何谓分支

为了理解 Git 分支的实现方式，我们需要回顾一下 Git 是如何储存数据的。或许你还记得第一章的内容，Git 保存的不是文件差异或者变化量，而只是一系列文件快照。

在 Git 中提交时，会保存一个提交（commit）对象，该对象包含一个指向暂存内容快照的指针，包含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则有多个祖先。

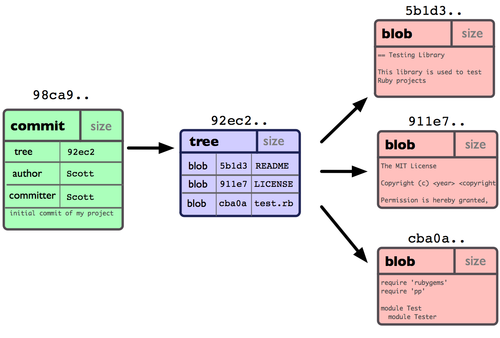
为直观起见，我们假设在工作目录中有三个文件，准备将它们暂存后提交。暂存操作会对每一个文件计算校验和（即第一章中提到的 SHA-1 哈希字串），然后把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 类型的对象存储这些快照），并将校验和加入暂存区域：

$ git add README test.rb LICENSE

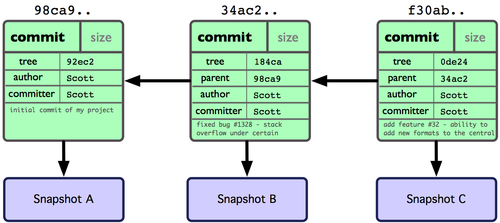
$ git commit -m 'initial commit of my project'

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，重现此次快照的内容了。

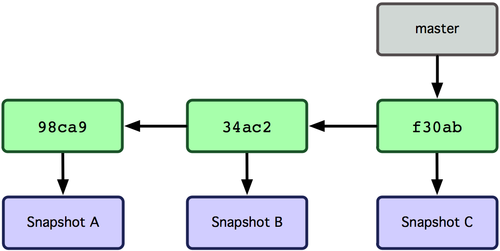
现在，Git 仓库中有五个对象：三个表示文件快照内容的 blob 对象；一个记录着目录树内容及其中各个文件对应 blob 对象索引的 tree 对象；以及一个包含指向 tree 对象（根目录）的索引和其他提交信息元数据的 commit 对象。概念上来说，仓库中的各个对象保存的数据和相互关系看起来如图 3-1 所示：

  
 图 3-1. 单个提交对象在仓库中的数据结构

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 3-2 的样子：

  
 图 3-2. 多个提交对象之间的链接关系

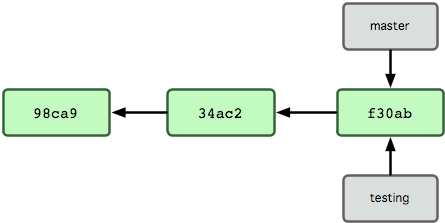
现在来谈分支。**Git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针。**Git 会使用 master 作为分支的默认名字。**在若干次提交后，你其实已经有了一个指向最后一次提交对象的 master 分支，它在每次提交的时候都会自动向前移动。**

  
 图 3-3. 分支其实就是从某个提交对象往回看的历史

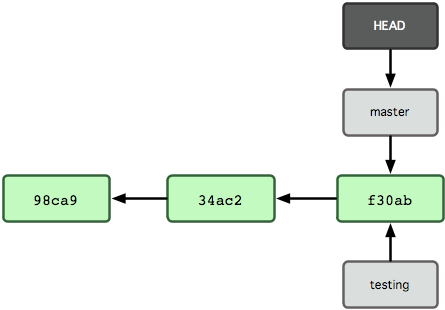
那么，Git 又是如何创建一个新的分支的呢？答案很简单，创建一个新的分支指针。比如新建一个 testing 分支，可以使用 git branch 命令：

$ git branch testing

这会在当前 commit 对象上新建一个分支指针（见图 3-4）。

  
图 3-4. 多个分支指向提交数据的历史

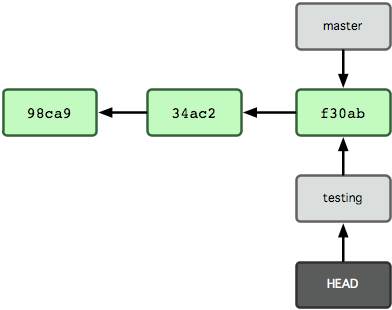
那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它**保存着一个名为 HEAD 的特别指针。**请注意它和你熟知的许多其他版本控制系统（比如 Subversion 或 CVS）里的 HEAD 概念大不相同。**在 Git 中，它是一个指向你正在工作中的本地分支的指针（译注：将 HEAD 想象为当前分支的别名。）。**运行 git branch 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 master 分支里工作（参考图 3-5）。

  
 图 3-5. HEAD 指向当前所在的分支

要切换到其他分支，可以执行 git checkout 命令。我们现在转换到新建的 testing 分支：

$ git checkout testing

这样 HEAD 就指向了 testing 分支（见图3-6）。

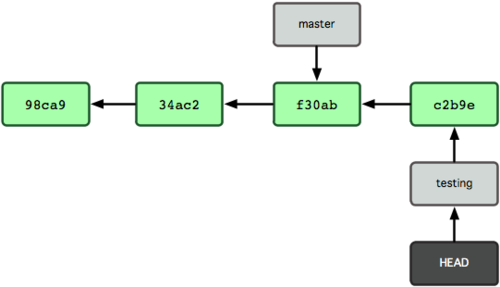
  
图 3-6. HEAD 在你转换分支时指向新的分支

这样的实现方式会给我们带来什么好处呢？好吧，现在不妨再提交一次：

$ vim test.rb

$ git commit -a -m 'made a change'

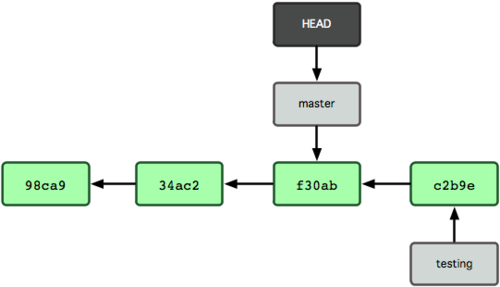
图 3-7 展示了提交后的结果。

  
图 3-7. 每次提交后 HEAD 随着分支一起向前移动

非常有趣，现在 testing 分支向前移动了一格，而 master 分支仍然指向原先 git checkout 时所在的 commit 对象。现在我们回到 master 分支看看：

$ git checkout master

图 3-8 显示了结果。

  
图 3-8. HEAD 在一次 checkout 之后移动到了另一个分支

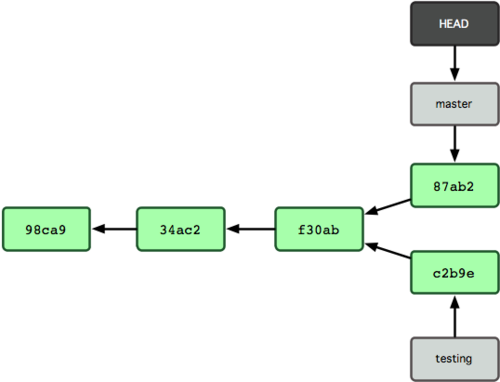
这条命令做了两件事。它把 HEAD 指针移回到 master 分支，并把工作目录中的文件换成了 master 分支所指向的快照内容。也就是说，现在开始所做的改动，将始于本项目中一个较老的版本。它的主要作用是将 testing 分支里作出的修改暂时取消，这样你就可以向另一个方向进行开发。

我们作些修改后再次提交：

$ vim test.rb

$ git commit -a -m 'made other changes'

现在我们的项目提交历史产生了分叉（如图 3-9 所示），因为刚才我们创建了一个分支，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。这些改变分别孤立在不同的分支里：我们可以在不同分支里反复切换，并在时机成熟时把它们合并到一起。而所有这些工作，仅仅需要 branch 和 checkout 这两条命令就可以完成。

  
 图 3-9. 不同流向的分支历史

由于 Git 中的分支实际上仅是一个包含所指对象校验和（40 个字符长度 SHA-1 字串）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。

这和大多数版本控制系统形成了鲜明对比，它们管理分支大多采取备份所有项目文件到特定目录的方式，所以根据项目文件数量和大小不同，可能花费的时间也会有相当大的差别，快则几秒，慢则数分钟。而 Git 的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换。同时，因为每次提交时都记录了祖先信息（译注：即 parent 对象），将来要合并分支时，寻找恰当的合并基础（译注：即共同祖先）的工作其实已经自然而然地摆在那里了，所以实现起来非常容易。Git 鼓励开发者频繁使用分支，正是因为有着这些特性作保障。

接下来看看，我们为什么应该频繁使用分支。