

个人资料



Jack_CJ

+ 关注

发私信

访问：5782次

积分：792

等级：

BLDG

3

排名：千里之外

文章搜索



文章分类

Linux (110)

windows (2)

3D (1)

文章存档

2016年10月 (59)

2016年09月 (59)

阅读排行

- 怎么用树莓派和 iPad Pro ... (183)
- Docker 基础技术：Linux N... (163)
- 最实用的30个Linux命令！ (162)
- 掌握 Linux PC 性能之基准... (141)
- 教你摸清 Linux PC 的性能... (114)
- Git秘籍:在 Git 中进行版本回退 (114)
- 提升服务效率就这么简单 (97)
- 百度运用 FPGA 方法大规模... (96)
- Spark不是唯一，三种新兴的... (92)
- SSH如何通过公钥连接云服... (87)

评论排行

- 分享记录我的Linux系统入门... (0)
- 在Linux用户空间做内核空间... (0)
- 实战Centos系统部署Codis... (0)
- Git秘籍:在 Git 中进行版本回退 (0)
- 常用的Git Tips (0)
- 霸气！Nginx 中缓存静态文... (0)
- Linux下6种优秀的邮件传输... (0)
- smem - Linux 内存监视软件 (0)
- 在 Ubuntu 16.04 上安装 L... (0)
- Docker 容器测试全探索 (0)

【1024程序员节】获奖结果公布 【观点】有了深度学习，你还学传统机器学习算法么？ 【资源库】火爆了的React Native都在研究什么

原 Git分支的前世今生

标签：Linux Git

2016-10-29 14:43

387人阅读

评论(0)

收藏

举报

分类：Linux (109)

版权声明：本文为博主原创文章，未经博主允许不得转载。

导读

几乎所有的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把你的工作从开发主线上分离开来，以免影响开发主线。在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。



1.1 Git 分支 - 分支简介

有人把 Git 的分支模型称为它的“必杀技特性”，也正因为这一特性，使得 Git 从众多版本控制系统中脱颖而出。为何 Git 的分支模型如此出众呢？Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。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

为了真正理解 Git 处理分支的方式，我们需要回顾一下 Git 是如何保存数据的。

或许你还记得 **起步** 的内容，Git 保存的不是文件的变化或者差异，而是一系列不同时刻的文件快照。

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，我们可以很自然的想到——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、提交时输入的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，普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象，

为了说得更加形象，我们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。暂存操作会为每一个文件计算校验和（使用我们在 **起步** 中提到的 SHA-1 哈希算法），然后会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），最终将校验和加入到暂存区域等待提交：

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

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为树对象。随后，Git 便会创建一个提交对象，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。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）。

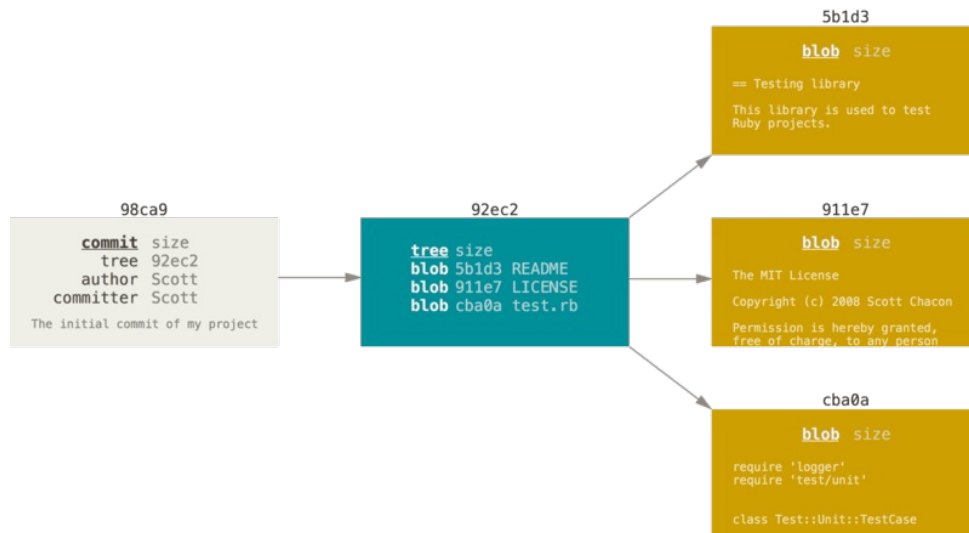


Figure 3-1. 首次提交对象及其树结构做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。

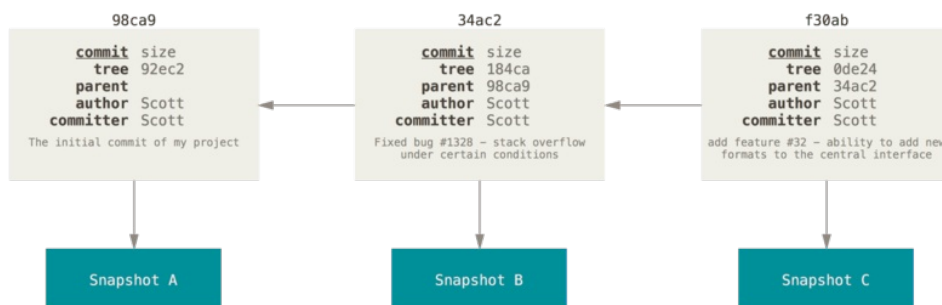


Figure 3-2. 提交对象及其父对象Git 的分支，其实本质上仅仅是指向提交对象的可变指针。Git 的默认分支名字是 master。在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。它会在每次的提交操作中自动向前移动。

NOTE

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。

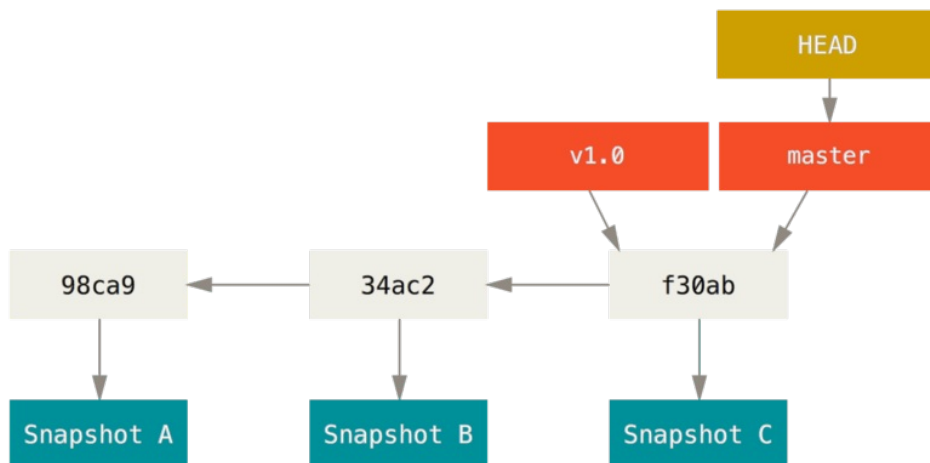


Figure 3-3. 分支及其提交历史

分支创建

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：

```
git branch testing
```

这会在当前所在的提交对象上创建一个指针。

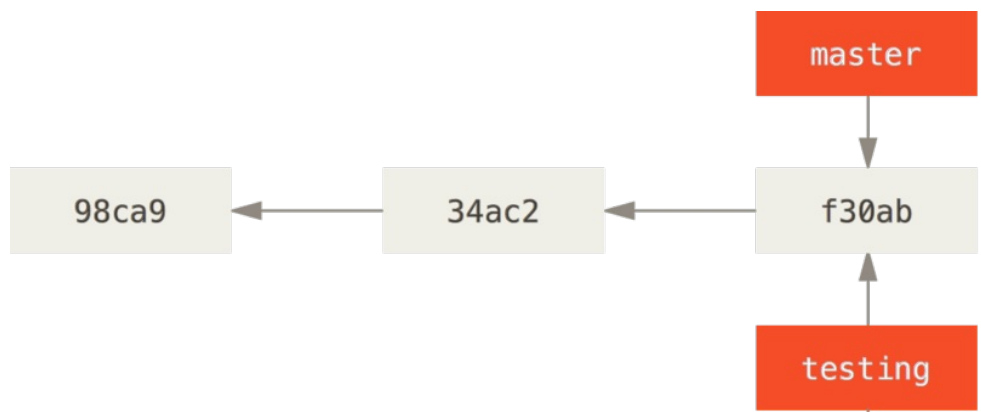


Figure 3-4. 两个指向相同提交历史的分支那么，Git 又是怎么知道当前在哪个分支上呢？也很简单，它有一个名为 HEAD 的特殊指针。请注意它和许多其它版本控制系统（如 Subversion 或 CVS）里的 HEAD 概念完全不同。在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 HEAD 想象为当前分支的别名）。在本例中，你仍然在 master 分支上。因为 git branch 命令仅仅创建一个新分支，并不会自动切换到新分支中去。

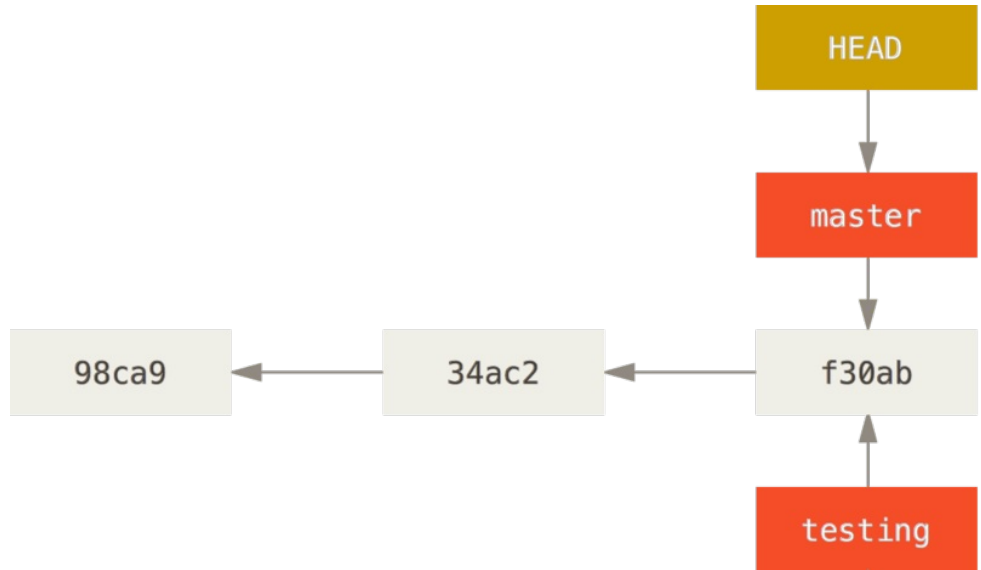


Figure 3-5. HEAD 指向当前所在的分支你可以简单地使用 git log 命令查看各个分支当前所指的提交对象。提供这一功能的参数是 --decorate。

```
git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

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：

```
git checkout testing
```

这样 HEAD 就指向 testing 分支了。

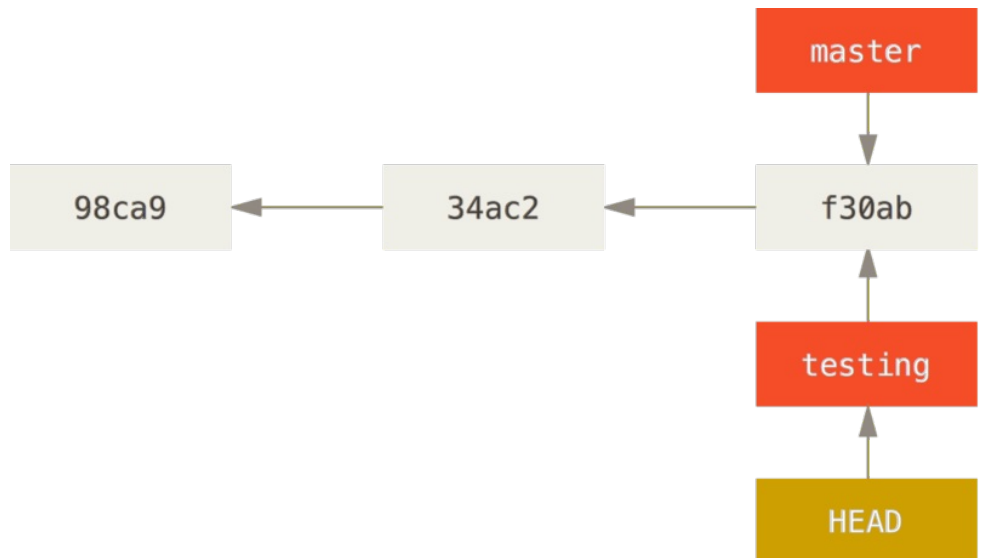


Figure 3-6. HEAD 指向当前所在的分支那么，这样的实现方式会给我们带来什么好处呢？现在不妨再提交一次：

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

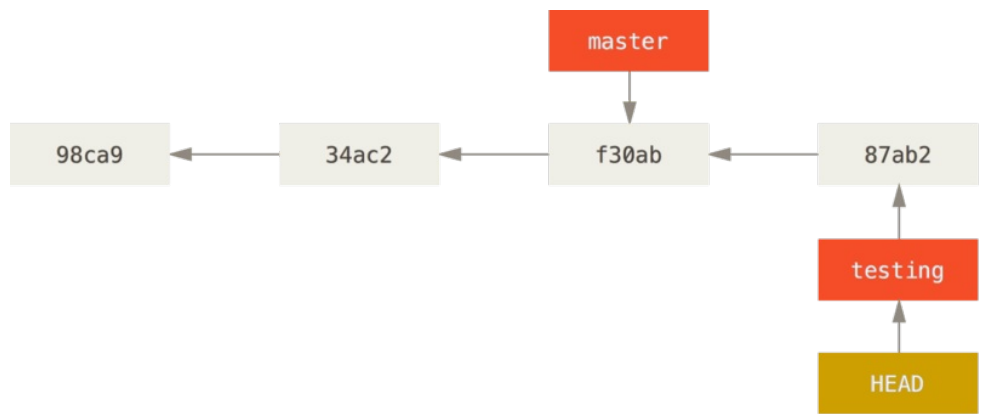


Figure 3-7. HEAD 分支随着提交操作自动向前移动如图所示，你的testing分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。这就有意思了，现在我们切换回 master 分支看看：

```
git checkout master
```

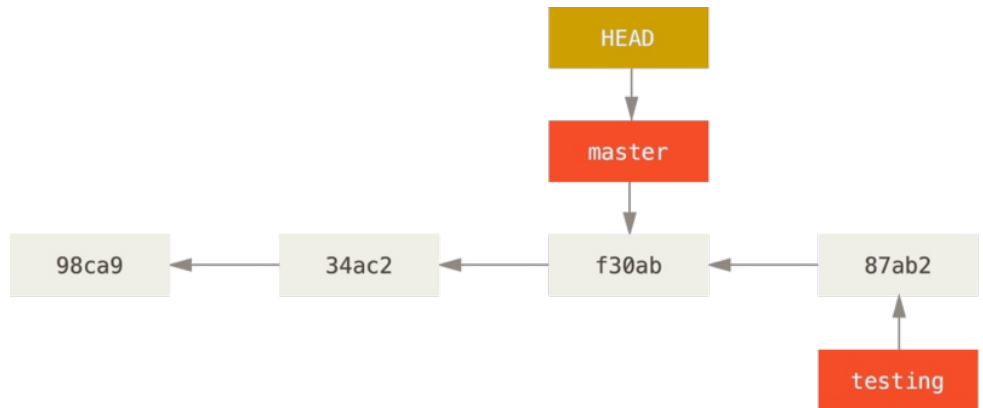


Figure 3-8. 检出时 HEAD 随之移动这条命令做了两件事。一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。也就是说，你现在做修改的话，项目将始于一个较旧的版本。本质上讲，这就是忽略testing 分支所做的修改，以便于向另一个方向进行开发。

分支切换会改变你工作目录中的文件

在切换分支时，一定要注意你工作目录里的文件会被改变。如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。如果 Git 不能干净利落地完成这个任务，它将禁止切换分支。

我们不妨再稍微做些修改并提交：

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

现在，这个项目的提交历史已经产生了分叉（参见 Figure 3-9）。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。

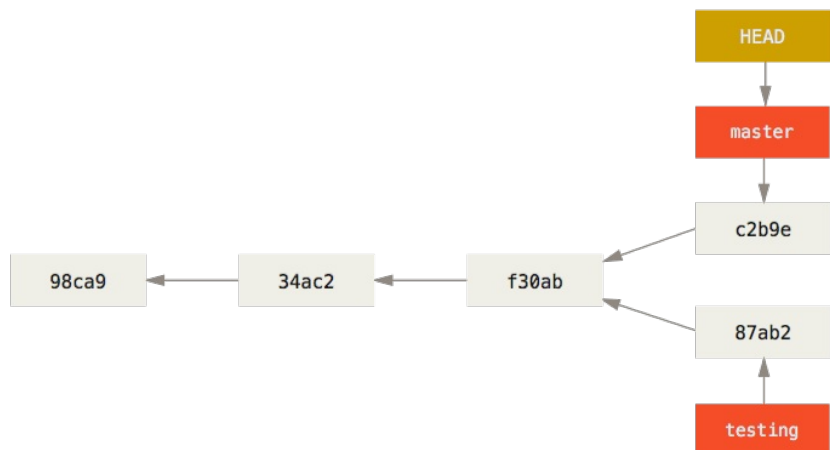


Figure 3-9. 项目分叉历史你可以简单地使用 git log 命令查看分叉历史。运行 git log --oneline --decorate --graph --all，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。

```
$ git log --oneline --decorate --graph --all
```

```
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

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。创建一个新分支就像是往一个文件中写入 41 个字节（40 个字符和 1 个换行符），如此的简单能不快吗？

这与过去大多数版本控制系统形成了鲜明的对比，它们在创建分支时，将所有的项目文件都复制一遍，并保存到一个特定的目录。完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。而在 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。

接下来，让我们看看为什么你应该这么做？

Git 分支 - 分支的新建与合并

分支的新建与合并

让我们来看一个简单的分支新建与分支合并的例子，实际工作中你可能会用到类似的工作流。你将经历如下步骤：

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

正此时，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：

- 切换到你的线上分支（production branch）。
- 为这个紧急任务新建一个分支，并在其中修复它。
- 在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到线上分支。
- 切换回你最初工作的分支上，继续工作。

新建分支

首先，我们假设你正在你的项目上工作，并且已经有一些提交。

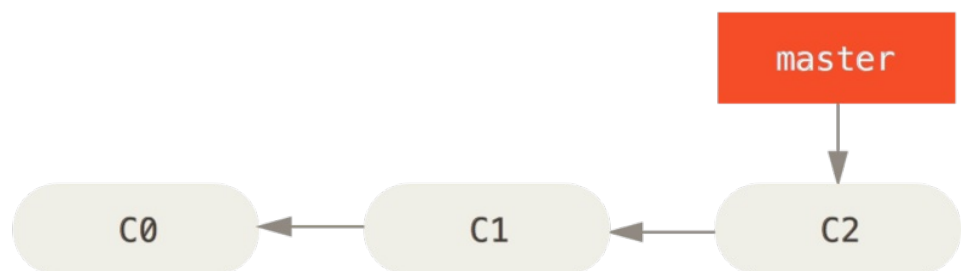


Figure 3-10. 一个简单提交历史现在，你已经决定要解决你的公司使用的问题追踪系统中的 #53 问题。想要新建一个分支并同时切换到那个分支上，你可以运行一个带有 -b 参数的 git checkout 命令：

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

它是下面两条命令的简写：

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

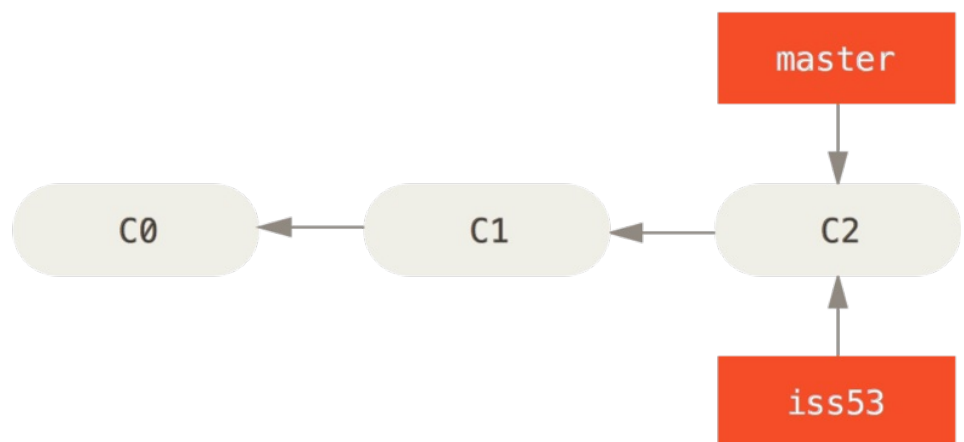


Figure 3-11. 创建一个新分支指针你继续在 #53 问题上工作，并且做了一些提交。在此过程中，iss53 分支在不断的向前推进，因为你已经检出到该分支（也就是说，你的 HEAD 指针指向了 iss53 分支）

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

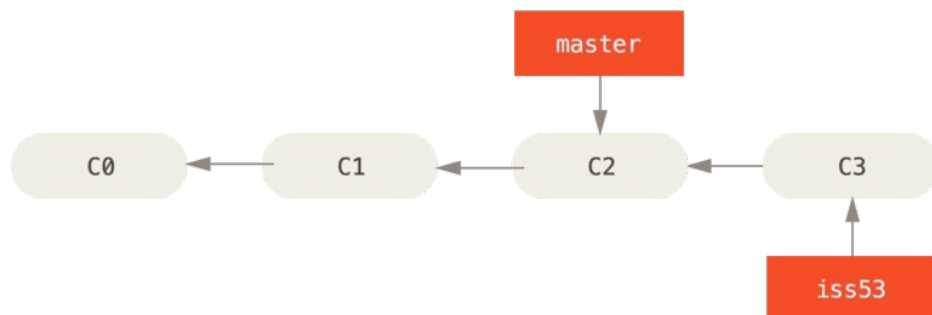


Figure 3-12. iss53 分支随着工作的进展向前推进现在你接到那个电话，有个紧急问题等待你来解决。有了 Git 的帮助，你不必把这个紧急问题和 iss53 的修改混在一起，你也不需要花大力气来还原关于 53# 问题的修改，然后再添加关于这个紧急问题的修改，最后将这个修改提交到线上分支。你所要做的仅仅是切换回 master 分支。

但是，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（即，保存进度（stashing）和 修补提交（commit amending）），我们会在 **储藏与清理** 中看到关于这两个命令的介绍。现在，我们假设你已经把你的修改全部提交了，这时你可以切换回 master 分支了：

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

这个时候，你的工作目录和你在开始 #53 问题之前一模一样，现在你可以专心修复紧急问题了。请记住：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。

接下来，你要修复这个紧急问题。让我们建立一个针对该紧急问题的分支（hotfix branch），在该分支上工作直到问题解决：

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix lfb7853] fixed the broken email address
1 file changed, 2 insertions(+)
```

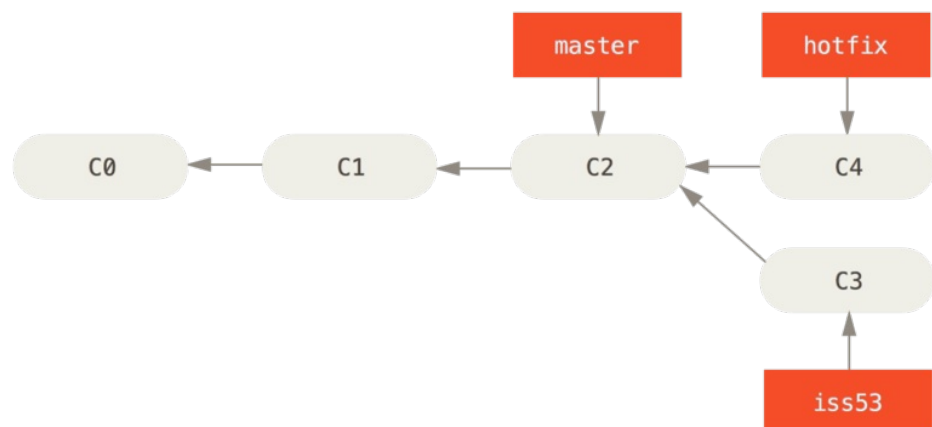


Figure 3-13. 基于 master 分支的紧急问题分支 hotfix branch 你可以运行你的测试，确保你的修改是正确的，然后将其合并回你的 master 分支来部署到线上。你可以使用 git merge 命令来达到上述目的：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
1 file changed, 2 insertions(+)
```

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。

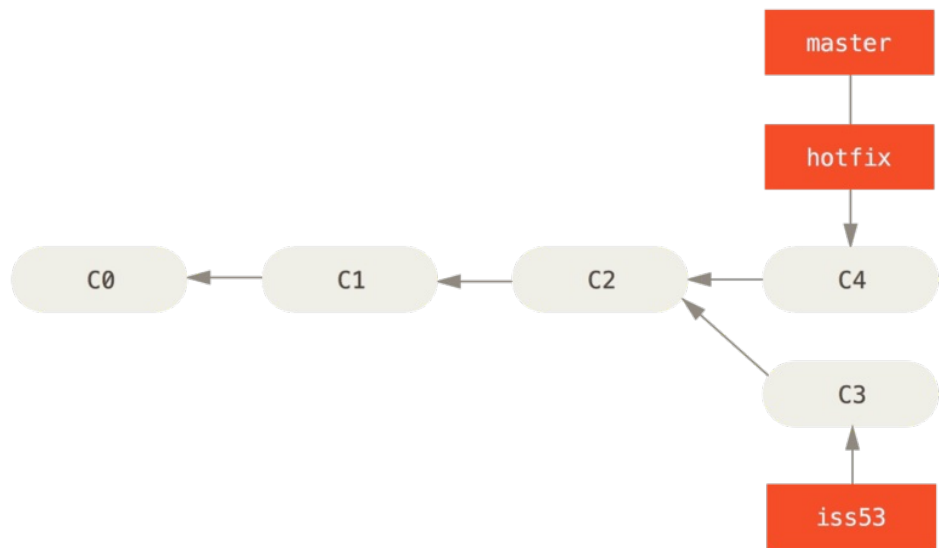


Figure 3-14. `master` 被快速进到 `hotfix` 关于这个紧急问题的解决方案发布之后，你准备回到被打断之前时的工作中。然而，你应该先删除 `hotfix` 分支，因为你已经不再需要它了——`master` 分支已经指向了同一个位置。你可以使用带 `-d` 选项的 `git branch` 命令来删除分支：

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

现在你可以切换回你正在工作的分支继续你的工作，也就是针对 #53 问题的那个分支（`iss53` 分支）。

```
$ 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(+)
```

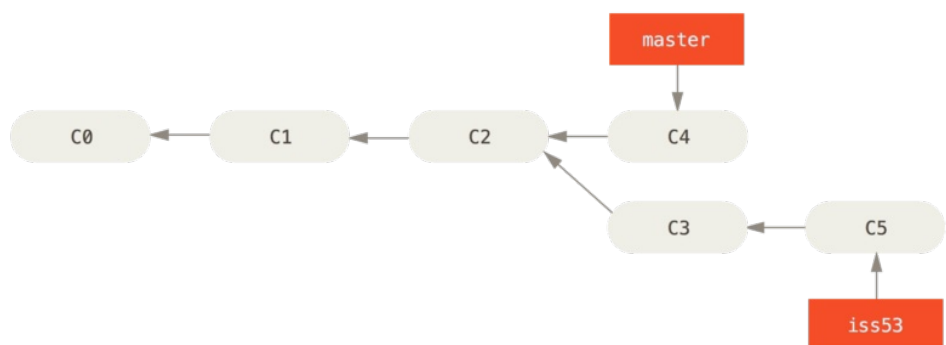


Figure 3-15. 继续在 `iss53` 分支上的工作你在 `hotfix` 分支上所做的工作并没有包含到 `iss53` 分支中。如果你需要拉取 `hotfix` 所做的修改，你可以使用 `git merge master` 命令将 `master` 分支合并入 `iss53` 分支，或者你也可以等到 `iss53` 分支完成其使命，再将其合并回 `master` 分支。

分支的合并

假设你已经修正了 #53 问题，并且打算将你的工作合并入 `master` 分支。为此，你需要合并 `iss53` 分支到 `master` 分支，这和之前你合并 `hotfix` 分支所做的工作差不多。你只需要检出到你想要合并入的分支，然后运行 `git merge` 命令：

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)
```

这和你之前合并 `hotfix` 分支的时候看起来有一点不一样。在这种情况下，你的开发历史从一个更早的地方开始分叉开来（`diverged`）。因为，`master` 分支所在提交并不是 `iss53` 分支所在提交的直接祖先，Git 不得不做一些额外的工作。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）以及这两个分支的工作祖先（C2），做一个简单的三方合并。

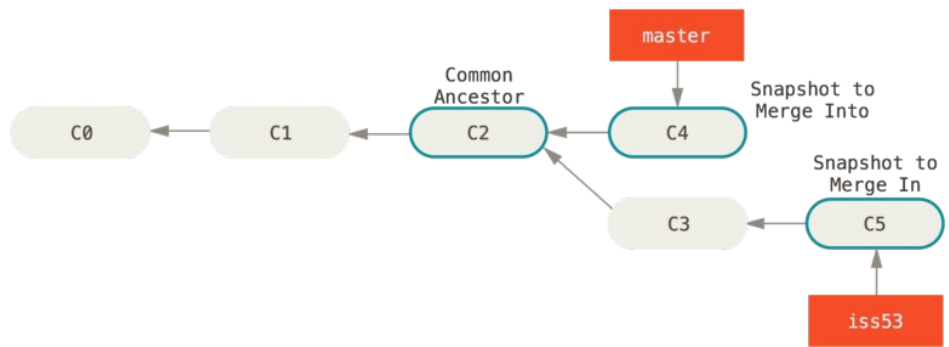


Figure 3-16. 一次典型合并中所用到的三个快照和之间将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。这个被称作一次合并提交，它的特别之处在于他有一个不止一个父提交。

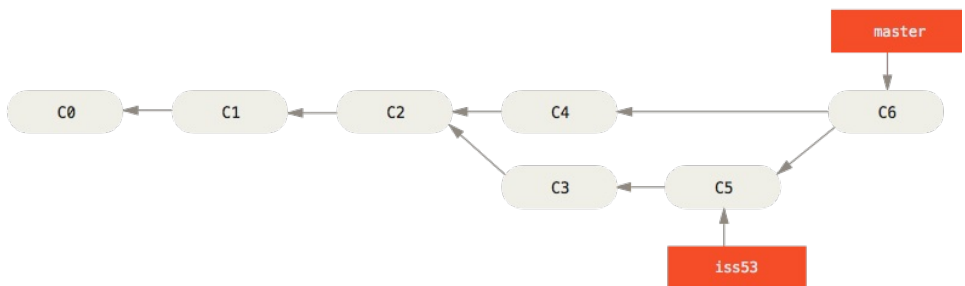


Figure 3-17. 一个合并提交需要指出的是，Git 会自行决定选取哪一个提交作为最优的共同祖先，并以此作为合并的基础；这和更加古老的 CVS 系统或者 Subversion（1.5 版本之前）不同，在这些古老的版本管理系统中，用户需要自己选择最佳的合并基础。Git 的这个优势使其在合并操作上比其他系统要简单很多。

既然你的修改已经合并进来了，你已不再需要 iss53 分支了。现在你可以在任务追踪系统中关闭此项任务，并删除这个分支。

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

遇到冲突时的分支合并

有时候合并操作不会如此顺利。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。如果你对 #53 问题的修改和有关 hotfix 的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

```
$ 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 Git 做了合并，但是没有自动地创建一个新的合并提交。Git 会暂停下来，等待你去解决合并产生的冲突。你可以在合并冲突后的任意时刻使用 `git status` 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

```
$ 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")
```

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

这表示 HEAD 所指示的版本（也就是你的 master 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）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。例如，你可以通过把这段内容换成下面的样子来解决冲突：

```
<div id="footer">
  please contact us at email.support@github.com</div>
```


上述的冲突解决方案仅保留了其中一个分支的修改，并且 <<<<<<<, =====, 和 >>>>>>> 这些行被完全删除了。在你解决了所有文件里的冲突之后，对每个文件使用 `git add` 命令来将其标记为冲突已解决。一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

如果你想使用图形化工具来解决冲突，你可以运行 `git mergetool`，该命令会为你启动一个合适的可视化合并工具，并带领你一步一步解决这些冲突：

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vim
diff emerge
Merging:
index.html

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

如果你想使用除默认工具（在这里 Git 使用 `opendiff` 做为默认的合并工具，因为作者在 Mac 上运行该程序）外的其他合并工具，你可以在“下列工具中（one of the following tools）”这句后面看到所有支持的合并工具。然后输入你喜欢的工具名字就可以了。

NOTE

如果你需要更加高级的工具来解决复杂的合并冲突，我们会在 [高级合并](#) 介绍更多关于分支合并的内容。

等你退出合并工具之后，Git 会询问刚才的合并是否成功。如果你回答是，Git 会暂存那些文件以表明冲突已解决：你可以再次运行 `git status` 来确认所有的合并冲突都被解决：

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
  modified:   index.html
```

如果你对结果感到满意，并且确定之前有冲突的文件都已经暂存了，这时你可以输入 `git commit` 来完成合并提交。默认情况下提交信息看起来像下面这个样子：

```
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.

# Please enter the commit message for your changes. Lines starting
# '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

如果你觉得上述的信息不够充分，不能完全体现分支合并的过程，你可以修改上述信息，添加一些细节给未来检视这个合并的读者一些帮助，告诉他们你是如何解决合并冲突的，以及理由是什么。

Git 分支 - 分支管理

| 分支管理 |

现在已经创建、合并、删除了一些分支，让我们看看一些常用的分支管理工具。

`git branch` 命令不只是可以创建与删除分支。如果不加任何参数运行它，会得到当前所有分支的一个列表：

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

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：

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

`--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。如果要查看哪些分支已经合并到当前分支，可以运行 `git branch --merged`：

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

因为之前已经合并了 iss53 分支，所以现在看到它在列表中。 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。

查看所有包含未合并工作的分支，可以运行 git branch --no-merged：

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

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：

```
$ 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'.
```

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。

1.4 Git 分支 - 分支开发工作流

分支开发工作流

现在你已经学会新建和合并分支，那么你可以或者应该用它来做什么呢？ 在本节，我们会介绍一些常见的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，才衍生出这些典型的工作模式，你可以根据项目实际情况选择一种用用看。

长期分支

因为 Git 使用简单的三方合并，所以就算在一段较长的时间内，反复把一个分支合并入另一个分支，也不是什么难事。 也就是说，在整个项目开发周期的不同阶段，你可以同时拥有多个开放的分支；你可以定期地把某些特性分支合并入其他分支中。

许多使用 Git 的开发者都喜欢使用这种方式来工作，比如只在 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。

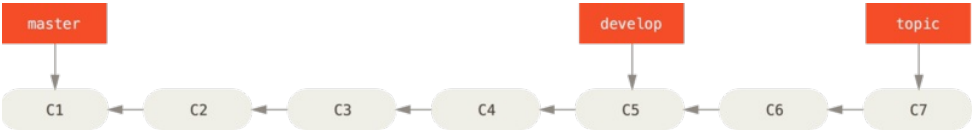


Figure 3-18. 渐进稳定分支的线性图通常把他们想象成流水线（work silos）可能更好理解一点，那些经过测试考验的提交会被遴选到更加稳定的流水线上上去。

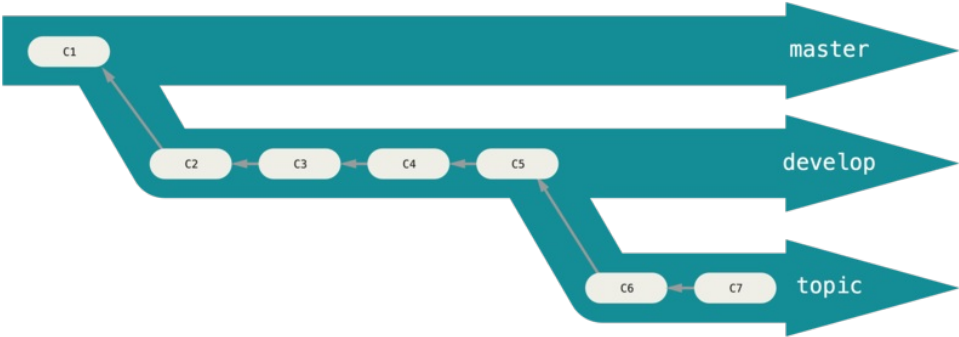


Figure 3-19. 渐进稳定分支的流水线（“silo”）视图你可以用这种方法维护不同层次的安全性。 一些大型项目还有一个 proposed（建议）或 pu: proposed updates（建议更新）分支，它可能因包含一些不成熟的内容而不能进入 next 或者 master分支。 这么做的目的是使你的分支具有不同级别的安全性；当它们具有一定程度的安全性后，再把它们合并入具有更高级别安全性的分支中。 再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

特性分支

特性分支对任何规模的项目都适用。 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。 也许你从来没有在其他的版本控制系统（VCS）上这么做过，因为在那些版本控制系统中创建和合并分支通常很费劲。 然而，在 Git 中一天之内多次创建、使用、合并、删除分支都很常见。

你已经在上一节中你创建的 iss53 和 hotfix 特性分支中看到过这种用法。 你在上一节用到的特性分支（iss53 和 hotfix 分支）中提交了一些更新，并且在它们合并入主干分支之后，你又删除了它们。 这项技术能让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，而不在乎它们建立的顺序或工作进度。

考虑这样一个例子，你在 master 分支上工作到 C1，这时为了解决一个问题而新建 iss91 分支，在iss91 分支上工作到 C4，然而对于那个问题你有了新的想法，于是你再新建一个 iss91v2 分支试图用另一种方法解决那个问题，接着你回到 master 分支工作了一会儿，你又冒出了一个不太确定的想法，你便在 C10 的时候新建一个 dumbidea 分支，并在上面做些实验。 你的提交历史看起来像下面这个样子：

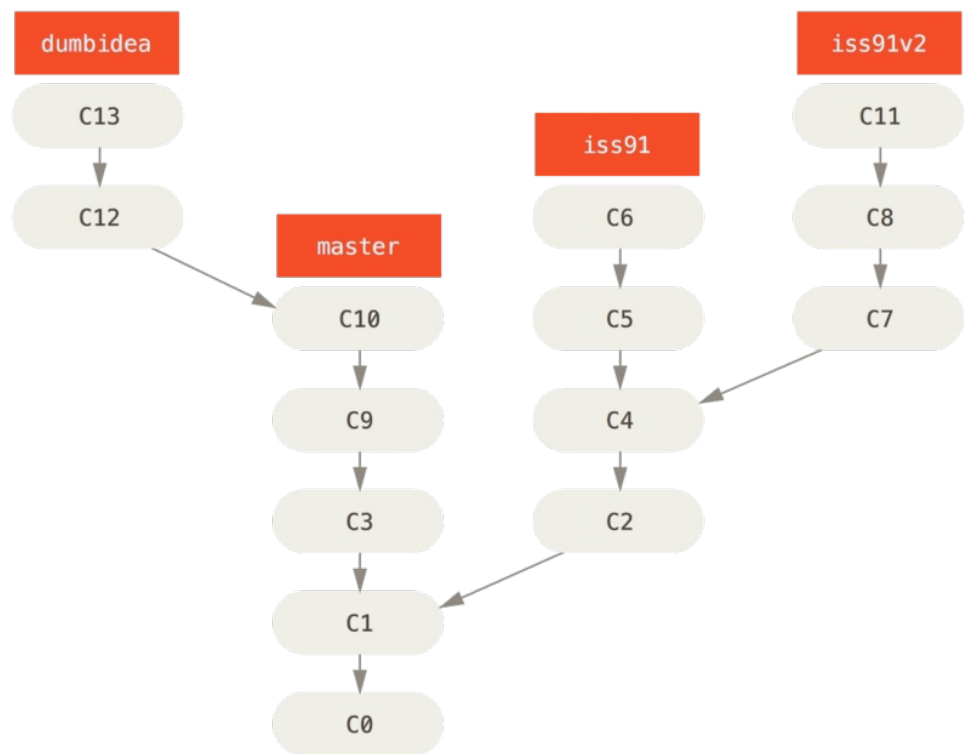


Figure 3-20. 拥有多个特性分支的提交历史现在，我们假设两件事情：你决定使用第二个方案来解决那个问题，即使用在 `iss91v2` 分支中方案；另外，你将 `dumbidea` 分支拿给你的同事看过之后，结果发现这是个惊人之举。这时你可以抛弃 `iss91` 分支（即丢弃 `C5` 和 `C6` 提交），然后把另外两个分支合并入主干分支。最终你的提交历史看起来像下面这个样子：

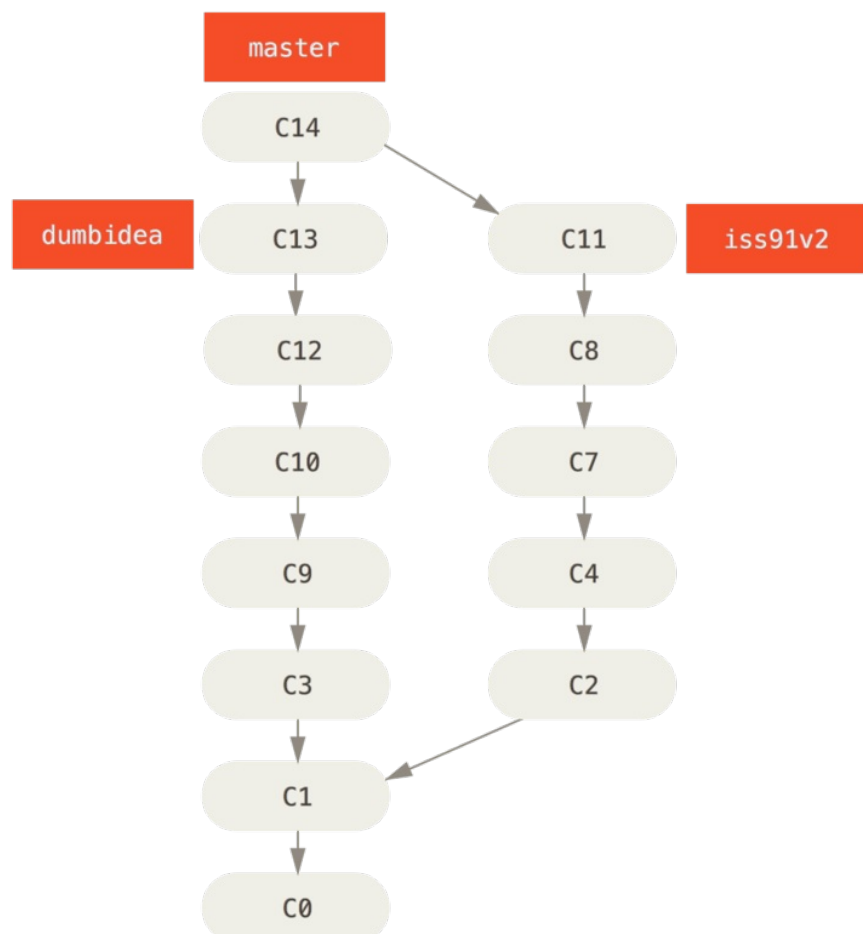


Figure 3-21. 合并了 `dumbidea` 和 `iss91v2` 分支之后的提交历史我们将会 在 [分布式 Git](#) 中向你揭示更多有关分支工作流的细节，因此，请确保你阅读完那个章节之后，再来决定你的下个项目要使用什么样的分支策略（branching scheme）。

请牢记，当你做这么多操作的时候，这些分支全部都存于本地。当你新建和合并分支的时候，所有这一切都只发生在你本地的 Git 版本库中——没有与服务器发生交互。

本文地址: <http://www.linuxprobe.com/git-branch-history-today.html>

免费提供最新Linux技术教程书籍，为开源技术爱好者努力做得更多更好: <http://www.linuxprobe.com/>

- ▲ 上一篇

分享记录我的Linux系统入门学习经验
- ▼ 下一篇

命令行 TODO 工具中的王者

我的同类文章

Linux (109)					
• 命令行 TODO 工具中的王者	2016-10-29	阅读 3	• 分享记录我的Linux系统入门学习经验	2016-10-28	阅读 19
• Linux新内核：提升系统性能	2016-10-28	阅读 15	• Git图形界面的使用	2016-10-27	阅读 44
• VIM的使用方法	2016-10-26	阅读 18	• 快速掌握grep命令及正则表达式	2016-10-26	阅读 11
• 快来使用HTTPS吧	2016-10-25	阅读 16	• 怎样Linux下修复U盘驱动器	2016-10-25	阅读 15
• Linux 五大初始化系统	2016-10-24	阅读 17	• 怎样Linux下修复U盘驱动器	2016-10-24	阅读 11
更多文章					

猜你在找

- HTML 5移动开发从入门到精通

▪ 零基础学HTML 5实战开发(第一季)

▪ Java之路

▪ Swift视频教程(第四季)

▪ C++语言基础
- Base64算法的前世今生一

▪ 无监督学习图像处理应用中的前…

▪ Android病毒分析报告- 手机支付…

▪ async & await 的前世今生

▪ UE4的前世今生

查看评论

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题

Hadoop

AWS

移动游戏

Java

Android

iOS

Swift

智能硬件

Docker

OpenStack

VPN

Spark

ERP

IE10

Eclipse

CRM

JavaScript

数据库

Ubuntu

NFC

WAP

jQuery

BI

HTML5

Spring

Apache

.NET

API

HTML

SDK

IIS

Fedora

XML

LBS

Unity

Splashtop

UML

components

Windows Mobile

Rails

QEMU

KDE

Cassandra

CloudStack

FTC

coremail

OPhone

CouchBa

Web App

SpringSide

Maemo

Compuware

大数据

aptech

Perl

Tornado

Ru

Pure

Solr

Angular

Cloud Foundry

Redis

Scala






Django

Bootstrap

4	哈佛大學（英國）
5	麻省理工學院（英國）
6	普林斯頓大學（英國）
7	劍橋大學（英國）
8	倫敦帝國學院（英國）
9	加州大學柏克萊分校（美國）
10	芝加哥大學（美國）
21	多倫多大學（加拿大）
30	卑詩大學（加拿大）

世界大学排名

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

 网站客服  杂志客服  微博客服  webmaster@csdn.net  400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 | 江苏
京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved 