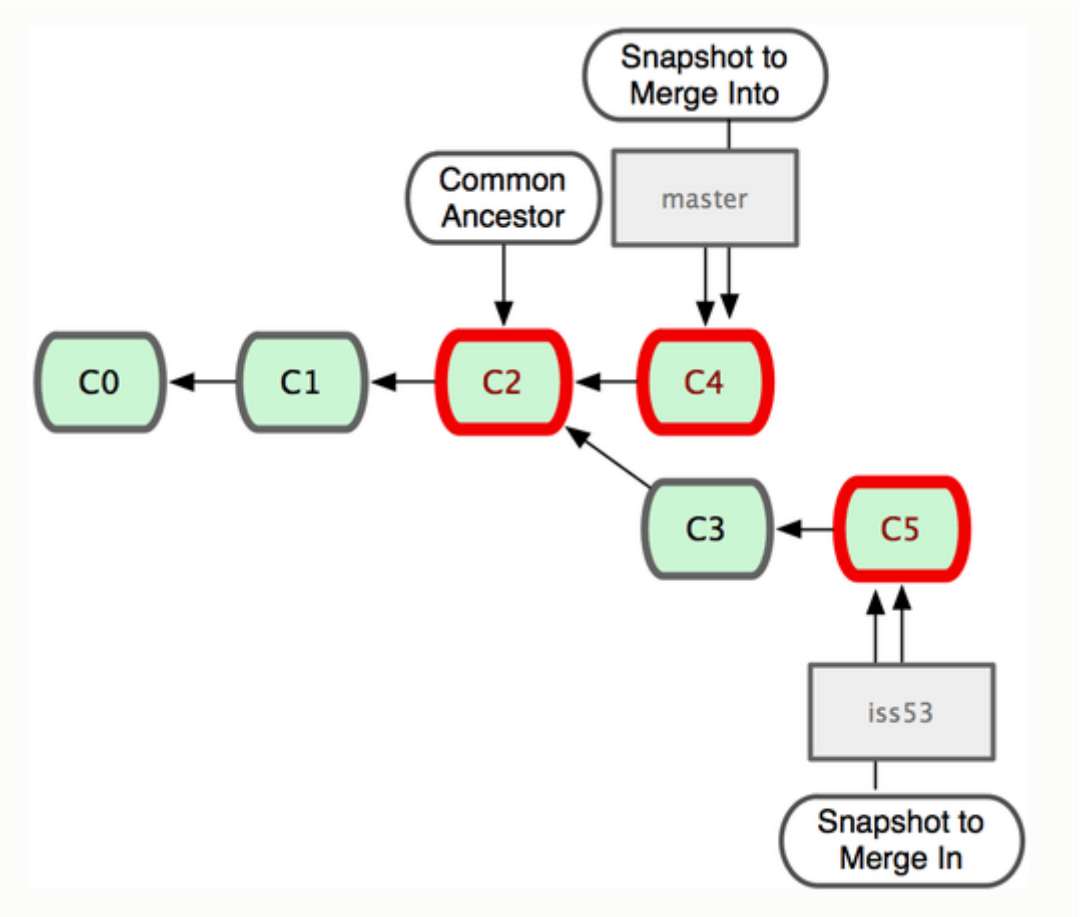


分支的合并

在问题 #53完成后，我们把它合并回master分支，实际操作和前面的 hotfix 分支差不多，只需要回到maser分支，运行 git merge 合并分支。

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

请注意,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的处理。就这个例子而言，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，下图红框标记了这三个对象。



这次，git没有简单的把分支指针右移，而是对三方合并后的结果重新做了一个新的快照，并自动创建一个指向它的提交对象（C6）。这个提交比较特殊，它有两个祖先（C4 和 C5）。git可以自己裁决出那个共同祖先才是最佳合并基础。

合并完成后，我们可以删除掉 iss53 分支

```
$ git branch -d iss53
```

- **分支合并冲突**

有时候合并不会如此顺利，如果在不同的分支中都修改了同一个文件的同一个部分，则 git 无法干净的把两者合并在一起，这种时候只能由人来判断并手工修改。如果我们在合并时遇到了如下的问题：

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

此时发生了冲突，git做了合并但是没有提交，它会停下来等你解决冲突，要看那些文件在合并时发生冲突，可以使用 git status 查看：

```
$ 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会在冲突文件里加入标砖的冲突标记，如下：

```
<<<<<<< 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 =======上半部分是master分支内容，下半部分是 iss53 分支的内容，解决方法就是二选一或者各取一部分，最后记得删掉git的冲突标记符号。在解决了所有的冲突之后，运行 git add 将他们标记为已经解决。

> 如果想用一个有图形界面的工具来解决这些问题，可以运行 git mergetool.

- **分支的管理**

git branch 不加任何参数，它会显示当前所有分支

```
$ git branch
* master
test
```

git branch -v 查看各个分支最后一个提交对象的信息

```
$ git branch -v
* master a2bbcc9 Merge branch 'test'
test 48d847e add d.txt
```

git branch --merged 和 --no-merged 可以查看当前已经合并的分支和没有合并的分支

```
$ git branch --merged
iss53
* master
```

```
$ git branch --no-merged
testing
```

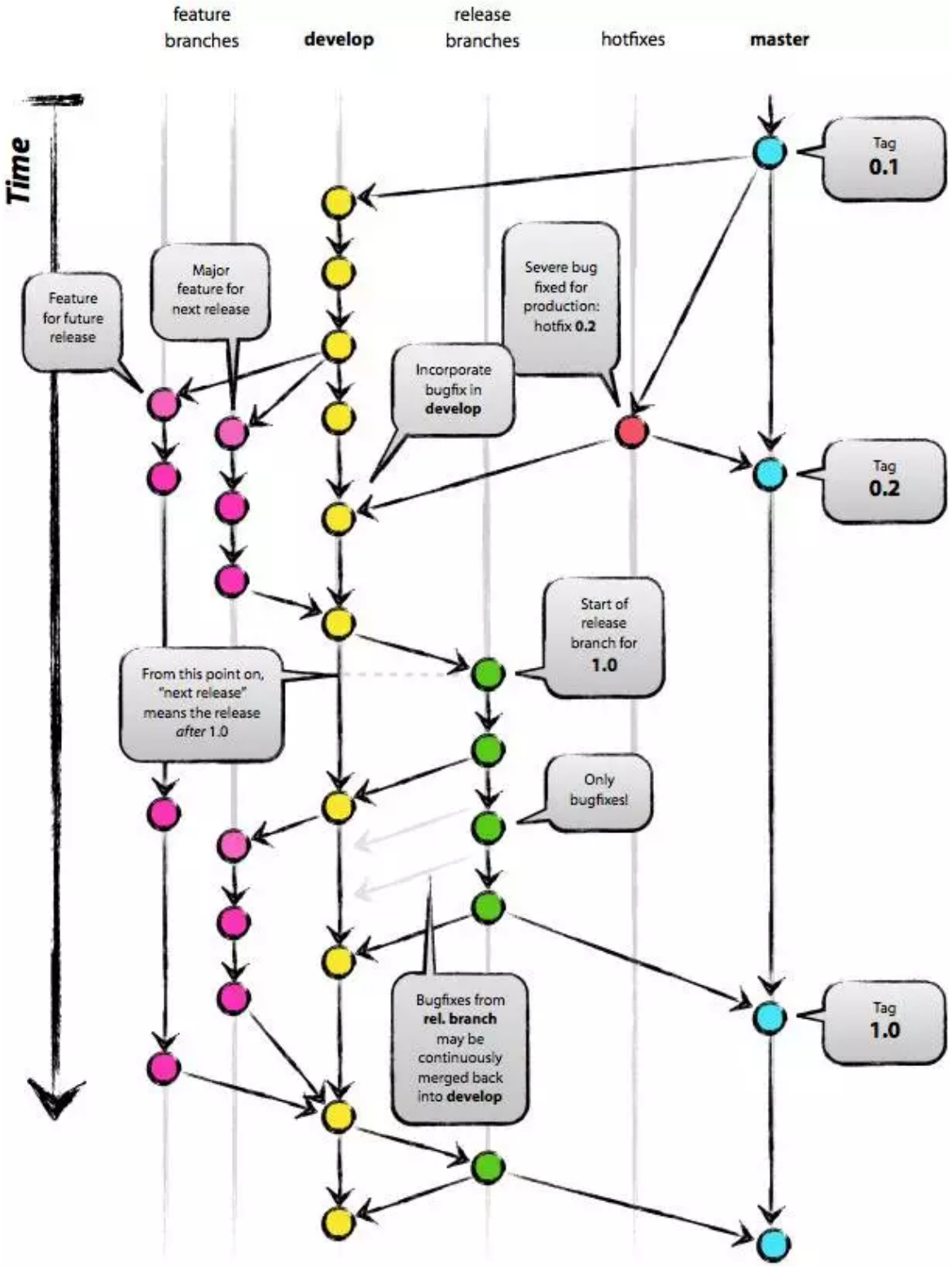
对于已经合并的分支，我们可以把它删除，而没有合并的分支，由于还包含未合并的工作成果，使用 git branch -d 会提示错误，会丢失数据。

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

我们可以用大写的 -D 强制删除

- **git工作流 Git flow**

Git Flow 是一种比较成熟的分支管理流程，下图是它的整个工作流程：



一般情况下，大部分开发会拥有两个分支，master 和 develop，它们的职责分别是：

- **master : 永远是处在即将发布版本的状态**
- **develop:最新的开发状态**

master、develop 分支大部分情况下都会保持一致，只有在上线前的测试阶段 develop比master的代码多，一旦测试没问题了，这时候会把develop分支合并到master 分支。

但是我们发布之后又会进行下一版本的功能开发，开发中间可能又会遇到需要紧急修复 bug ，一个功能开发完成之后突然需求变动了等情况，所以 Git Flow 除了以上 master 和 develop 两个主要分支以外，还提出了以下三个辅助分支：

- **feature: 开发新功能的分支, 基于 develop, 完成后 merge 回 develop**
- **release: 准备要发布版本的分支, 用来修复 bug，基于 develop，完成后 merge 回 develop 和 master**
- **hotfix: 修复 master 上的问题, 等不及 release 版本就必须马上上线. 基于 master, 完成后 merge 回 master 和 develop**

举个例子，假设我们已经有 master 和 develop 两个分支了，这个时候我们准备做一个功能 A，第一步我们要做的，就是基于 develop 分支新建个分支：

git branch feature/A

看到了吧，其实就是一个规范，规定了所有开发的功能分支都以 feature 为前缀。

但是这个时候做着做着发现线上有一个紧急的 bug 需要修复，那赶紧停下手头的工作，立刻切换到 master 分支，然后再此基础上新建一个分支：

git branch hotfix/B

代表新建了一个紧急修复分支，修复完成之后直接合并到 develop 和 master ，然后发布。

然后再切回我们的 feature/A 分支继续着我们的开发，如果开发完了，那么合并回 develop 分支，然后在 develop 分支属于测试环境，跟后端对接并且测试的差不多了，感觉可以发布到正式环境了，这个时候再新建一个 release 分支：

git branch release/1.0

这个时候所有的 api、数据等都是正式环境，然后在这个分支上进行最后的测试，发现 bug 直接进行修改，直到测试 ok 达到了发布的标准，最后把该分支合并到 develop 和 master 然后进行发布。

> Git Flow 工具开源地址 Git Flow 安装与用法