Git 分支

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

有人把 Git 的分支模型称为“必杀技特性”，而正是因为它，将 Git 从版本控制系统家族里区分出来。Git 有何特别之处呢？Git 的分支可谓是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。

3.1 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：

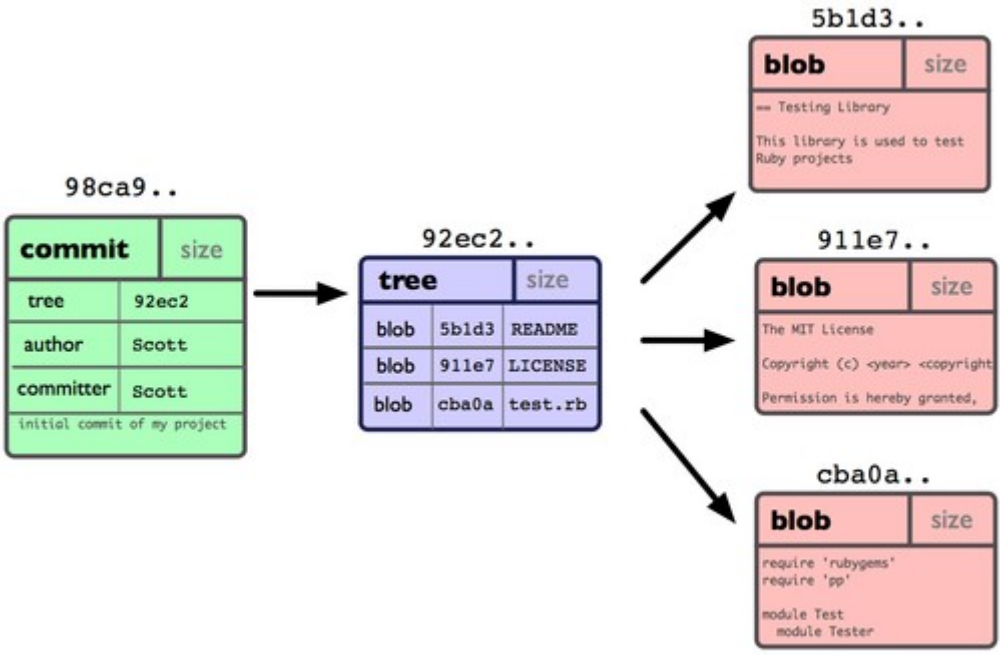

图 3-1. 单个提交对象在仓库中的数据结构

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：

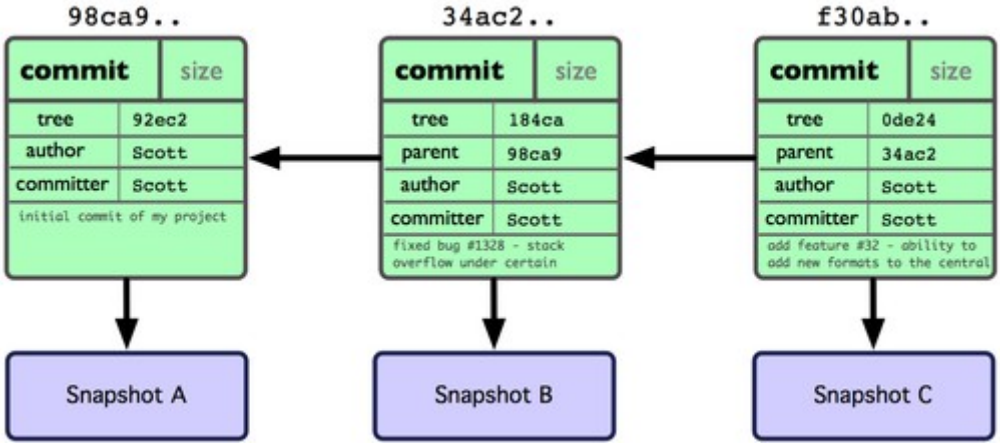

图 3-2. 多个提交对象之间的链接关系

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。

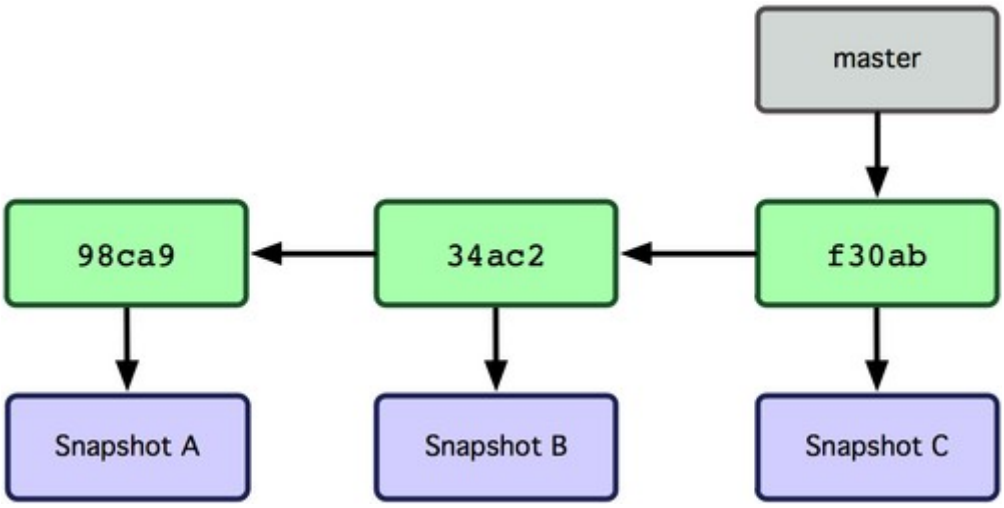

图 3-3. 分支其实就是从某个提交对象往回看的历史

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）。

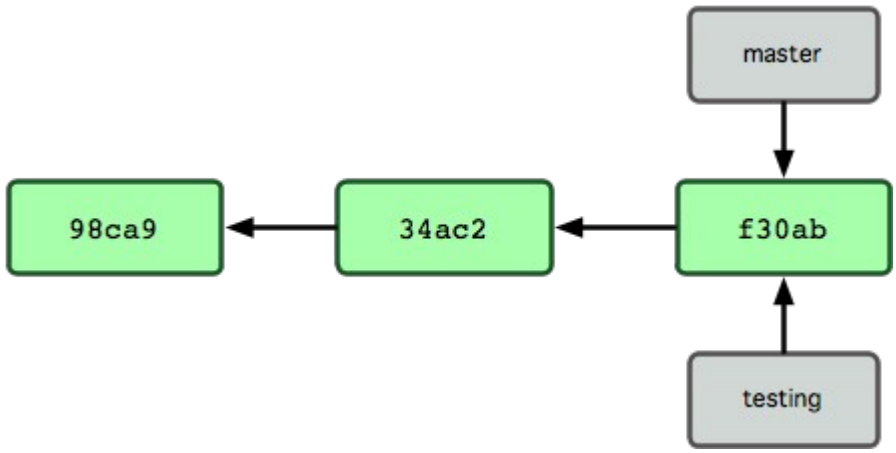

图 3-4. 多个分支指向提交数据的历史

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）。

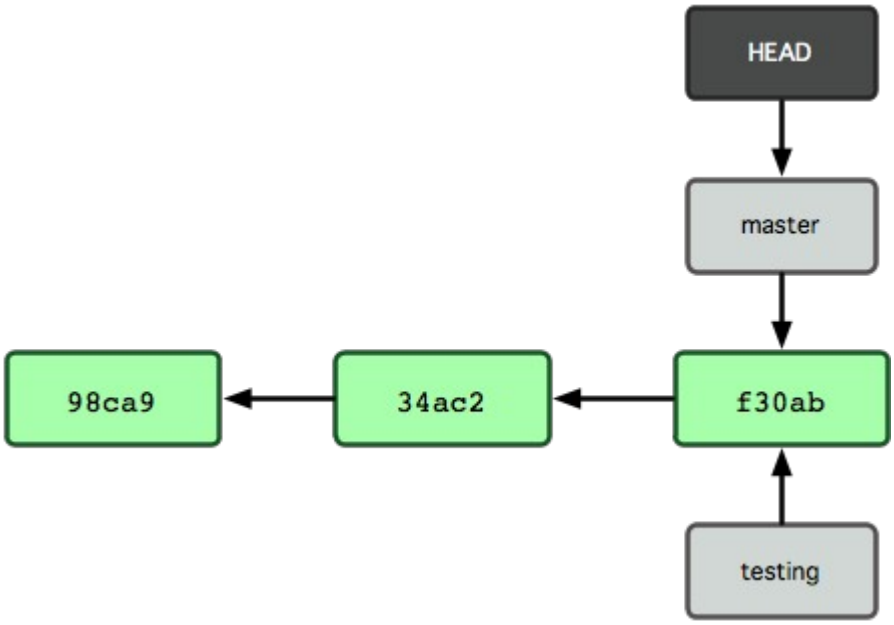

图 3-5. HEAD 指向当前所在的分支

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）。

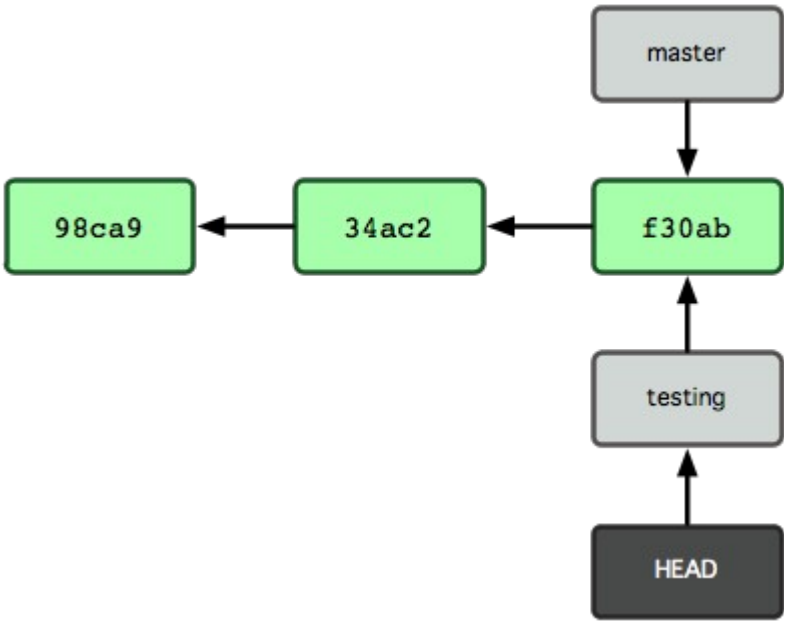

图 3-6. HEAD 在你转换分支时指向新的分支

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。

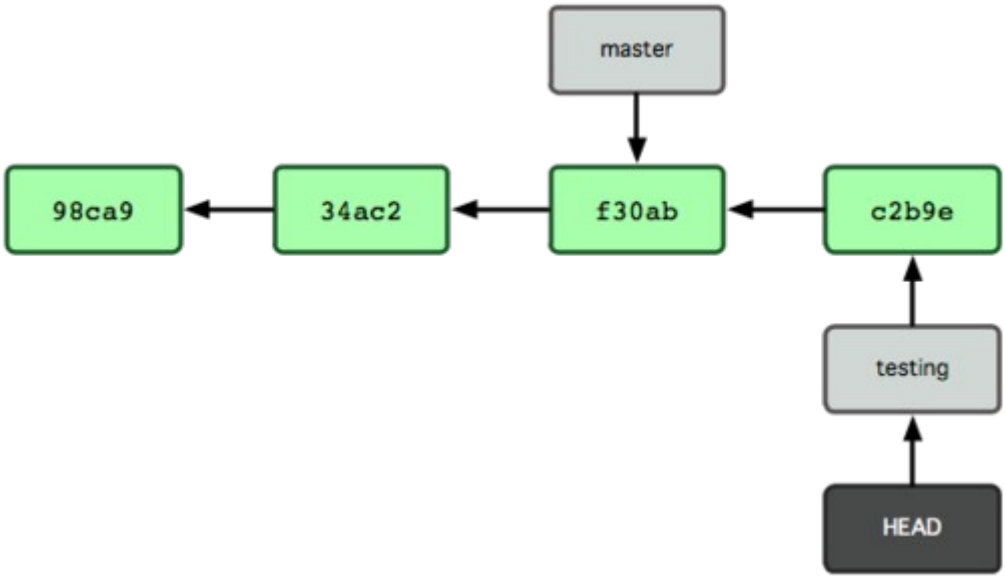

图 3-7. 每次提交后 HEAD 随着分支一起向前移动

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。

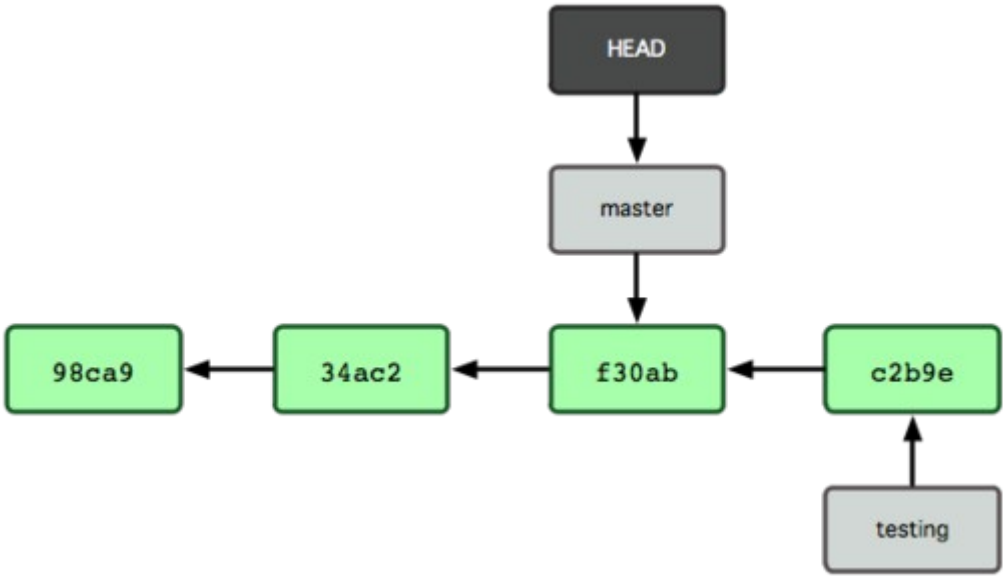

图 3-8. HEAD 在一次 checkout 之后移动到了另一个分支

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。

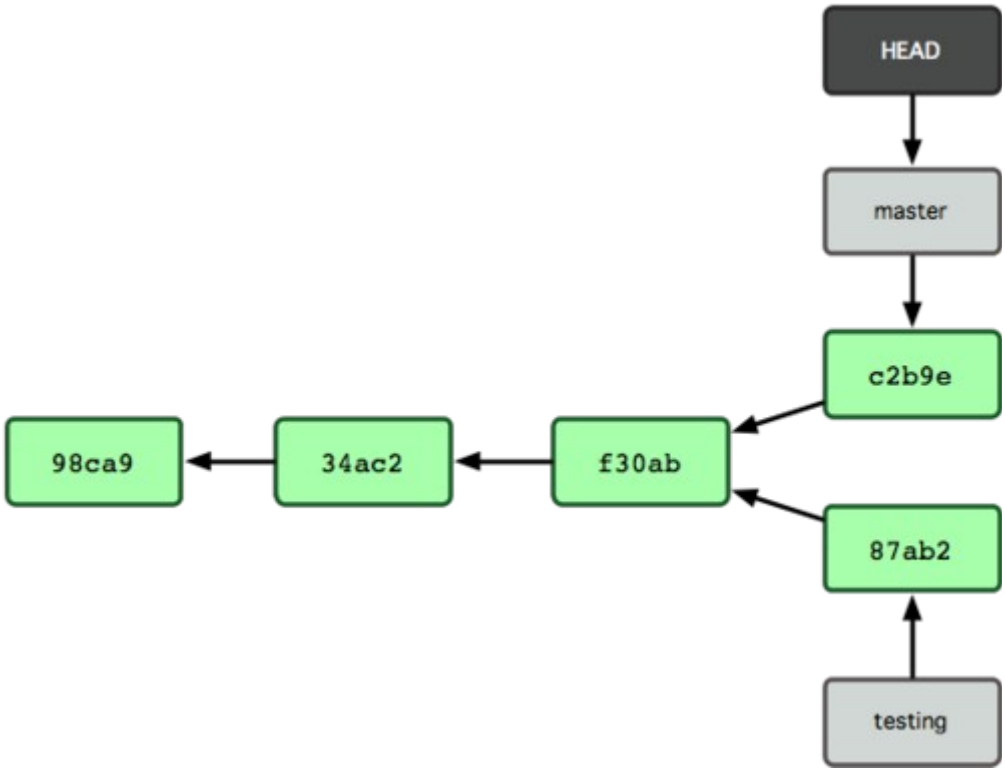

图 3-9. 不同流向的分支历史

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。

3.2 分支的新建与合并

现在让我们来看一个简单的分支与合并的例子，实际工作中大体也会用到这样的工作流程：

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。

假设此时，你突然接到一个电话说有个很严重的问题需要紧急修补，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上的分支。 2. 为这次紧急修补建立一个新分支，并在其中修复问题。 3. 通过测试后，回到生产服务器所在的分支，将修补分支合并进来，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。

分支的新建与切换

首先，我们假设你正在项目中愉快地工作，并且已经提交了几次更新（见图 3-10）。

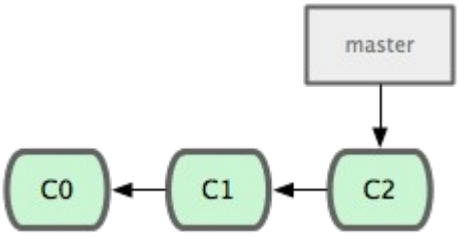

图 3-10. 一个简短的提交历史

现在，你决定要修补问题追踪系统上的 #53 问题。顺带说明下，Git 并不同任何特定的问题追踪系统打交道。这里为了说明要解决的问题，才把新建的分支取名为 iss53。要新建并切换到该分支，运行 `git checkout` 并加上 `-b` 参数：

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

这相当于执行下面这两条命令：

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

图 3-11 示意该命令的执行结果。

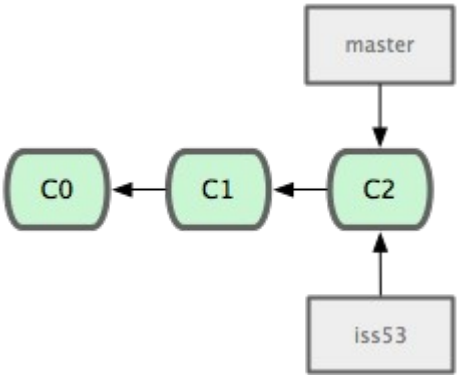

图 3-11. 创建了一个新分支的指针

接着你开始尝试修复问题，在提交了若干次更新后，iss53 分支的指针也会随着向前推进，因为它就是当前分支（换句话说，当前的 HEAD 指针正指向 iss53，见图 3-12）：

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

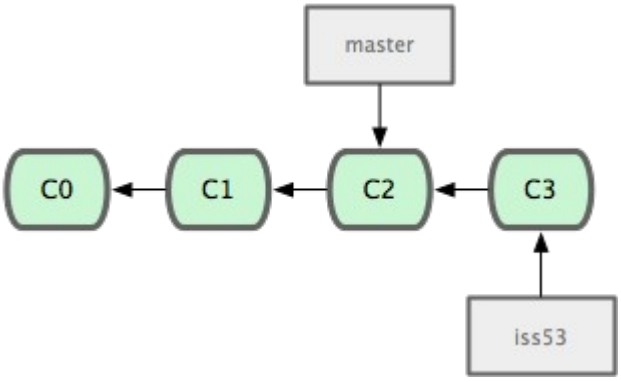

图 3-12. iss53 分支随工作进展向前推进

现在你就接到了那个网站问题的紧急电话，需要马上修补。有了 Git，我们就不需要同时发布这个补丁和 iss53 里作出的修改，也不需要在创建和发布该补丁到服务器之前花费大力气来复原这些修改。唯一需要的仅仅是切换回 master 分支。

不过在此之前，留心你的暂存区或者工作目录里，那些还没有提交的修改，它会和你即将检出的分支产生冲突从而阻止 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。稍后会介绍几个绕过这种问题的办法（分别叫做 stashing 和 commit amending）。目前已经提交了所有的修改，所以接下来可以正常转换到 master 分支：

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

此时工作目录中的内容和你在解决问题 #53 之前一模一样，你可以集中精力进行紧急修补。这一点值得牢记：Git 会把工作目录的内容恢复为检出某分支时它所指向的那个提交对象的快照。它会自动添加、删除和修改文件以确保目录的内容和你当时提交时完全一样。

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，直到搞定（见图 3-13）：

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

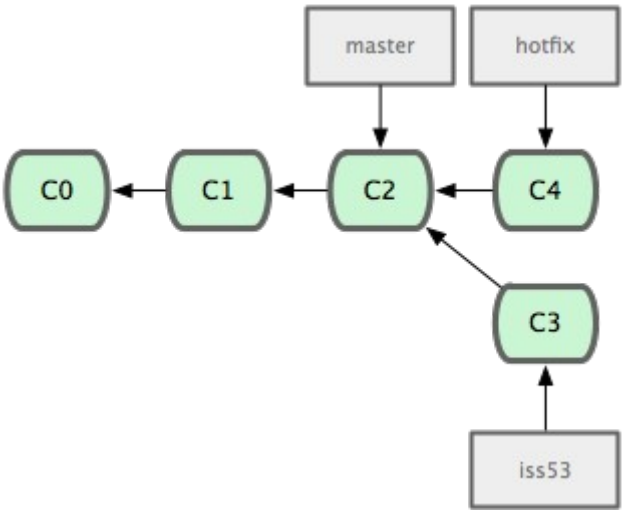

图 3-13. hotfix 分支是从 master 分支所在点分化出来的

有必要作些测试，确保修补是成功的，然后回到 master 分支并把它合并进来，然后发布到生产服务器。用 git merge 命令来进行合并：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

请注意，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（见图 3-14）。

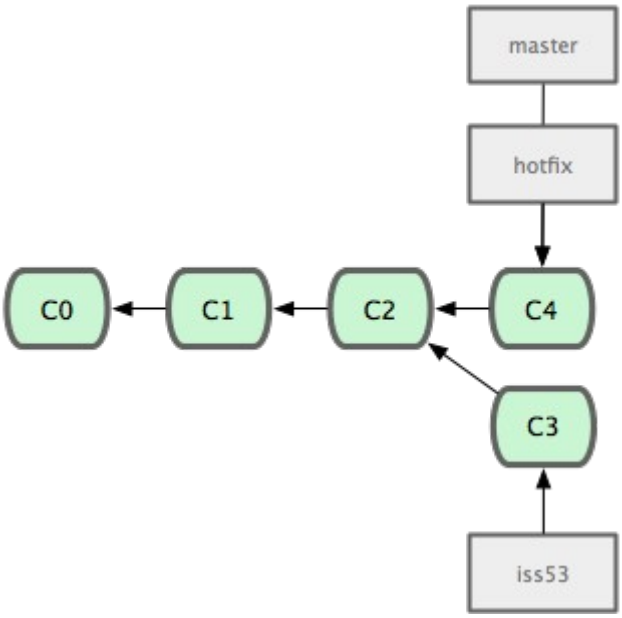

图 3-14. 合并之后，master 分支和 hotfix 分支指向同一位置。

在那个超级重要的修补发布以后，你想要回到被打扰之前的工作。由于当前 hotfix 分支和 master 都指向相同的提交对象，所以 hotfix 已经完成了历史使命，可以删掉了。使用 git branch 的 -d 选项执行删除操作：

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

现在回到之前未完成的 #53 问题修复分支上继续工作（图 3-15）：

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

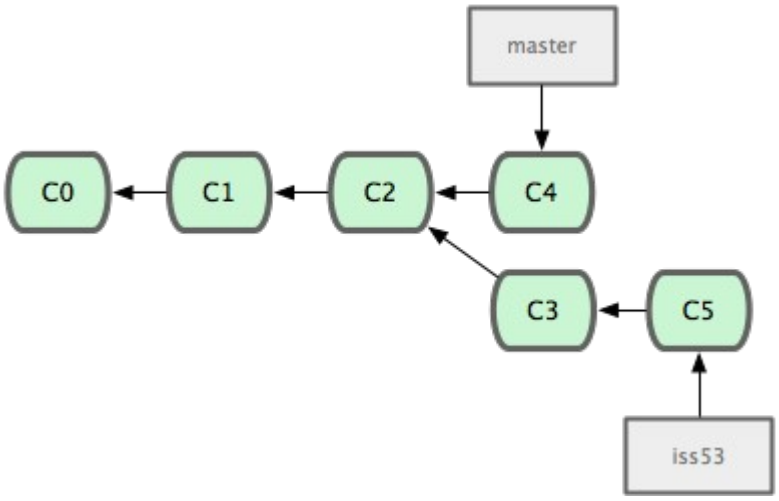

图 3-15. iss53 分支可以不受影响继续推进。

不用担心之前 hotfix 分支的修改内容尚未包含到 iss53 中来。如果确实需要纳入此次修补，可以用 git merge master 把 master 分支合并到 iss53；或者等 iss53 完成之后，再将 iss53 分支中的更新并入 master。

分支的合并

在问题 #53 相关的工作完成之后，可以合并回 master 分支。实际操作同前面合并 hotfix 分支差不多，只需回到 master 分支，运行 git merge 命令指定要合并进来的分支：

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

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。图 3-16 用红框标出了 Git 用于合并的三个提交对象：

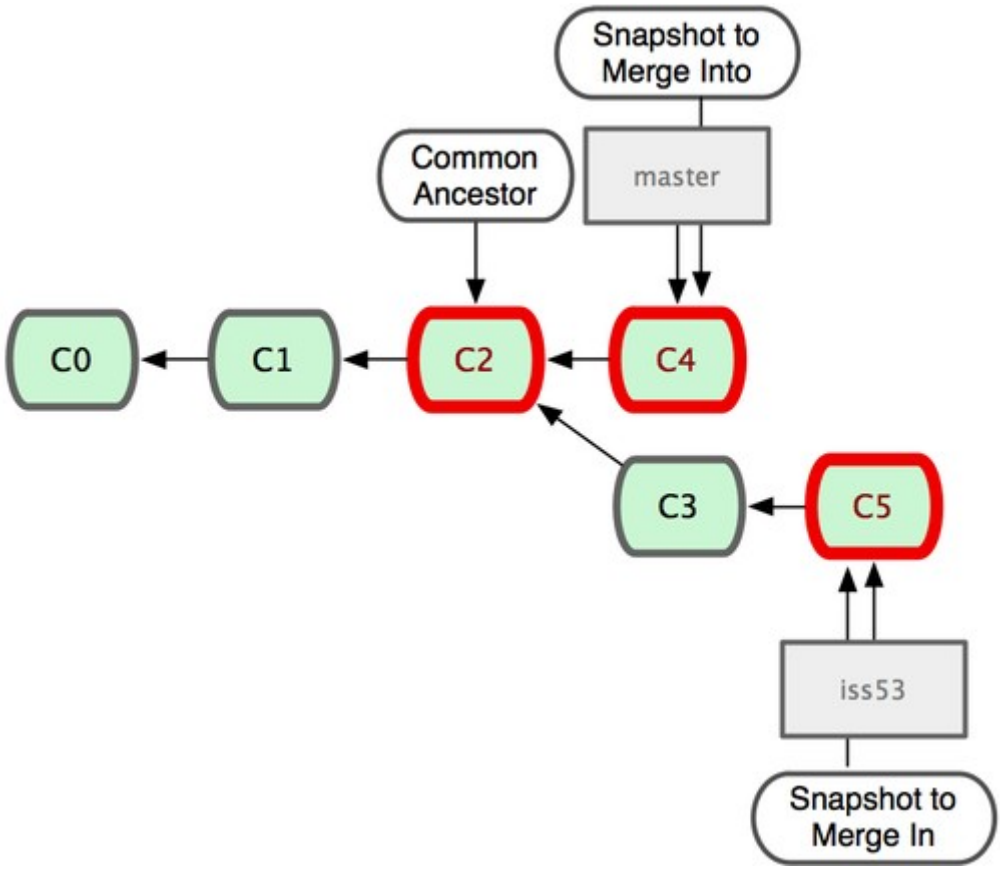

图 3-16. Git 为分支合并自动识别出最佳的同源合并点。

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）（见图 3-17）。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；这和 CVS 或 Subversion（1.5 以后的版本）不同，它们需要开发者手工指定合并基础。所以此特性让 Git 的合并操作比其他系统都要简单不少。

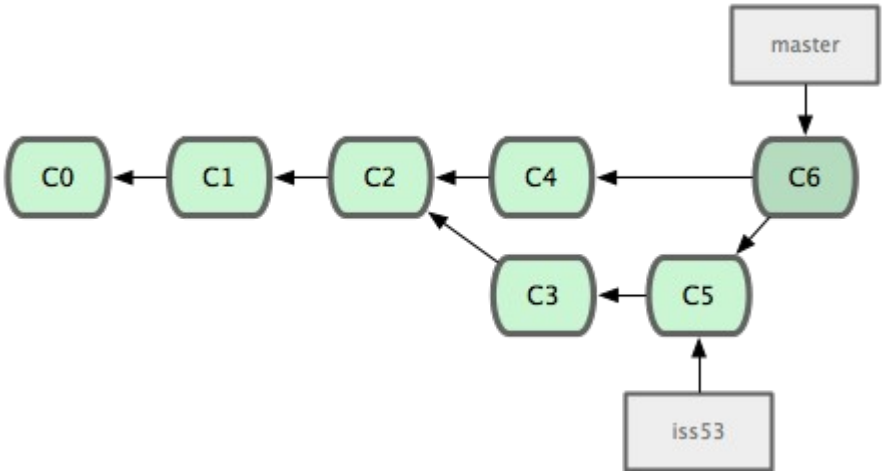

图 3-17. Git 自动创建了一个包含了合并结果的提交对象。

既然之前的工作成果已经合并到 `master` 了，那么 `iss53` 也就没用了。你可以就此删除它，并在问题追踪系统里关闭该问题。

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

遇到冲突时的分支合并

有时候合并操作并不会如此顺利。如果在不同的分支中都修改了同一个文件的同一部分，Git 就无法干净地把两者合到一起（译注：逻辑上说，这种问题只能由人来裁决。）。如果你在解决问题 #53 的过程中修改了 hotfix 中修改的部分，将得到类似下面的结果：

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

Git 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：

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add
```

```
   ..." to update what will be committed)
#   (use "git checkout --
```

```
   ..." to discard changes in working directory)
#
#       unmerged:   index.html
#
```

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:index.html

   contact : email.support@github.com

=======

 please contact us at support@github.com

>>>>>>> iss53:index.html
```

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。解决冲突的办法无非是二者选其一或者由你亲自整合到一起。比如你可以通过把这段内容替换为下面这样来解决：

```
please contact us at email.support@github.com
```

这个解决方案各采纳了两个分支中的一部分内容，而且我还删除了 **<<<<<<<**，**=======** 和 **>>>>>>>** 这些行。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 mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge
vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
 {local}: modified
 {remote}: modified
Hit return to start merge resolution tool (opendiff):
```

如果不想用默认的合并工具（Git 为我默认选择了 `opendiff`，因为我在 Mac 上运行了该命令），你可以在上方”merge tool candidates”里找到可用的合并工具列表，输入你想用的工具名。我们将在第七章讨论怎样改变环境中的默认值。

退出合并工具以后，Git 会询问你合并是否成功。如果回答是，它会为你把相关文件暂存起来，以表明状态为已解决。

再运行一次 `git status` 来确认所有冲突都已解决：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD
```

```
   ..." to unstage)
#
#       modified:   index.html
#
```

如果觉得满意了，并且确认所有冲突都已解决，也就是进入了暂存区，就可以用 `git commit` 来完成这次合并提交。提交的记录差不多是这样：

```
Merge branch 'iss53'

Conflicts:
 index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
#
```

如果想给将来看这次合并的人一些方便，可以修改该信息，提供更多合并细节。比如你都作了哪些改动，以及这么做的原因。有时候裁决冲突的理由并不直接或明显，有必要略加注解。

3.3 分支的管理

到目前为止，你已经学会了如何创建、合并和删除分支。除此之外，我们还需要学习如何管理分支，在日后的常规工作中会经常用到下面介绍的管理命令。

`git branch` 命令不仅仅能创建和删除分支，如果不加任何参数，它会给出当前所有分支的清单：

```
$ git branch
 iss53
* master
 testing
```

注意看 `master` 分支前的 `*` 字符：它表示当前所在的分支。也就是说，如果现在提交更新，`master` 分支将随着开发进度前移。若要查看各个分支最后一个提交对象的信息，运行 `git branch -v`：

```
$ git branch -v
 iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
 testing 782fd34 add scott to the author list in the readmes
```

要从该清单中筛选出你已经（或尚未）与当前分支合并的分支，可以用 `--merge` 和 `--no-merged` 选项（Git 1.5.6 以上版本）。比如用 `git branch --merge` 查看哪些分支已被并入当前分支（译注：也就是说哪些分支是当前分支的直接上游。）：

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

之前我们已经合并了 `iss53`，所以在这里会看到它。一般来说，列表中没有 `*` 的分支通常都可以用 `git branch -d` 来删掉。原因很简单，既然已经把它们所包含的工作整合到了其他分支，删掉也不会损失什么。

另外可以用 `git branch --no-merged` 查看尚未合并的工作：

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

它会显示还未合并进来的分支。由于这些分支中还包含着尚未合并进来的工作成果，所以简单地用 `git branch -d` 删除该分支会提示错误，因为那样做会丢失数据：

```
$ git branch -d testing
```

```
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

不过，如果你确实想要删除该分支上的改动，可以用大写的删除选项 `-D` 强制执行，就像上面提示信息中给出的那样。

3.4 利用分支进行开发的工作流程

现在我们已经学会了新建分支和合并分支，可以（或应该）用它来做点什么呢？在本节，我们会介绍一些利用分支进行开发的工作流程。而正是由于分支管理的便捷，才衍生出了这类典型的工作模式，你可以根据项目的实际情况选择一种用用看。

长期分支

由于 Git 使用简单的三方合并，所以就算在较长一段时间内，反复多次把某个分支合并到另一分支，也不是什么难事。也就是说，你可以同时拥有多个开放的分支，每个分支用于完成特定的任务，随着开发的推进，你可以随时把某个特性分支的成果并到其他分支中。

许多使用 Git 的开发者都喜欢用这种方式来开展工作，比如仅在 `master` 分支中保留完全稳定的代码，即已经发布或即将发布的代码。与此同时，他们还有一个名为 `develop` 或 `next` 的平行分支，专门用于后续的开发，或仅用于稳定性测试 — 当然并不是说一定要绝对稳定，不过一旦进入某种稳定状态，便可以把它合并到 `master` 里。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，并且不会引入更多错误之后，就可以并到主干分支中，等待下一次的发布。

本质上我们刚才谈论的，是随着提交对象不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支总是比较靠前（见图 3-18）。

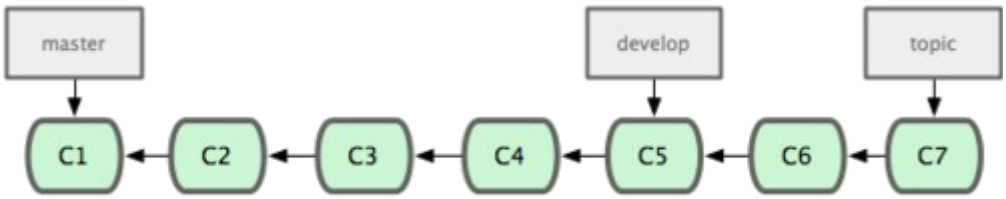

图 3-18. 稳定分支总是比较老旧。

或者把它们想象成工作流水线，或许更好理解一些，经过测试的提交对象集合被遴选到更稳定的流水线（见图 3-19）。

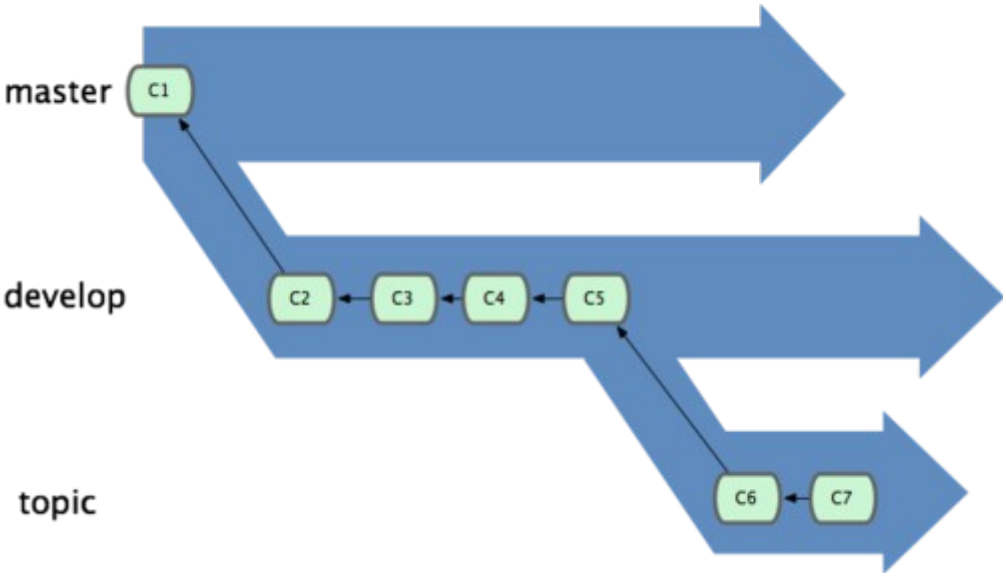

图 3-19. 想象成流水线可能会容易点。

你可以用这招维护不同层次的稳定性。某些大项目还会有个 proposed（建议）或 pu（proposed updates，建议更新）分支，它包含着那些可能还没有成熟到进入 next 或 master 的内容。这么做的目的是拥有不同层次的稳定性：当这些分支进入到更稳定的水平时，再把它们合并到更高层分支中去。再次说明下，使用多个长期分支的做法并非必需，不过一般来说，对于特大型项目或特复杂的项目，这么做确实更容易管理。

特性分支

在任何规模的项目中都可以使用特性（Topic）分支。一个特性分支是指一个短期的，用来实现单一特性或与其相关工作的分支。可能你在以前的版 本控 制系统里从未做过类似这样的事情，因为通常创建与合并分支消耗太大。然而在 Git 中，一天之内建立、使用、合并再删除多个分支是常见的事。

我们在上节的例子里已经见过这种用法了。我们创建了 iss53 和 hotfix 这两个特性分支，在提交了若干更新后，把它们合并到主干分支，然后删除。该技术允许你迅速且完全的进行语境切换 — 因为你的工作分散在不同的流水线里，每个分支里的改变都和它的目标特性相关，浏览代码之类的事情因而变得更简单了。你可以把作出的改变保持在特性分支中几 分钟，几天甚至几个月，等它们成熟以后再合并，而不用在乎它们建立的顺序或者进度。

现在我们来看一个实际的例子。请看图 3-20，由下往上，起先我们在 master 工作到 C1，然后开始一个新分支 iss91 尝试修复 91 号缺陷，提交到 C6 的时候，又冒出一个解决该问题的新办法，于是从之前 C4 的地方又分出一个分支 iss91v2，干到 C8 的时候，又回到主干 master 中提交了 C9 和 C10，再回到 iss91v2 继续工作，提交 C11，接着，又冒出个不太确定的想法，从 master 的最新提交 C10 处开了个新的分支 dumbidea 做些试验。

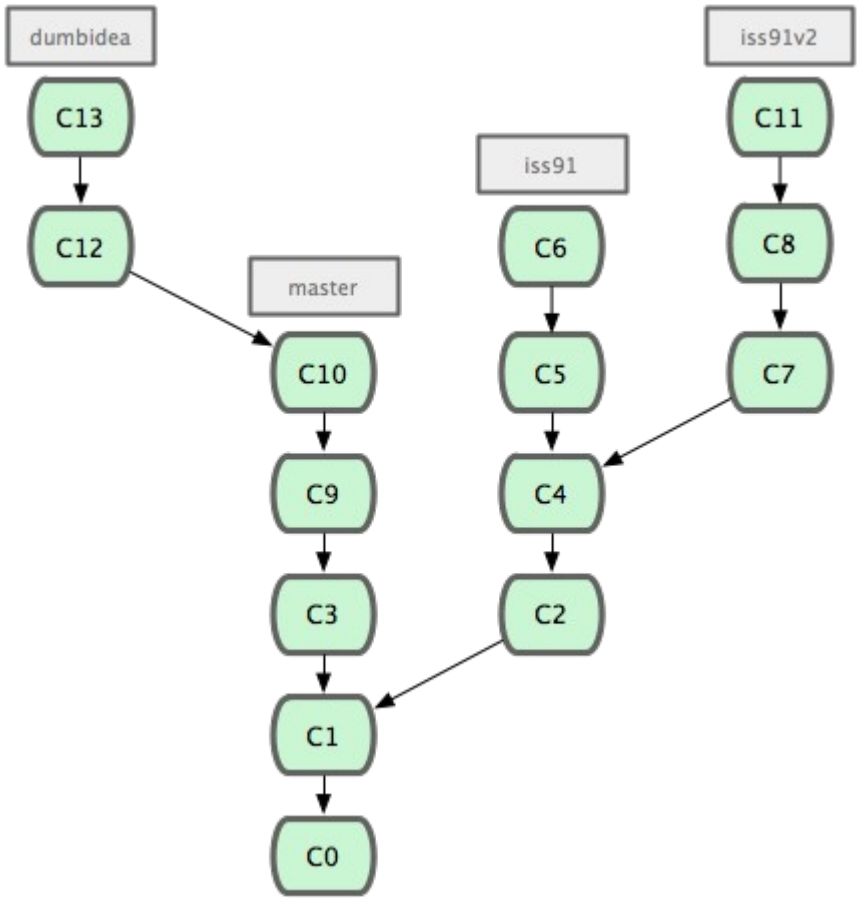

图 3-20. 拥有多个特性分支的提交历史。

现在，假定两件事情：我们最终决定使用第二个解决方案，即 iss91v2 中的办法；另外，我们把 dumbidea 分支拿给同事们看了以后，发现它竟然是个天才之作。所以接下来，我们准备抛弃原来的 iss91 分支（实际上会丢弃 C5 和 C6），直接在主干中并入另外两个分支。最终的提交历史将变成图 3-21 这样：

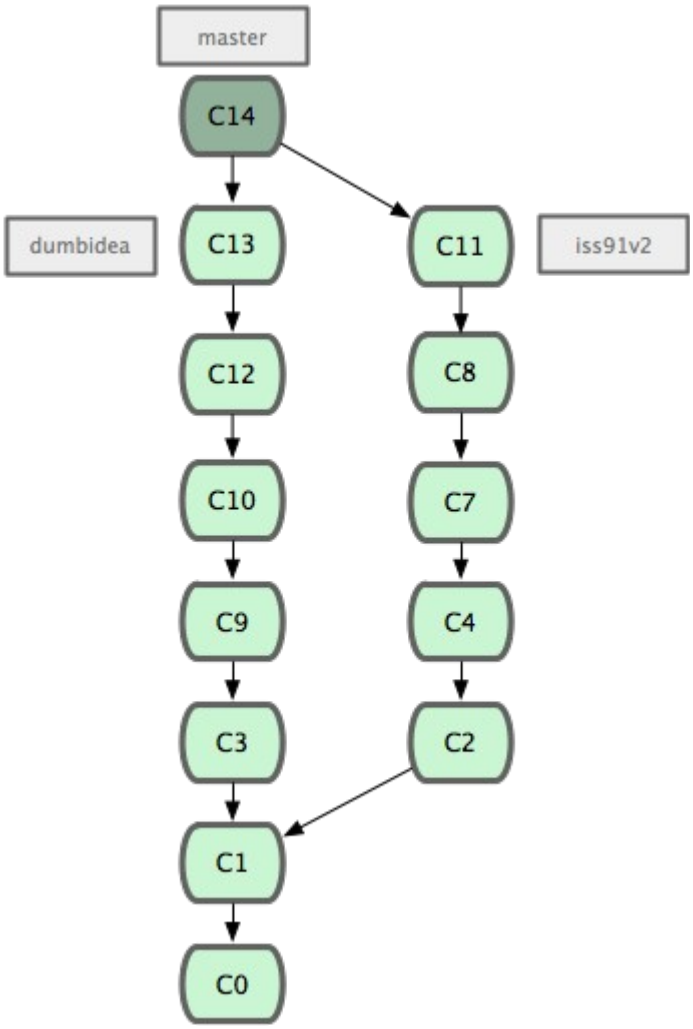

图 3-21. 合并了 dumbidea 和 iss91v2 后的分支历史。

请务必牢记这些分支全部都是本地分支，这一点很重要。当你在使用分支及合并的时候，一切都是在你自己的 Git 仓库中进行的 — 完全不涉及与服务器的交互。

3.5 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。

我们用 (远程仓库名)/(分支名) 这样的形式表示远程分支。比如我们想看看上次同 origin 仓库通讯时 master 的样子，就应该查看 origin/master 分支。如果你和同伴一起修复某个问题，但他们先推送了一个 iss53 分支到远程仓库，虽然你可能也有一个本地的 iss53 分支，但指向服务器上最新更新的却应该是 origin/iss53 分支。

可能有点乱，我们不妨举例说明。假设你们团队有个地址为 git.ourcompany.com 的 Git 服务器。如果你从这里克隆，Git 会自动为你将此远程仓库命名为 origin，并下载其中所有的数据，建立一个

指向它的 master 分支的指针，在本地命名为 origin/master，但你无法在本地更改其数据。接着，Git 建立一个属于你自己的本地 master 分支，始于 origin 上 master 分支相同的位置，你可以就此开始工作（见图 3-22）：

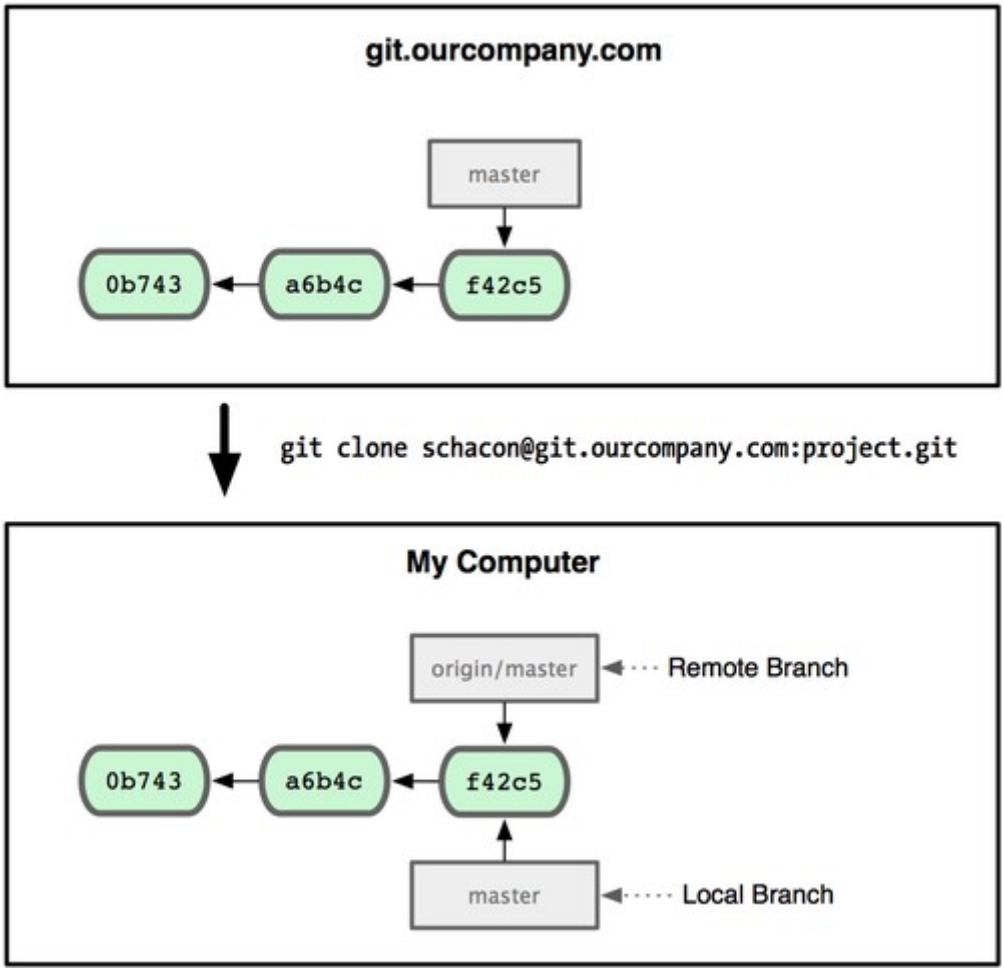

图 3-22. 一次 Git 克隆会建立你自己的本地分支 master 和远程分支 origin/master，它们都指向 origin/master 分支的最后一次提交。

如果你在本地 master 分支做了些改动，与此同时，其他人向 git.ourcompany.com 推送了他们的更新，那么服务器上的 master 分支就会向前推进，而于此同时，你在本地的提交历史正朝向不同方向发展。不过只要你不和服务器通讯，你的 origin/master 指针仍然保持原位不会移动（见图 3-23）。

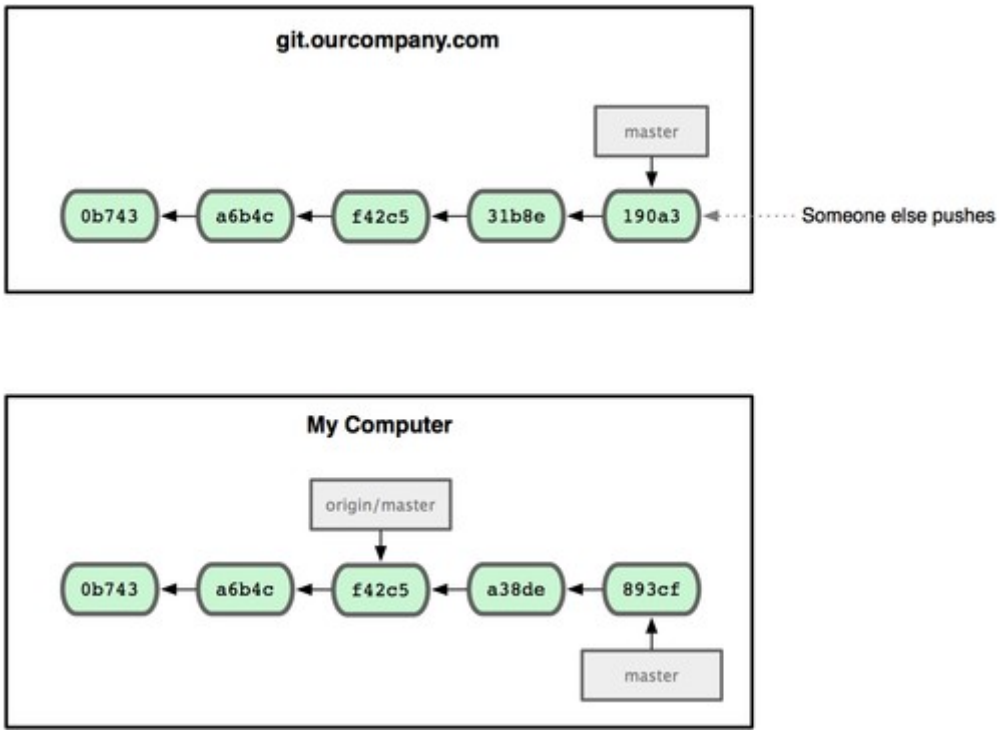

图 3-23. 在本地工作的同时有人向远程仓库推送内容会让提交历史开始分流。

可以运行 `git fetch origin` 来同步远程服务器上的数据到本地。该命令首先找到 `origin` 是哪个服务器（本例为`git.ourcompany.com`），从上面获取你尚未拥有的数据，更新你本地的数据库，然后把 `origin/master` 的指针移到它最新的位置上（见图 3-24）。

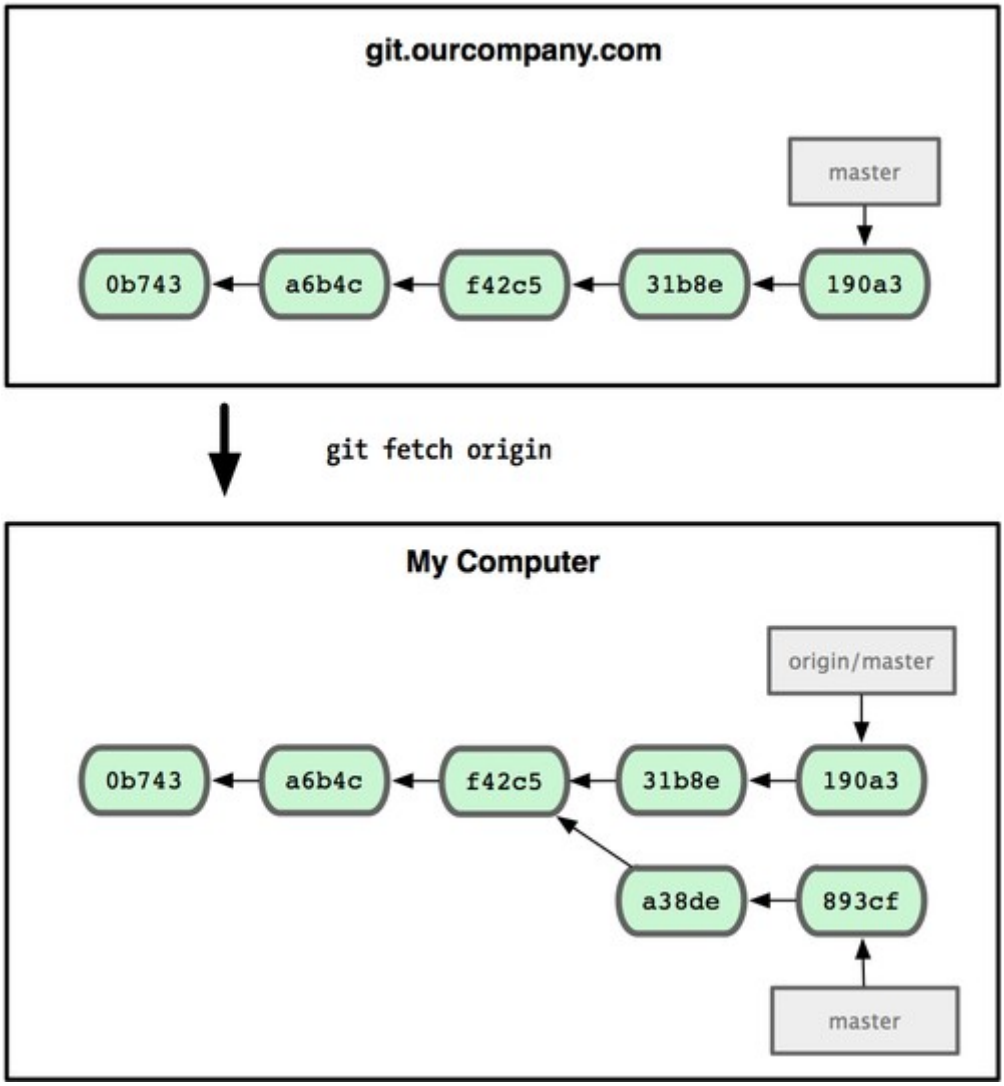

图 3-24. git fetch 命令会更新 remote 索引。

为了演示拥有多个远程分支（在不同的远程服务器上）的项目是如何工作的，我们假设你还有另一个仅供你的敏捷开发小组使用的内部服务器 `git.team1.ourcompany.com`。可以用第二章中提到的 `git remote add` 命令把它加为当前项目的远程分支之一。我们把它命名为 `teamone`，以便代替原始的 Git 地址（见图 3-25）。

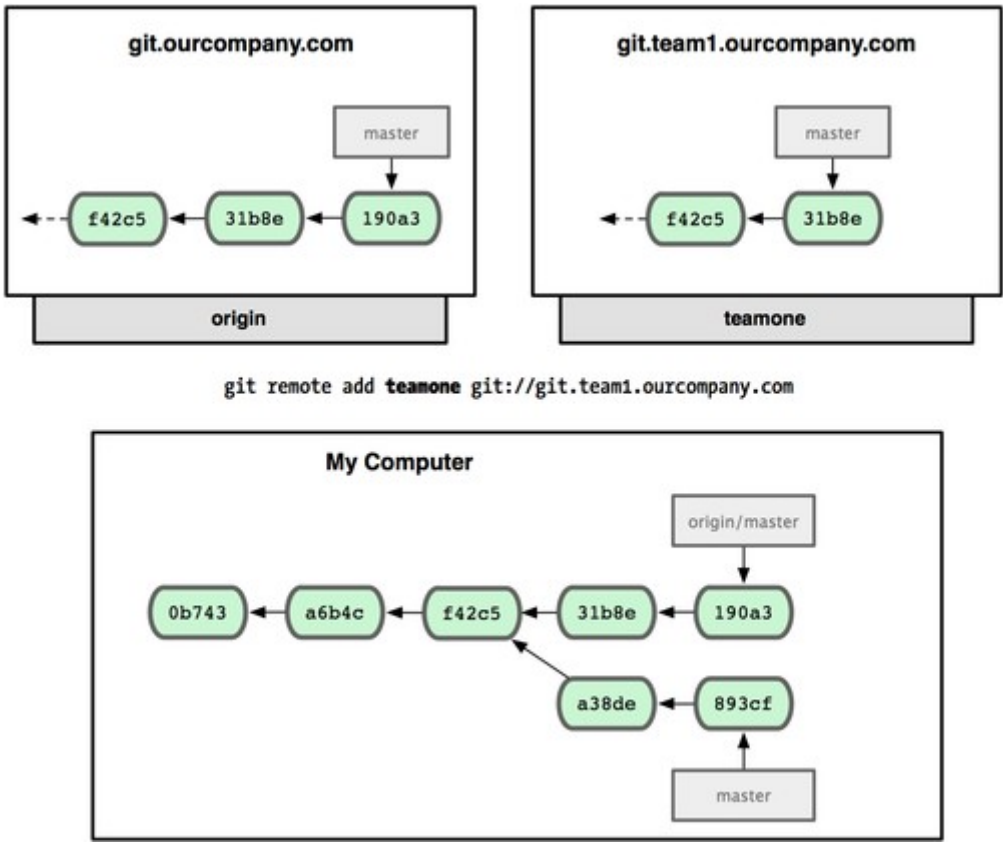

图 3-25. 把另一个服务器加为远程仓库

现在你可以用 `git fetch teamone` 来获取小组服务器上你还没有的数据了。由于当前该服务器上的内容是你 `origin` 服务器上的子集，Git 不会下载任何数据，而只是简单地创建一个名为 `teamone/master` 的分支，指向 `teamone` 服务器上 `master` 分支所在的提交对象 `31b8e`（见图 3-26）。

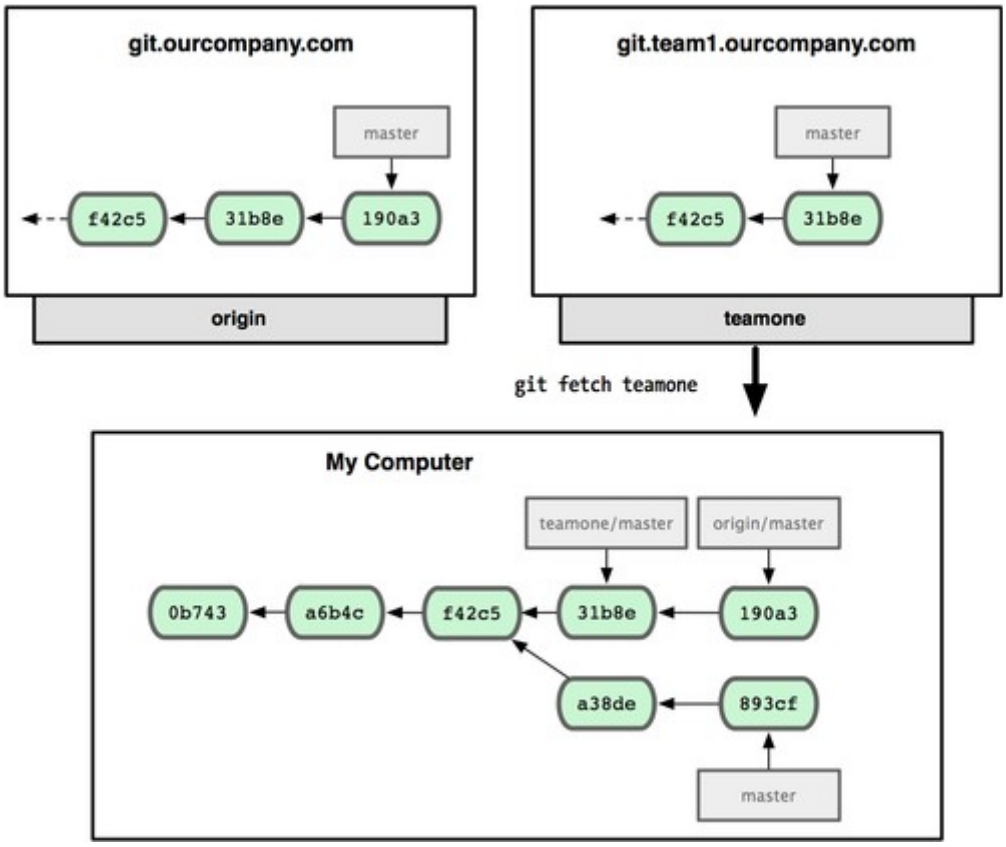

图 3-26. 你在本地有了一个指向 teamone 服务器上 master 分支的索引。

推送本地分支

要想和其他人分享某个本地分支，你需要把它推送到一个你拥有写权限的远程仓库。你的本地分支不会被自动同步到你引入的远程服务器上，除非你明确执行推送操作。换句话说，对于无意分享的分支，你尽管保留为私人分支好了，而只推送那些协同工作要用到的特性分支。

如果你有个叫 serverfix 的分支需要和他人一起开发，可以运行 git push (远程仓库名) (分支名)：

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

这其实有点像条捷径。Git 自动把 serverfix 分支名扩展为 refs/heads/serverfix:refs/heads/serverfix，意为“取出我在本地的 serverfix 分支，推送到远程仓库的 serverfix 分支中去”。我们将在第九章进一步介绍 refs/heads/ 部分的细节，不过一般使用的时候都可以省略它。也可以运行 git push origin serverfix:serferfix 来实现相同的效果，它的意思是“上传我本地的 serverfix 分支到远程仓库中去，仍旧称它为 serverfix 分支”。通过此语法，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。

接下来，当你的协作者再次从服务器上获取数据时，他们将得到一个新的远程分支 origin/serverfix：

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

值得注意的是，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，在本例中，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 refs/remotes/origin/serverfix.
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

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。反过来，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。当然，你可以随心所欲地设定为其它跟踪分支，比如 origin 上除了 master 之外的其它分支。刚才我们已经看到了这样的一个例子：git checkout -b [分支名] [远程名]/[分支名]。如果你有 1.6.2 以上版本的 Git，还可以用--track 选项简化：

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

要为本地分支设定不同于远程分支的名字，只需在前个版本的命令里换个名字：

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

现在你的本地分支 sf 会自动向 origin/serverfix 推送和抓取数据了。

删除远程分支

如果不再需要某个远程分支了，比如搞定了某个特性并把它合并进了远程的 master 分支（或任何其他存放稳定代码的地方），可以用这个非常无厘头的语法来删除它：git push [远程名] :[分支名]。如果想在服务器上删除 serverfix 分支，运行下面的命令：

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

咚！服务器上的分支没了。你最好特别留心这一页，因为你一定会用到那个命令，而且你很可能会忘掉它的语法。有种方便记忆这条命令的方法：记住我们不久前见过的 `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]`”。

3.6 分支的衍合

把一个分支整合到另一个分支的办法有两种：`merge` 和 `rebase`（译注：`rebase` 的翻译暂定为“衍合”，大家知道就可以了。）。在本章我们会学习什么是衍合，如何使用衍合，为什么衍合操作如此富有魅力，以及我们应该在什么情况下使用衍合。

基本的衍合操作

请回顾之前有关合并的一节（见图 3-27），你会看到开发进程分叉到两个不同分支，又各自提交了更新。

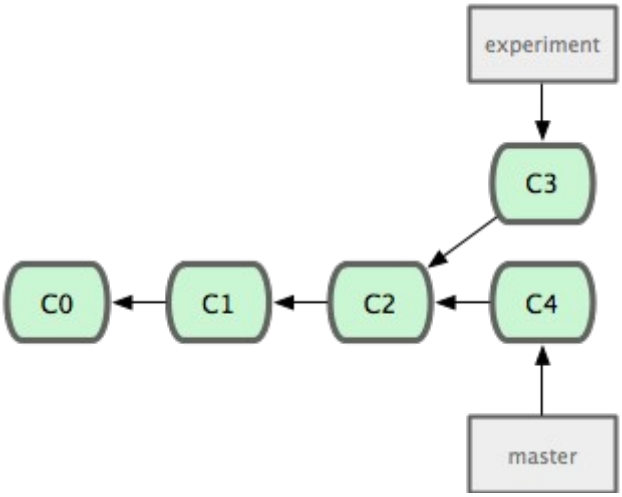

图 3-27. 最初分叉的提交历史。

之前介绍过，最容易的整合分支的方法是 `merge` 命令，它会把两个分支最新的快照（C3 和 C4）以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 3-28 所示：

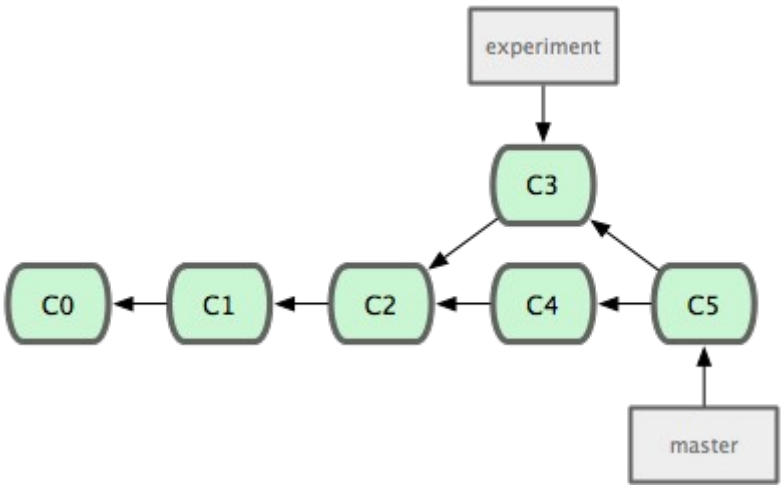

图 3-28. 通过合并一个分支来整合分叉了的历史。

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。

在上面这个例子中，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 3-29 所示：

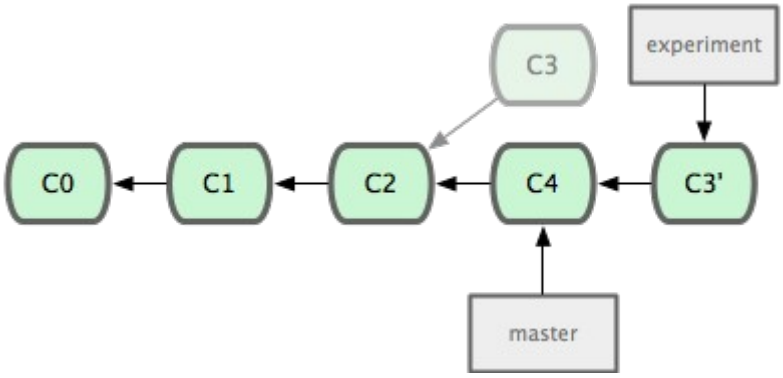

图 3-29. 把 C3 里产生的改变到 C4 上重演一遍。

现在回到 master 分支，进行一次快进合并（见图 3-30）：

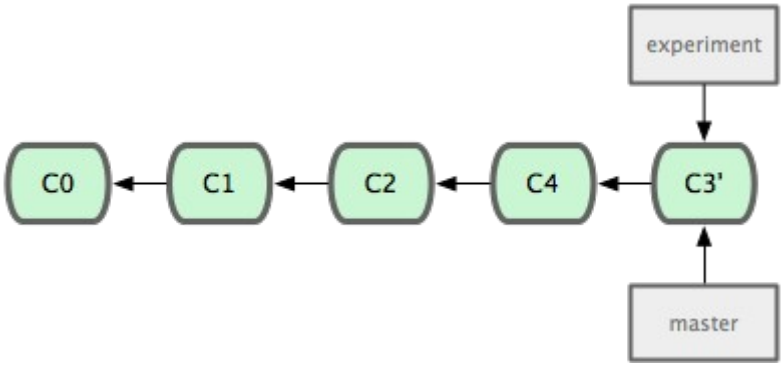

图 3-30. master 分支的快进。

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。

有趣的衍合

衍合也可以放到其他分支进行，并不一定非得根据分化之前的分支。以图 3-31 的历史为例，我们为了给服务器端代码添加一些功能而创建了特性分支 server，然后提交 C3 和 C4。然后又从 C3 的地方再增加一个 client 分支来对客户端代码进行一些相应修改，所以提交了 C8 和 C9。最后，又回到 server 分支提交了 C10。

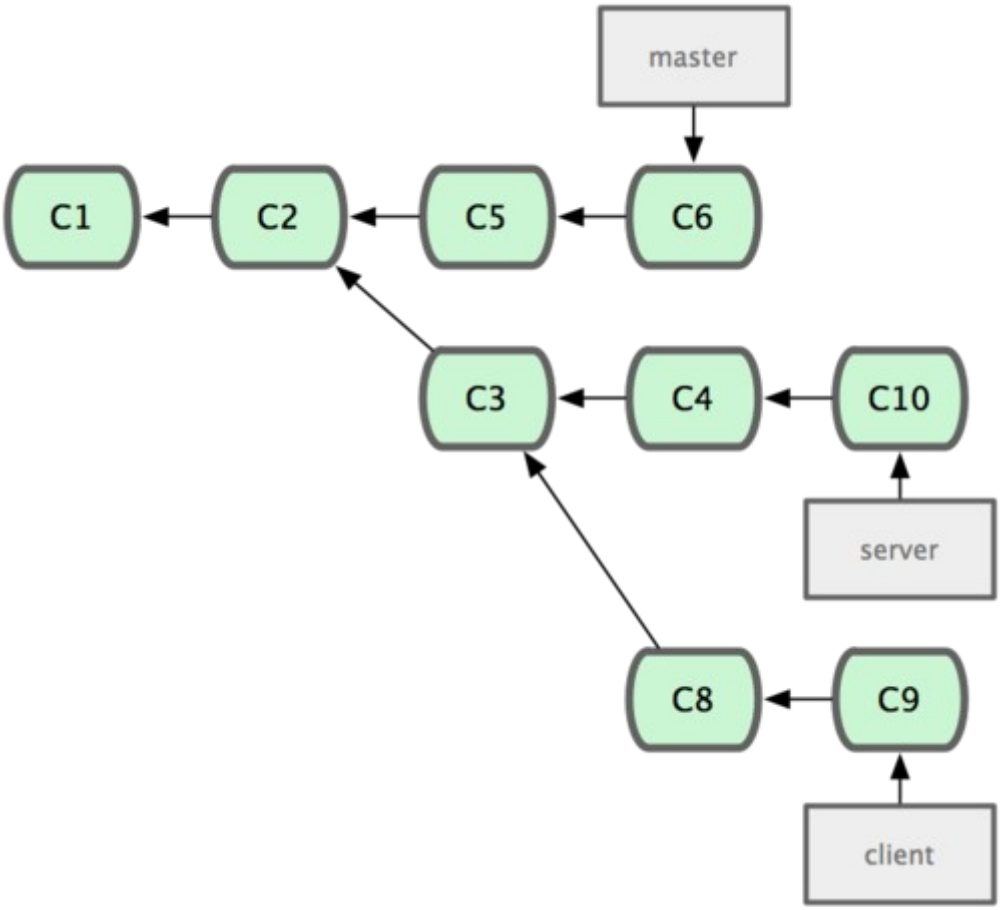

图 3-31. 从一个特性分支里再分出一个特性分支的历史。

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 server 分支而非 master 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
```

这好比在说：“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”。是不是有点复杂？不过它的结果如图 3-32 所示，非常酷（译注：虽然 client 里的 C8, C9 在 C3 之后，但这仅表明时间上的先后，而非在 C3 修改的基础上进一步改动，因为 server 和 client 这两个分支对应的代码应该是两套文件，虽然这么说不是很严格，但应理解为在 C3 时间点之后，对另外的文件所做的 C8，C9 修改，放到主干重演。）：

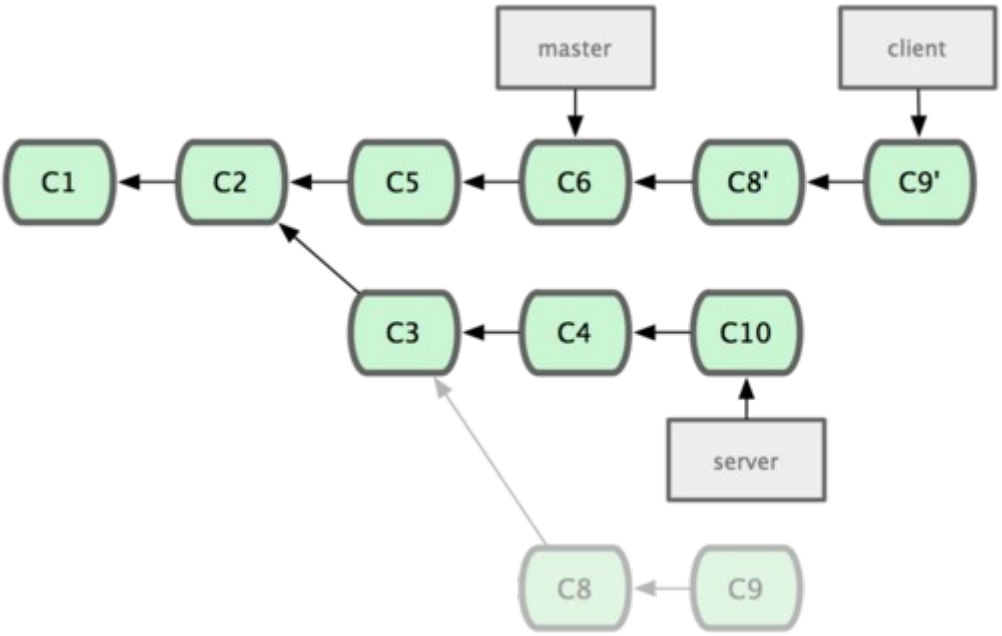

图 3-32. 将特性分支上的另一个特性分支衍合到其他分支。

现在可以快进 master 分支了（见图 3-33）：

```
$ git checkout master
$ git merge client
```

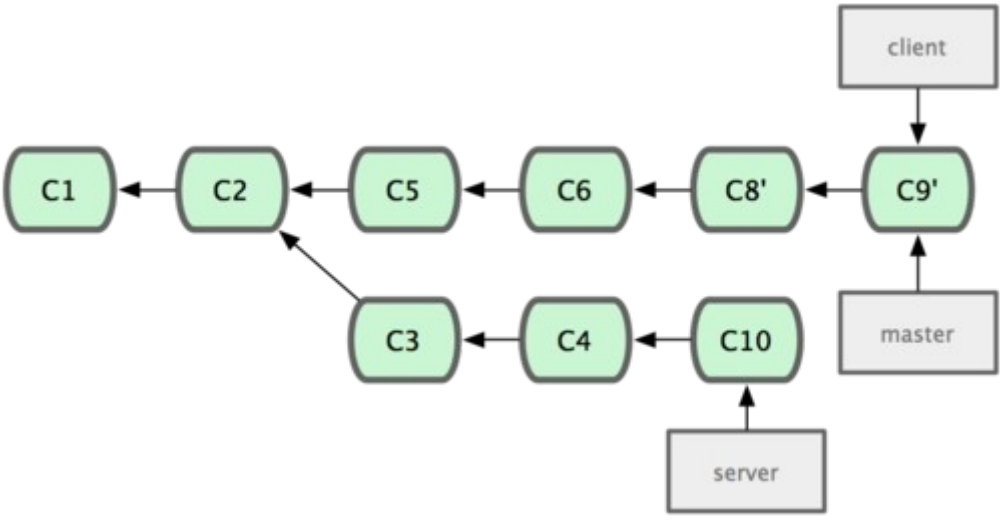

图 3-33. 快进 master 分支，使之包含 client 分支的变化。

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
```

于是，server 的进度应用到 master 的基础上，如图 3-34 所示：

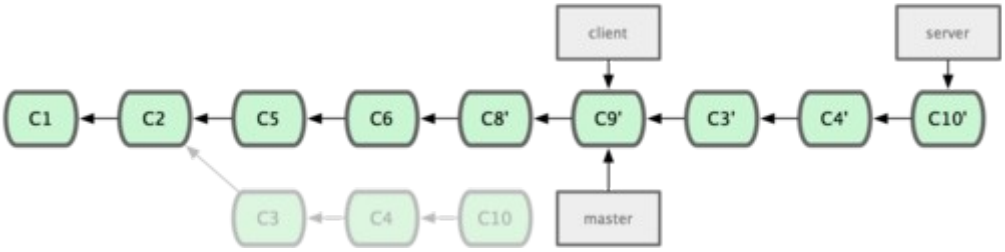

图 3-34. 在 master 分支上衍合 server 分支。

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 3-35 的样子：

```
$ git branch -d client
$ git branch -d server
```

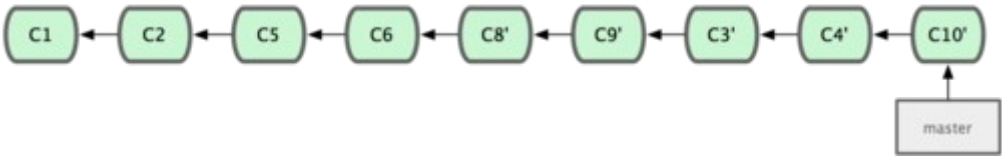

图 3-35. 最终的提交历史

衍合的风险

呃，奇妙的衍合也并非完美无缺，要用它得遵守一条准则：

一旦分支中的提交对象发布到公共仓库，就千万不要对该分支进行衍合操作。

如果你遵循这条金科玉律，就不会出差错。否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

在进行衍合的时候，实际上抛弃了一些现存的提交对象而创造了一些类似但不同的新的提交对象。如果你把原来分支中的提交对象发布出去，并且其他人更新下载后在其基础上开展工作，而稍后你又用 git rebase 抛弃这些提交对象，把新的重演后的提交对象发布出去的话，你的合作者就不得不重新合并他们的工作，这样当你再次从他们那里获取内容时，提交历史就会变得一团糟。

下面我们用一个实际例子来说明为什么公开的衍合会带来问题。假设你从一个中央服务器克隆然后在它的基础上搞了一些开发，提交历史类似图 3-36 所示：

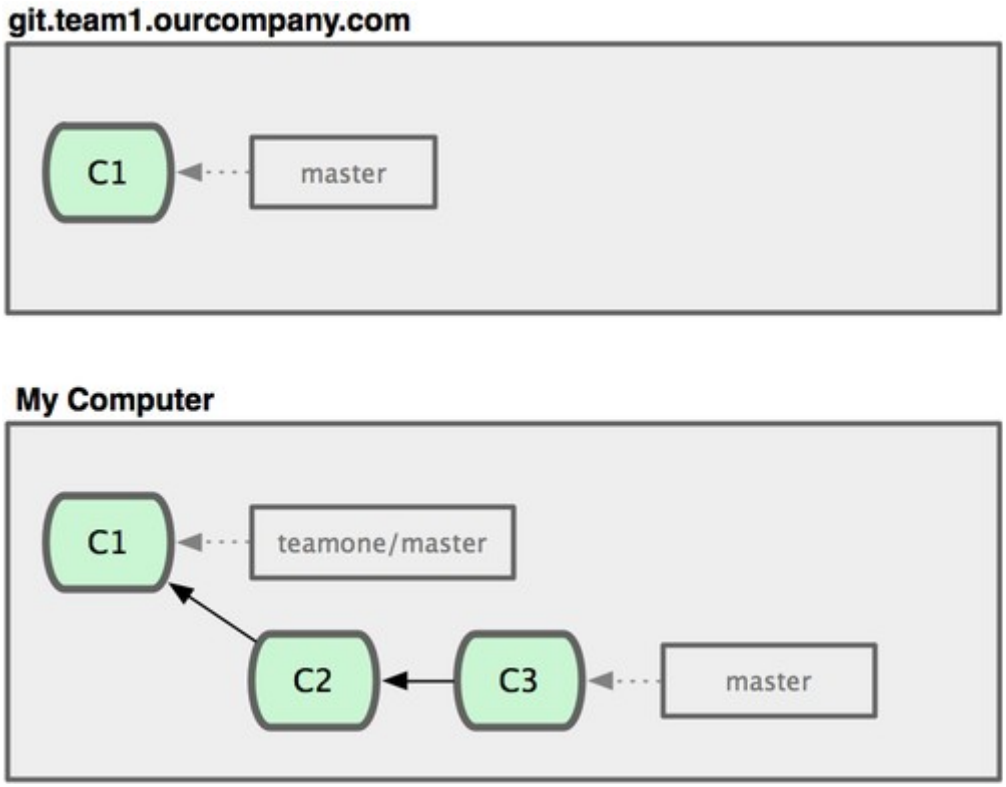

图 3-36. 克隆一个仓库，在其基础上工作一番。

现在，某人在 C1 的基础上做了些改变，并合并他自己的分支得到结果 C6，推送到中央服务器。当你抓取并合并这些数据到你本地的开发分支中后，会得到合并结果 C7，历史提交会变成图 3-37 这样：

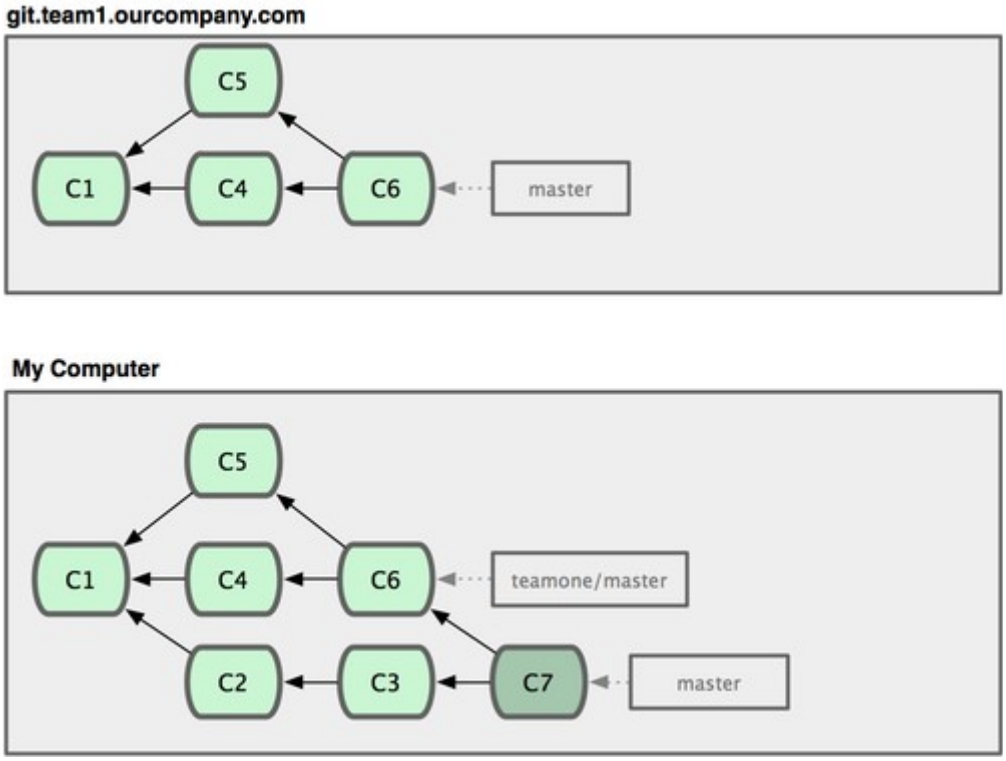

图 3-37. 抓取他人提交，并入自己主干。

接下来，那个推送 C6 上来的人决定用衍合取代之前的合并操作；继而又用 `git push --force` 覆盖了服务器上的历史，得到 C4'。而之后当你再从服务器上下载最新提交后，会得到：

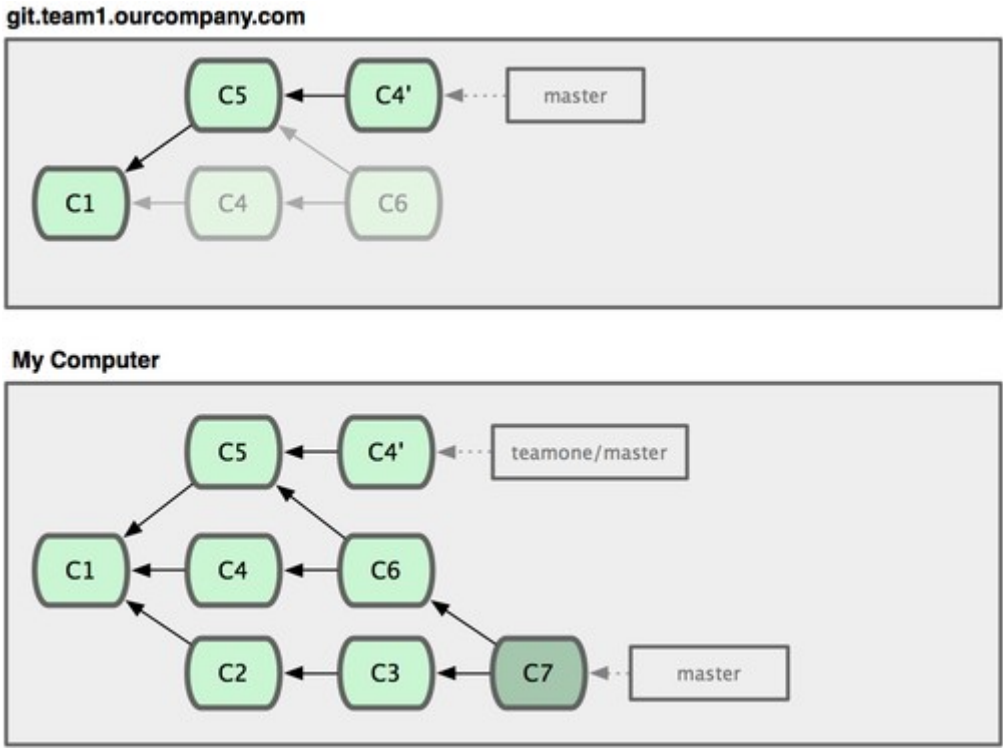

图 3-38. 有人推送了衍合后得到的 C4’，丢弃了你作为开发基础的 C4 和 C6。

下载更新后需要合并，但此时衍合产生的提交对象 C4’ 的 SHA-1 校验值和之前 C4 完全不同，所以 Git 会把它们当作新的提交对象处理，而实际上此刻你的提交历史 C7 中早已经包含了 C4 的修改内容，于是合并操作会把 C7 和 C4’ 合并为 C8（见图 3-39）：

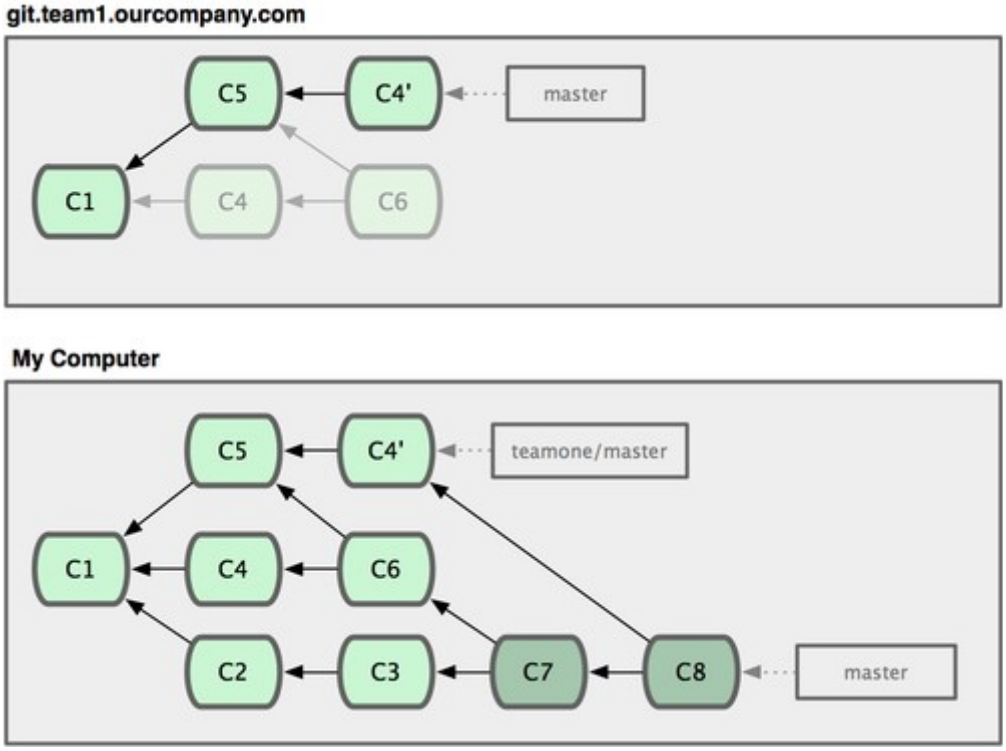

图 3-39. 你把相同的内容又合并了一遍，生成一个新的提交 C8。

C8 这一步的合并是迟早会发生的，因为只有这样你才能和其他协作者提交的内容保持同步。而在 C8 之后，你的提交历史里就会同时包含 C4 和 C4’，两者有着不同的 SHA-1 校验值，如果用 `git log` 查看历史，会看到两个提交拥有相同的作者日期与说明，令人费解。而更糟的是，当你把这样的历

史推送到服务器后，会再次把这些衍合后的提交引入到中央服务 器，进一步困扰其他人（译注：这个例子中，出问题的责任方是那个发布了 C6 后又用衍合发布 C4′ 的人，其他人会因此反馈双重历史到共享主干，从而混淆大家的视听。）。

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。

3.7 小结

读到这里，你应该已经学会了如何创建分支并切换到新分支，在不同分支间转换，合并本地分支，把分支推送到共享服务器上，使用共享分支与他人协作，以及在分享之前进行衍合。