返回 码云 首页(目录) 章节列表 ▾ Pro Git（中文版）

5 分布式 Git

为了便于项目中的所有开发者分享代码，我们准备好了一台服务器存放远程 Git 仓库。经过前面几章的学习，我们已经学会了一些基本的本地工作流程中所需用到的命令。接下来，我们要学习下如何利用 Git 来组织和完成分布式工作流程。

特别是，当作为项目贡献者时，我们该怎么做才能方便维护者采纳更新；或者作为项目维护者时，又该怎样有效管理大量贡献者的提交。

5.1 分布式工作流程

同传统的集中式版本控制系统（CVCS）不同，开发者之间的协作方式因着 Git 的分布式特性而变得更为灵活多样。在集中式系统上，每个开发者就像是连接在集线器上的节点，彼此的工作方式大体相像。而在 Git 网络中，每个开发者同时扮演着节点和集线器的角色，这就是说，每一个开发者都可以将自己的代码贡献到另外一个开发者的仓库中，或者建立自己的公共仓库，让其他开发者基于自己的工作开始，为自己的仓库贡献代码。于是，Git 的分布式协作便可以衍生出种种不同的工作流程，我会在接下来的章节介绍几种常见的应用方式，并分别讨论各自的优缺点。你可以选择其中的一种，或者结合起来，应用到你自己的项目中。

集中式工作流

通常，集中式工作流程使用的都是单点协作模型。一个存放代码仓库的中心服务器，可以接受所有开发者提交的代码。所有的开发者都是普通的节点，作为中心集线器的消费者，平时的工作就是和中心仓库同步数据（见图 5-1）。

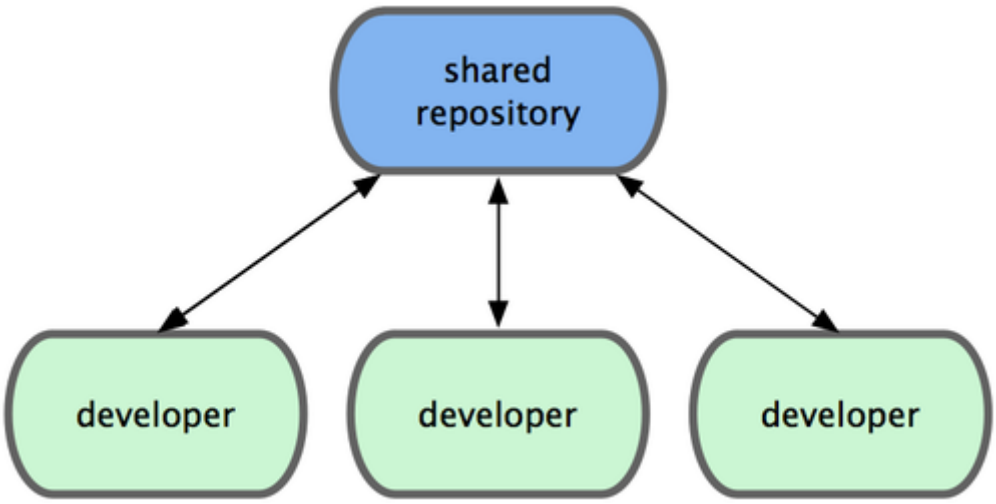

图 5-1. 集中式工作流

如果两个开发者从中心仓库克隆代码下来，同时作了一些修订，那么只有第一个开发者可以顺利地把数据推送到共享服务器。第二个开发者在提交他的修订之前，必须先下载合并服务器上的数据，解决冲突之后才能推送数据到共享服务器上。在 Git 中这么用也决无问题，这就好比是在用 Subversion（或其他 CVCS）一样，可以很好地工作。

如果你的团队不是很大，或者大家都已经习惯了使用集中式工作流程，完全可以采用这种简单的模式。只需要配置好一台中心服务器，并给每个人推送数据的权限，就可以开展工作了。但如果提交代码时有冲突， Git 根本就不会让用户覆盖他人代码，它直接驳回第二个人的提交操作。这就等于

告诉提交者，你所作的修订无法通过快近（fast-forward）来合并，你必须先拉取最新数据下来，手工解决冲突合并后，才能继续推送新的提交。绝大多数人都熟悉和了解这种模式的工作方式，所以使用也非常广泛。

集成管理员工作流

由于 Git 允许使用多个远程仓库，开发者便可以建立自己的公共仓库，往里面写数据并共享给他人，而同时又可以从别人的仓库中提取他们的更新过来。这种情形通常都会有个代表着官方发布的项目仓库（blessed repository），开发者们由此仓库克隆出一个自己的公共仓库（developer public），然后将自己的提交推送上去，请求官方仓库的维护者拉取更新合并到主项目。维护者在自己的本地也有个克隆仓库（integration manager），他可以将你的公共仓库作为远程仓库添加进来，经过测试无误后合并到主干分支，然后再推送到官方仓库。工作流程看起来就像图 5-2 所示：

1. 项目维护者可以推送数据到公共仓库 blessed repository。
2. 贡献者克隆此仓库，修订或编写新代码。
3. 贡献者推送数据到自己的公共仓库 developer public。
4. 贡献者给维护者发送邮件，请求拉取自己的最新修订。
5. 维护者在自己本地的 integration manger 仓库中，将贡献者的仓库加为远程仓库，合并更新并做测试。
6. 维护者将合并后的更新推送到主仓库 blessed repository。

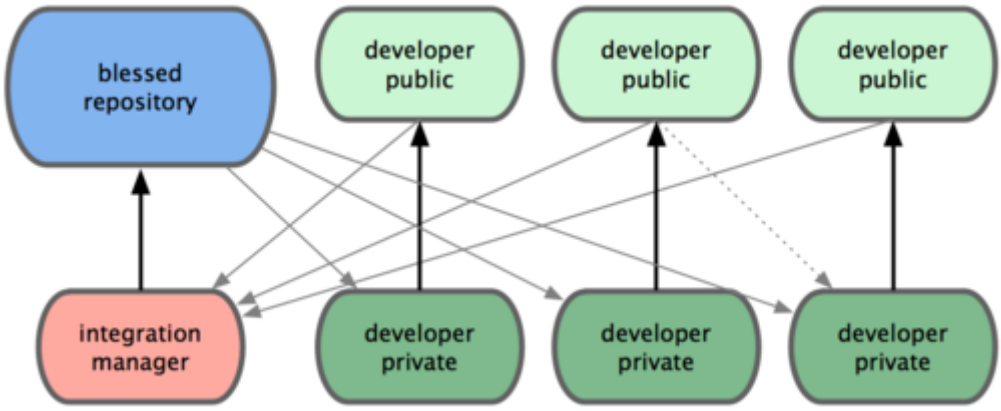

图 5-2. 集成管理员工作流

在 GitHub 网站上使用得最多的就是这种工作流。人们可以复制（fork 亦即克隆）某个项目到自己的列表中，成为自己的公共仓库。随后将自己的更新提交到这个仓库，所有人都可以看到你的每次更新。这么做最主要的优点在于，你可以按照自己的节奏继续工作，而不必等待维护者处理你提交的更新；而维护者也可以按照自己的节奏，任何时候都可以过来处理接纳你的贡献。

司令官与副官工作流

这其实是上一种工作流的变体。一般超大型的项目才会用到这样的工作方式，像是拥有数百协作开发者的 Linux 内核项目就是如此。各个集成管理员分别负责集成项目中的特定部分，所以称为副官（lieutenant）。而所有这些集成管理员头上还有一位负责统筹的总集成管理员，称为司令官（dictator）。司令官维护的仓库用于提供所有协作者拉取最新集成的项目代码。整个流程看起来如图 5-3 所示：

1. 一般的开发者在自己的特性分支上工作，并不定期地根据主干分支（dictator 上的 master）衍合。
2. 副官（lieutenant）将普通开发者的特性分支合并到自己的 master 分支中。

3. 司令官（dictator）将所有副官的 master 分支并入自己的 master 分支。
4. 司令官（dictator）将集成后的 master 分支推送到共享仓库 blessed repository 中，以便所有其他开发者以此为基础进行衍合。

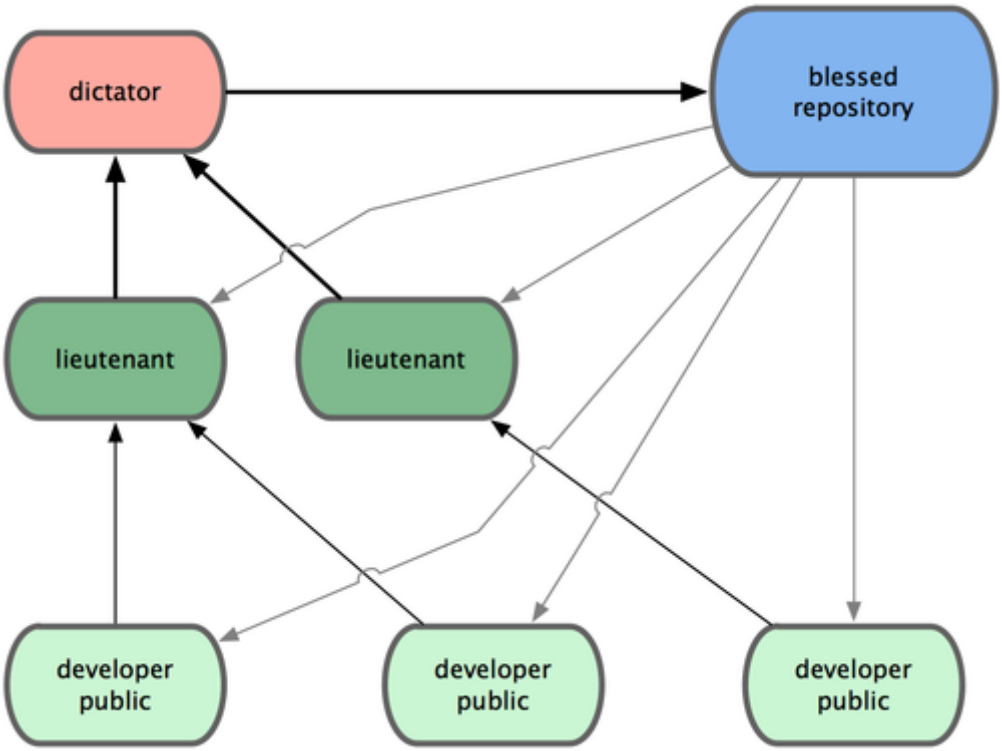

图 5-3. 司令官与副官工作流

这种工作流程并不常用，只有当项目极为庞杂，或者需要多级别管理时，才会体现出优势。利用这种方式，项目总负责人（即司令官）可以把大量分散的集成工作委托给不同的小组负责人分别处理，最后再统筹起来，如此各人的职责清晰明确，也不易出错（译注：此乃分而治之）。

以上介绍的是常见的分布式系统可以应用的工作流程，当然不止于 Git。在实际的开发工作中，你可能会遇到各种为了满足特定需求而有所变化的工作方式。我想现在你应该已经清楚，接下来自己需要用哪种方式开展工作了。下节我还会再举些例子，看看各式工作流中的每个角色具体应该如何操作。

5.2 为项目作贡献

接下来，我们来学习一下作为项目贡献者，会有哪些常见的工作模式。

不过要说清楚整个协作过程真的很难，Git 如此灵活，人们的协作方式便可以各式各样，没有固定不变的范式可循，而每个项目的具体情况又多少会有些不同，比如说参与者的规模，所选择的工作流程，每个人的提交权限，以及 Git 以外贡献等等，都会影响到具体操作的细节。

首当其冲的是参与者规模。项目中有多少开发者是经常提交代码的？经常又是多久呢？大多数两至三人的小团队，一天大约只有几次提交，如果不是什么热门项目的话就更少了。可要是在大公司里，或者大项目中，参与者可以多到上千，每天都会有十几个上百个补丁提交上来。这种差异带来的影响是显著的，越是多的人参与进来，就越难保证每次合并正确无误。你正在工作的代码，可能会因为合并进来其他人的更新而变得过时，甚至受创无法运行。而已经提交上去的更新，也可能在等着审核合并的过程中变得过时。那么，我们该怎样做才能确保代码是最新的，提交的补丁也是可用的呢？

接下来便是项目所采用的工作流。是集中式的，每个开发者都具有等同的写权限？项目是否有专人负责检查所有补丁？是不是所有补丁都做过同行复阅（peer-review）再通过审核的？你是否参与审核过程？如果使用副官系统，那你是不是限定于只能向此副官提交？

还有你的提交权限。有或没有向主项目提交更新的权限，结果完全不同，直接决定最终采用怎样的工作流。如果不能直接提交更新，那该如何贡献自己的代码呢？是不是该有个什么策略？你每次贡献代码会有多少量？提交频率呢？

所有以上这些问题都会或多或少影响到最终采用的工作流。接下来，我会在一系列由简入繁的具体用例中，逐一阐述。此后在实践时，应该可以借鉴这里的例子，略作调整，以满足实际需要构建自己的工作流。

提交指南

开始分析特定用例之前，先来了解下如何撰写提交说明。一份好的提交指南可以帮助协作者更轻松更有效地配合。Git 项目本身就提供了一份文档（Git 项目源代码目录中 `Documentation/SubmittingPatches`），列数了大量提示，从如何编撰提交说明到提交补丁，不一而足。

首先，请不要在更新中提交多余的白字符（whitespace）。Git 有种检查此类问题的方法，在提交之前，先运行 `git diff --check`，会把可能的多余白字符修正列出来。下面的示例，我已经把终端中显示为红色的白字符用 `X` 替换掉：

```
$ git diff --check
    lib/simplegit.rb:5: trailing whitespace.
    + @git_dir = File.expand_path(git_dir)XX
    lib/simplegit.rb:7: trailing whitespace.
    + XXXXXXXXXXX
    lib/simplegit.rb:26: trailing whitespace.
    + def command(git_cmd)XXXX
```

这样在提交之前你就可以看到这类问题，及时解决以免困扰其他开发者。

接下来，请将每次提交限定于完成一次逻辑功能。并且可能的话，适当地分解为多次小更新，以便每次小型提交都更易于理解。请不要在周末穷追猛打一次性解决五个问题，而最后拖到周一再提交。就算是这样也请尽可能利用暂存区域，将之前的改动分解为每次修复一个问题，再分别提交和加注说明。如果针对两个问题改动的是同一个文件，可以试试看 `git add --patch` 的方式将部分内容置入暂存区域（我们会在第六章再详细介绍）。无论是五次小提交还是混杂在一起的大提交，最终分支末端的项目快照应该还是一样的，但分解开来之后，更便于其他开发者复阅。这么做也方便自己将来取消某个特定问题的修复。我们将在第六章介绍一些重写提交历史，同暂存区域交互的技巧和工具，以便最终得到一个干净有意义，且易于理解的提交历史。

最后需要谨记的是提交说明的撰写。写得好可以让大家协作起来更轻松。一般来说，提交说明最好限制在一行以内，50 个字符以下，简明扼要地描述更新内容，空开一行后，再展开详细注解。Git 项目本身需要开发者撰写详尽注解，包括本次修订的因由，以及前后不同实现之间的比较，我们也该借鉴这种做法。另外，提交说明应该用祈使现在式语态，比如，不要说成 “I added tests for” 或 “Adding tests for” 而应该用 “Add tests for”。下面是来自 tpope.net 的 Tim Pope 原创的提交说明格式模版，供参考：

```
本次更新的简要描述（50 个字符以内）

    如果必要，此处展开详尽阐述。段落宽度限定在 72 个字符以内。
    某些情况下，第一行的简要描述将用作邮件标题，其余部分作为邮件正文。
    其间的空行是必要的，以区分两者（当然没有正文另当别论）。
    如果并在一起，rebase 这样的工具就可能会迷惑。
```

```
另起空行后，再进一步补充其他说明。

- 可以使用这样的条目列举式。

- 一般以单个空格紧跟短划线或者星号作为每项条目的起始符。每个条目间用一空行隔开。
不过这里按自己项目的约定，可以略作变化。
```

如果你的提交说明都用这样的格式来书写，好多事情就可以变得十分简单。Git 项目本身就是这样要求的，我强烈建议你到 Git 项目仓库下运行 `git log --no-merges` 看看，所有提交历史的说明是怎样撰写的。（译注：如果现在还没有克隆 git 项目源代码，是时候 `git clone git://git.kernel.org/pub/scm/git/git.git` 了。）

为简单起见，在接下来的例子（及本书随后的所有演示）中，我都不会用这种格式，而使用 `-m` 选项提交 `git commit`。不过请还是按照我之前讲的做，别学我这里偷懒的方式。

私有的小型团队

我们从最简单的情况开始，一个私有项目，与你一起协作的还有另外一到两位开发者。这里说私有，是指源代码不公开，其他人无法访问项目仓库。而你和其他开发者则都具有推送数据到仓库的权限。

这种情况下，你们可以用 Subversion 或其他集中式版本控制系统类似的工作流来协作。你仍然可以得到 Git 带来的其他好处：离线提交，快速分支与合并等等，但工作流程还是差不多的。主要区别在于，合并操作发生在客户端而非服务器上。让我们来看看，两个开发者一起使用同一个共享仓库，会发生些什么。第一个人，John，克隆了仓库，作了些更新，在本地提交。（下面的例子中省略了常规提示，用 `...` 代替以节约版面。）

```
# John's Machine
    $ git clone john@githost:simplegit.git
    Initialized empty Git repository in /home/john/simplegit/.git/
    ...
    $ cd simplegit/
    $ vim lib/simplegit.rb
    $ git commit -am 'removed invalid default value'
    [master 738ee87] removed invalid default value
    1 files changed, 1 insertions(+), 1 deletions(-)
```

第二个开发者，Jessica，一样这么做：克隆仓库，提交更新：

```
# Jessica's Machine
    $ git clone jessica@githost:simplegit.git
    Initialized empty Git repository in /home/jessica/simplegit/.git/
    ...
    $ cd simplegit/
    $ vim TODO
    $ git commit -am 'add reset task'
    [master fbff5bc] add reset task
    1 files changed, 1 insertions(+), 0 deletions(-)
```

现在，Jessica 将她的工作推送到服务器上：

```
# Jessica's Machine
    $ git push origin master
    ...
    To jessica@githost:simplegit.git
    1edee6b..fbff5bc master -> master
```

John 也尝试推送自己的工作上去：

```
# John's Machine
    $ git push origin master
    To john@githost:simplegit.git
    ! [rejected] master -> master (non-fast forward)
    error: failed to push some refs to 'john@githost:simplegit.git'
```

John 的推送操作被驳回，因为 Jessica 已经推送了新的数据上去。请注意，特别是你用惯了 Subversion 的话，这里其实修改的是两个文件，而不是同一个文件的同一个地方。Subversion 会在服务器端自动合并提交上来的更新，而 Git 则必须先在本地合并后才能推送。于是，John 不得不先把 Jessica 的更新拉下来：

```
$ git fetch origin
    ...
    From john@githost:simplegit
    + 049d078...fbff5bc master -> origin/master
```

此刻，John 的本地仓库如图 5-4 所示：

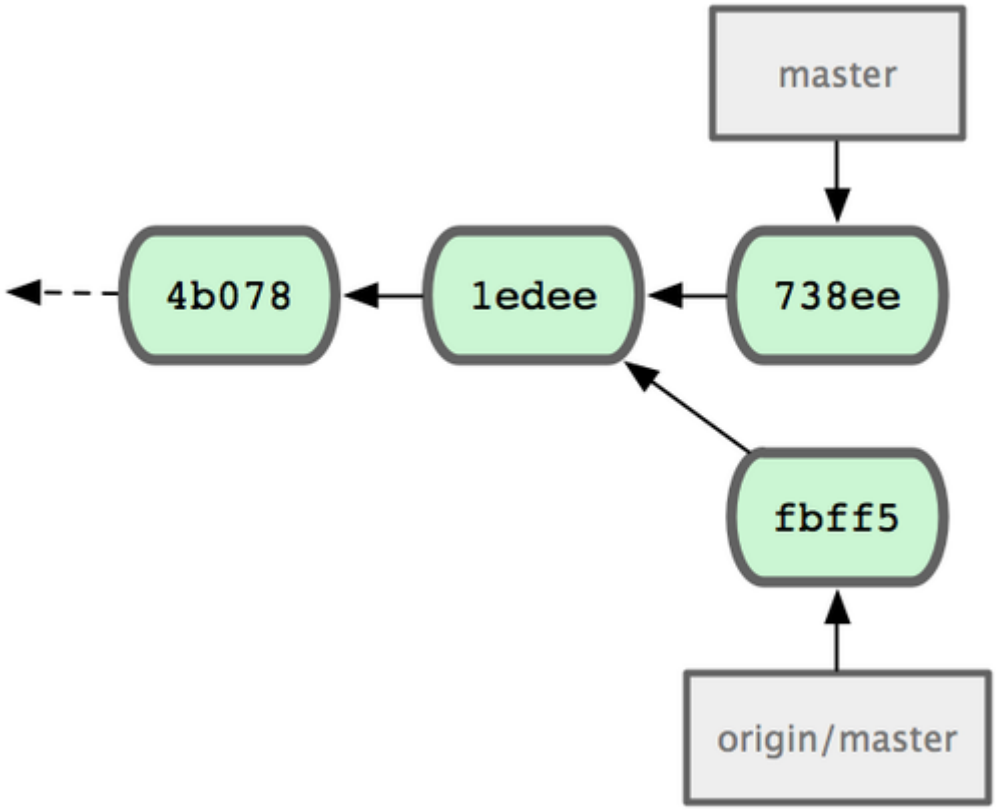

图 5-4. John 的仓库历史

虽然 John 下载了 Jessica 推送到服务器的最近更新（fbff5），但目前只是 `origin/master` 指针指向它，而当前的本地分支 `master` 仍然指向自己的更新（738ee），所以需要先把她的提交合并过来，才能继续推送数据：

```
$ git merge origin/master
    Merge made by recursive.
    TODO | 1 +
    1 files changed, 1 insertions(+), 0 deletions(-)
```

还好，合并过程非常顺利，没有冲突，现在 John 的提交历史如图 5-5 所示：

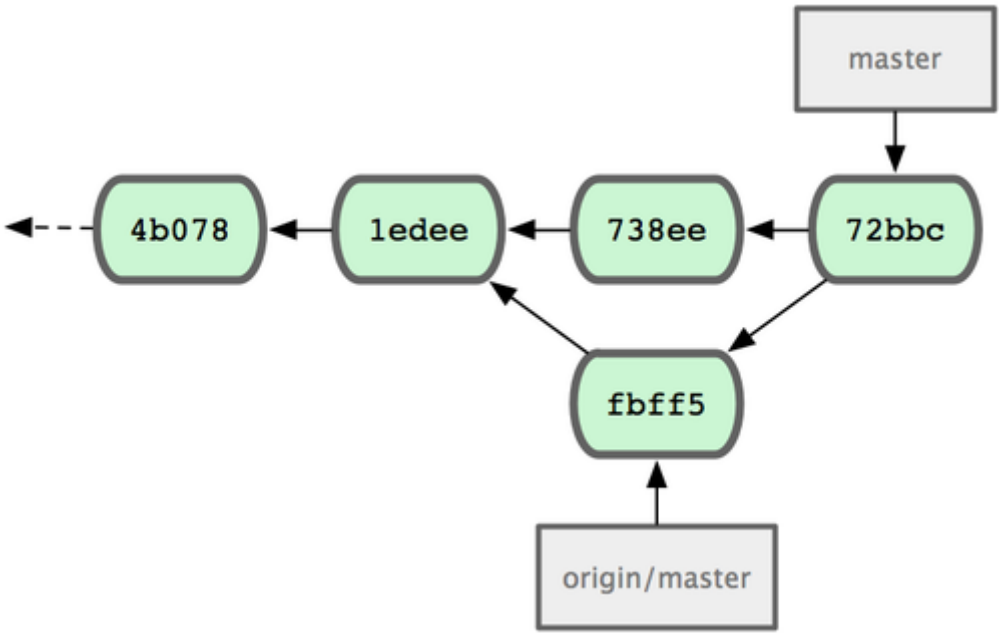

图 5-5. 合并 origin/master 后 John 的仓库历史

现在，John 应该再测试一下代码是否仍然正常工作，然后将合并结果（72bbc）推送到服务器上：

```
$ git push origin master
    ...
    To john@githost:simplegit.git
    fbff5bc..72bbc59 master -> master
```

最终，John 的提交历史变为图 5-6 所示：

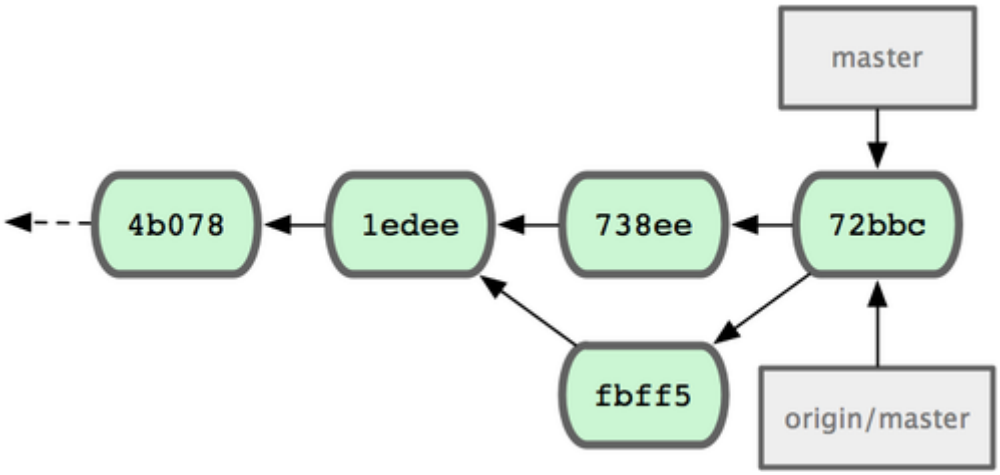

图 5-6. 推送后 John 的仓库历史

而在这段时间，Jessica 已经开始在另一个特性分支工作了。她创建了 `issue54` 并提交了三次更新。她还没有下载 John 提交的合并结果，所以提交历史如图 5-7 所示：

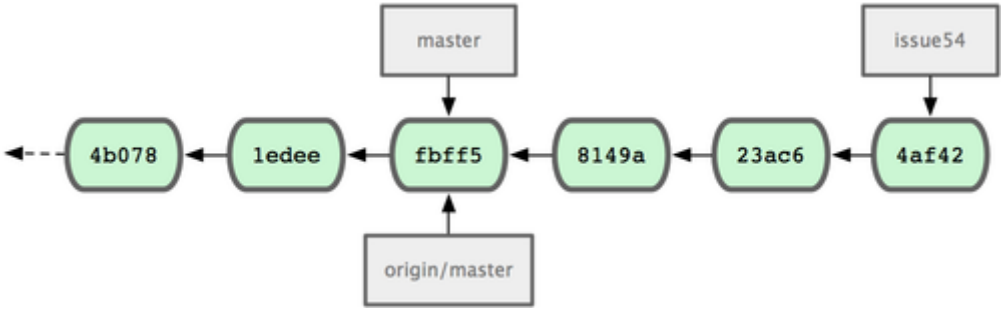

图 5-7. Jessica 的提交历史

Jessica 想要先和服务器上的数据同步，所以先下载数据：

```
# Jessica's Machine
    $ git fetch origin
    ...
    From jessica@githost:simplegit
    fbff5bc..72bbc59 master -> origin/master
```

于是 Jessica 的本地仓库历史多出了 John 的两次提交（738ee 和 72bbc），如图 5-8 所示：

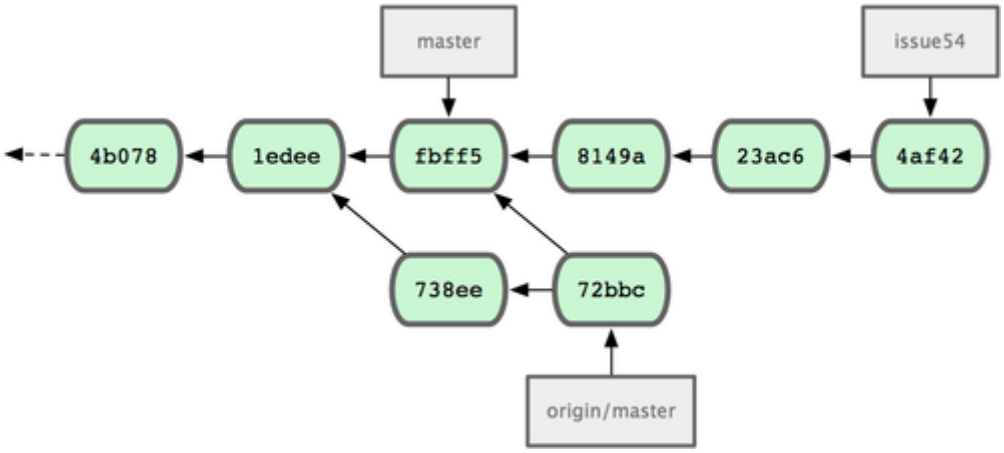

图 5-8. 获取 John 的更新之后 Jessica 的提交历史

此时，Jessica 在特性分支上的工作已经完成，但她想在推送数据之前，先确认下要并进来的数据究竟是什么，于是运行 `git log` 查看：

```
$ git log --no-merges origin/master ^issue54
    commit 738ee872852dfaa9d6634e0dea7a324040193016
    Author: John Smith <jsmith@example.com>
    Date: Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

现在，Jessica 可以将特性分支上的工作并到 `master` 分支，然后再并入 John 的工作（`origin/master`）到自己的 `master` 分支，最后再推送回服务器。当然，得先切回主分支才能集成所有数据：

```
$ git checkout master
    Switched to branch "master"
    Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

要合并 `origin/master` 或 `issue54` 分支，谁先谁后都没有关系，因为它们都在上游（upstream）（译注：想像分叉的更新像是汇流成河的源头，所以上游 upstream 是指最新的提交），所以无所谓先后顺序，最终合并后的内容快照都是一样的，而仅是提交历史看起来会有些先后差别。Jessica 选择先合并 `issue54`：

```
$ git merge issue54
    Updating fbff5bc..4af4298
    Fast forward
    README | 1 +
    lib/simplegit.rb | 6 +++++-
    2 files changed, 6 insertions(+), 1 deletions(-)
```

正如所见，没有冲突发生，仅是一次简单快进。现在 Jessica 开始合并 John 的工作（`origin/master`）：

```
$ git merge origin/master
    Auto-merging lib/simplegit.rb
    Merge made by recursive.
    lib/simplegit.rb | 2 +-
    1 files changed, 1 insertions(+), 1 deletions(-)
```

所有的合并都非常干净。现在 Jessica 的提交历史如图 5-9 所示：

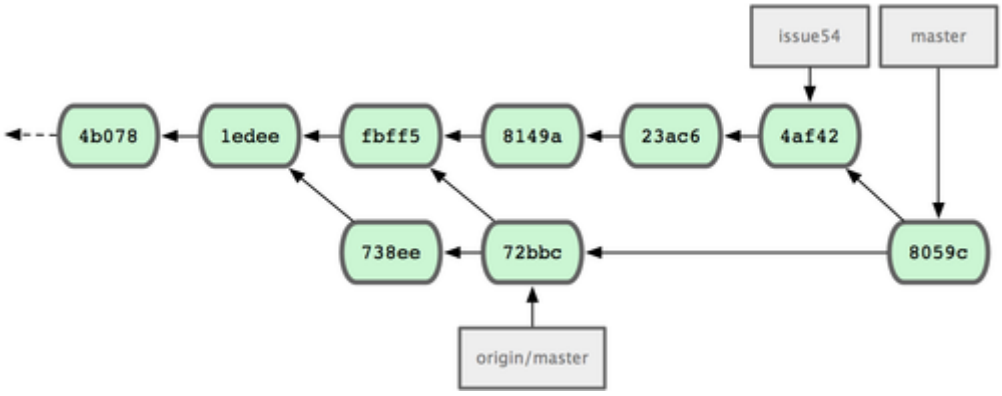

图 5-9. 合并 John 的更新后 Jessica 的提交历史

现在 Jessica 已经可以在自己的 `master` 分支中访问 `origin/master` 的最新改动了，所以她应该可以成功推送最后的合并结果到服务器上（假设 John 此时没再推送新数据上来）：

```
$ git push origin master
    ...
    To jessica@githost:simplegit.git
    72bbc59..8059c15 master -> master
```

至此，每个开发者都提交了若干次，且成功合并了对方的工作成果，最新的提交历史如图 5-10 所示：

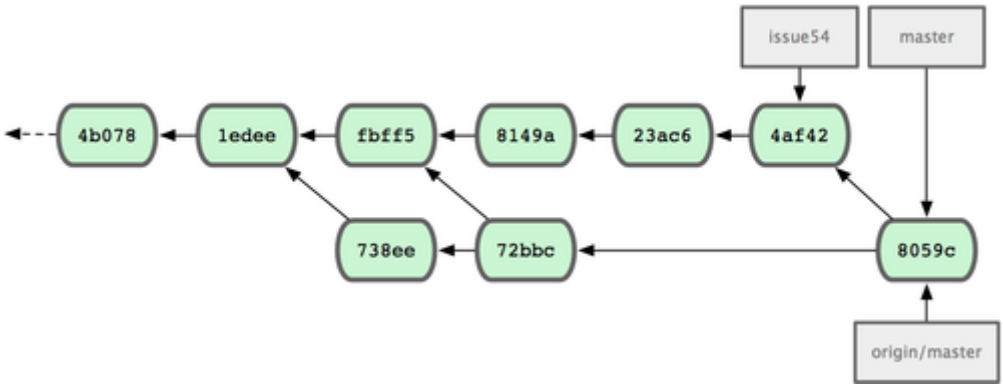

图 5-10. Jessica 推送数据后的提交历史

以上就是最简单的协作方式之一：先在自己的特性分支中工作一段时间，完成后合并到自己的 `master` 分支；然后下载合并 `origin/master` 上的更新（如果有的话），再推回远程服务器。一般的协作流程如图 5-11 所示：

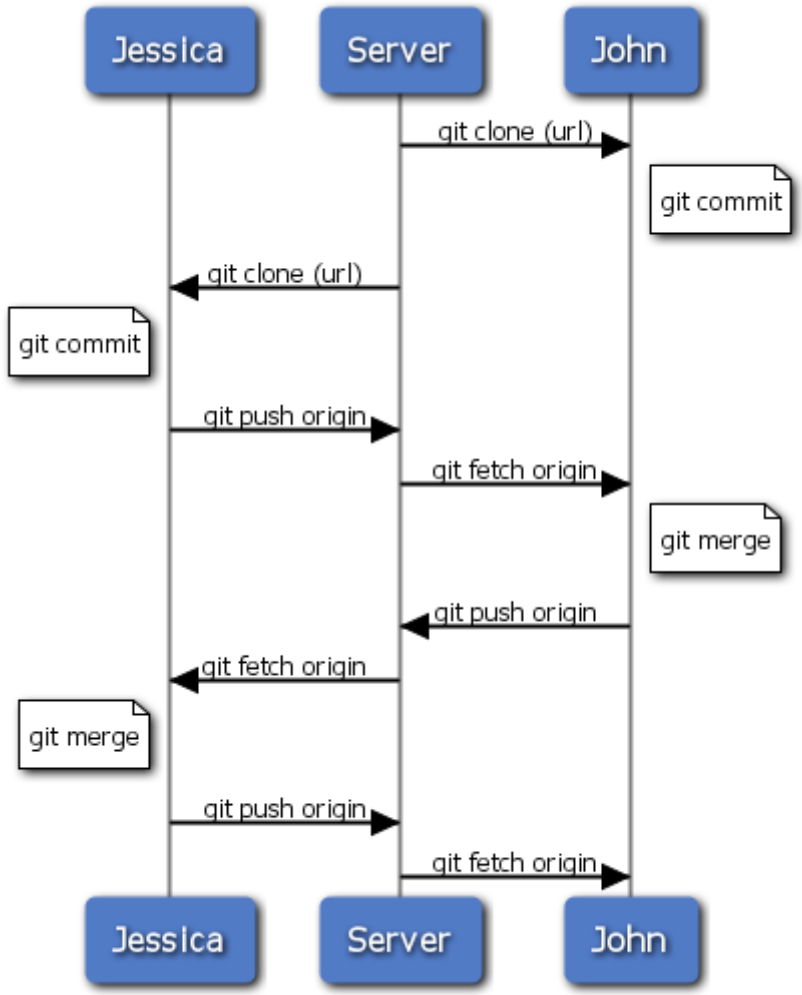

图 5-11. 多用户共享仓库协作方式的一般工作流程时序

私有团队间协作

现在我们来看更大一点规模的私有团队协作。如果有几个小组分头负责若干特性的开发和集成，那他们之间的协作过程是怎样的。

假设 John 和 Jessica 一起负责开发某项特性 A，而同时 Jessica 和 Josie 一起负责开发另一项功能 B。公司使用典型的集成管理员式工作流，每个组都有一名管理员负责集成本组代码，及更新项目主

仓库的 `master` 分支。所有开发都在代表小组的分支上进行。

让我们跟随 Jessica 的视角看看她的工作流程。她参与开发两项特性，同时和不同小组的开发者一起协作。克隆生成本地仓库后，她打算先着手开发特性 A。于是创建了新的 `featureA` 分支，继而编写代码：

```
# Jessica's Machine
    $ git checkout -b featureA
    Switched to a new branch "featureA"
    $ vim lib/simplegit.rb
    $ git commit -am 'add limit to log function'
    [featureA 3300904] add limit to log function
    1 files changed, 1 insertions(+), 1 deletions(-)
```

此刻，她需要分享目前的进展给 John，于是她将自己的 `featureA` 分支提交到服务器。由于 Jessica 没有权限推送数据到主仓库的 `master` 分支（只有集成管理员有此权限），所以只能将此分支推上去同 John 共享协作：

```
$ git push origin featureA
    ...
    To jessica@githost:simplegit.git
    * [new branch] featureA -> featureA
```

Jessica 发邮件给 John 让他上来看看 `featureA` 分支上的进展。在等待他的反馈之前，Jessica 决定继续工作，和 Josie 一起开发 `featureB` 上的特性 B。当然，先创建此分支，分叉点以服务器上的 `master` 为起点：

```
# Jessica's Machine
    $ git fetch origin
    $ git checkout -b featureB origin/master
    Switched to a new branch "featureB"
```

随后，Jessica 在 `featureB` 上提交了若干更新：

```
$ vim lib/simplegit.rb
    $ git commit -am 'made the ls-tree function recursive'
    [featureB e5b0fdc] made the ls-tree function recursive
    1 files changed, 1 insertions(+), 1 deletions(-)
    $ vim lib/simplegit.rb
    $ git commit -am 'add ls-files'
    [featureB 8512791] add ls-files
    1 files changed, 5 insertions(+), 0 deletions(-)
```

现在 Jessica 的更新历史如图 5-12 所示：

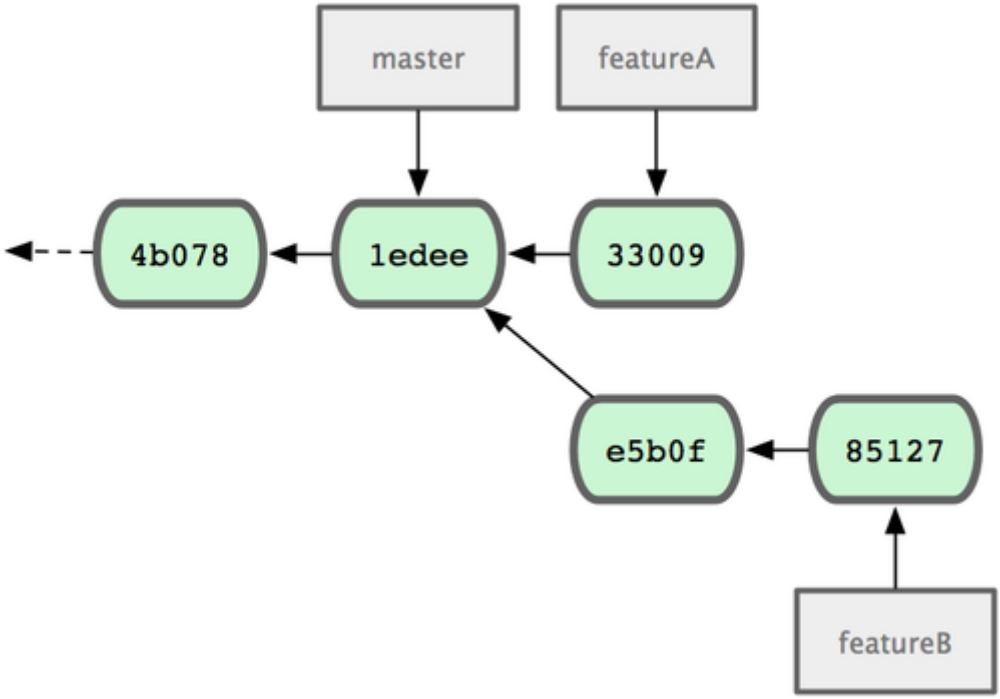

图 5-12. Jessica 的更新历史

Jessica 正准备推送自己的进展上去，却收到 Josie 的来信，说是她已经将自己的工作推到服务器上的 `featureBee` 分支了。这样，Jessica 就必须先将 Josie 的代码合并到自己本地分支中，才能再一起推送回服务器。她用 `git fetch` 下载 Josie 的最新代码：

```
$ git fetch origin
    ...
    From jessica@githost:simplegit
    * [new branch] featureBee -> origin/featureBee
```

然后 Jessica 使用 `git merge` 将此分支合并到自己分支中：

```
$ git merge origin/featureBee
    Auto-merging lib/simplegit.rb
    Merge made by recursive.
    lib/simplegit.rb | 4 ++++
    1 files changed, 4 insertions(+), 0 deletions(-)
```

合并很顺利，但另外有个小问题：她要推送自己的 `featureB` 分支到服务器上的 `featureBee` 分支上去。当然，她可以使用冒号（:）格式指定目标分支：

```
$ git push origin featureB:featureBee
    ...
    To jessica@githost:simplegit.git
    fba9af8..cd685d1 featureB -> featureBee
```

我们称此为*refspec*。更多有关于 Git refspec 的讨论和使用方式会在第九章作详细阐述。

接下来，John 发邮件给 Jessica 告诉她，他看了之后作了些修改，已经推回服务器 `featureA` 分支，请她过目下。于是 Jessica 运行 `git fetch` 下载最新数据：

```
$ git fetch origin
    ...
    From jessica@githost:simplegit
    3300904..aad881d featureA -> origin/featureA
```

接下来便可以用 `git log` 查看更新了些什么：

```
$ git log origin/featureA ^featureA
    commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
    Author: John Smith <jsmith@example.com>
    Date: Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

最后，她将 John 的工作合并到自己的 `featureA` 分支中：

```
$ git checkout featureA
    Switched to branch "featureA"
    $ git merge origin/featureA
    Updating 3300904..aad881d
    Fast forward
    lib/simplegit.rb | 10 +++++++++-
    1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica 稍做一番修整后同步到服务器：

```
$ git commit -am 'small tweak'
    [featureA ed774b3] small tweak
    1 files changed, 1 insertions(+), 1 deletions(-)
    $ git push origin featureA
    ...
    To jessica@githost:simplegit.git
    3300904..ed774b3 featureA -> featureA
```

现在的 Jessica 提交历史如图 5-13 所示：

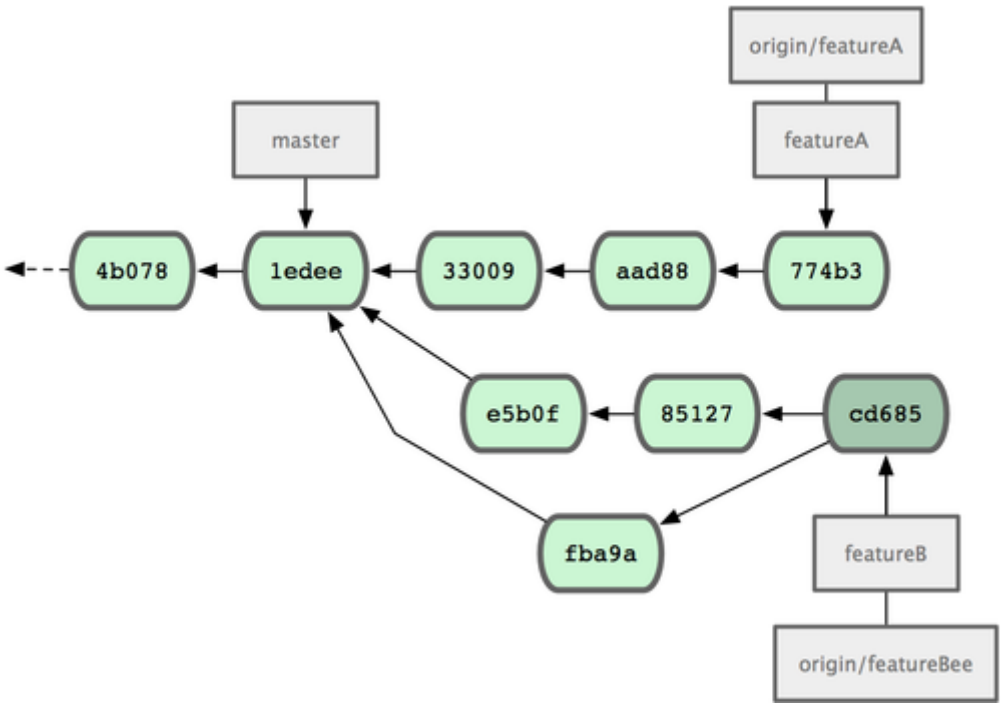

图 5-13. 在特性分支中提交更新后的提交历史

现在，Jessica，Josie 和 John 通知集成管理员服务器上的 `featureA` 及 `featureBee` 分支已经准备好，可以并入主线了。在管理员完成集成工作后，主分支上便多出一个新的合并提交（5399e），用 fetch 命令更新到本地后，提交历史如图 5-14 所示：

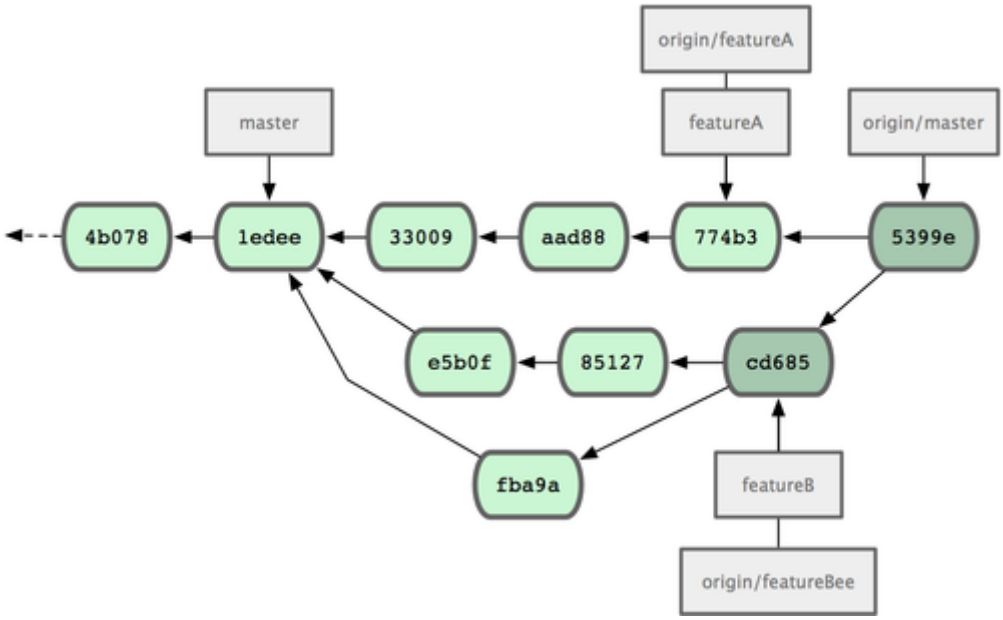

图 5-14. 合并特性分支后的 Jessica 提交历史

许多开发小组改用 Git 就是因为它允许多个小组间并行工作，而在稍后恰当时机再行合并。通过共享远程分支的方式，无需干扰整体项目代码便可以开展工作，因此使用 Git 的小型团队间协作可以变得非常灵活自由。以上工作流程的时序如图 5-15 所示：

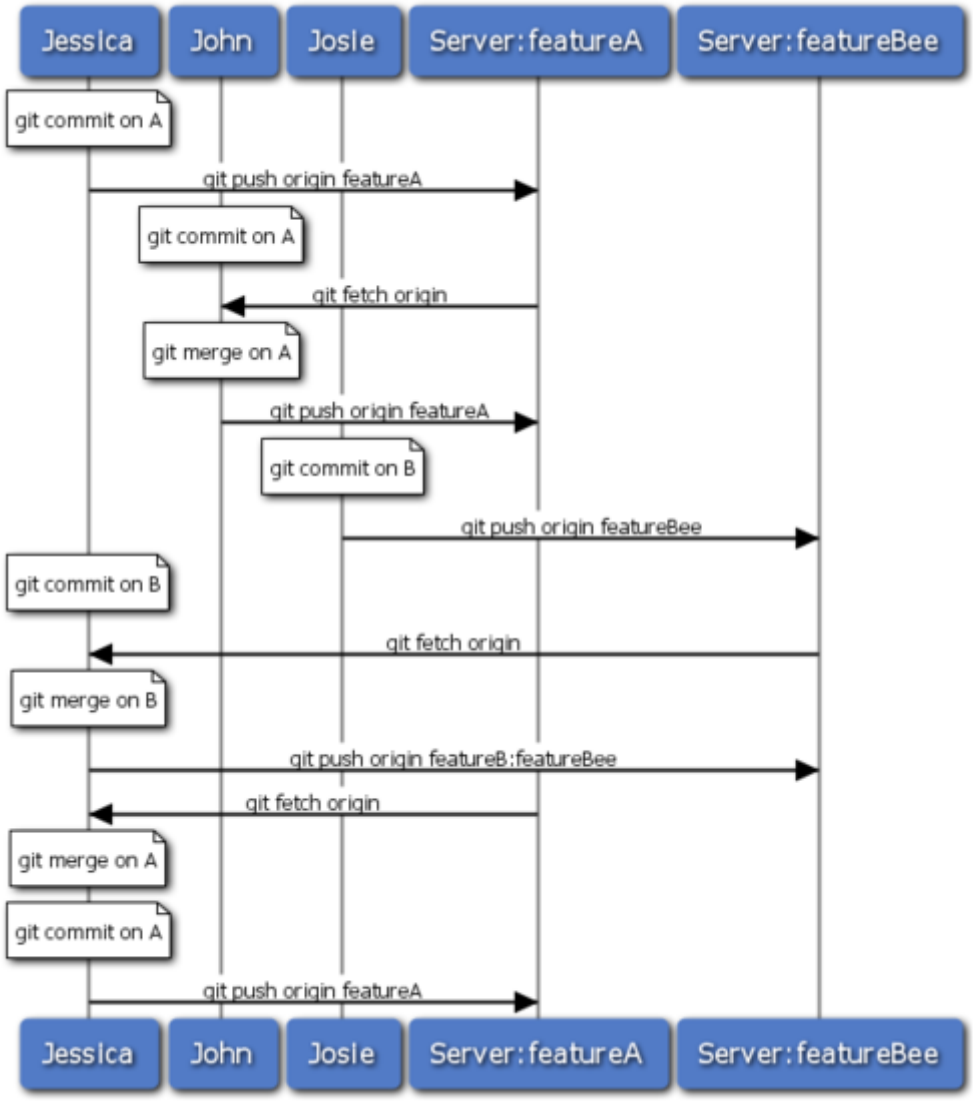

图 5-15. 团队间协作工作流程基本时序

公开的小型项目

上面说的是私有项目协作，但要给公开项目作贡献，情况就有些不同了。因为你没有直接更新主仓库分支的权限，得寻求其它方式把工作成果交给项目维护人。下面会介绍两种方法，第一种使用 git 托管服务商提供的仓库复制功能，一般称作 fork，比如 repo.or.cz 和 GitHub 都支持这样的操作，而且许多项目管理员都希望大家使用这样的方式。另一种方法是通过电子邮件寄送文件补丁。

但不管哪种方式，起先我们总需要克隆原始仓库，而后创建特性分支开展工作。基本工作流程如下：

```
$ git clone (url)
    $ cd project
    $ git checkout -b featureA
    $ (work)
    $ git commit
    $ (work)
    $ git commit
```

你可能想到用 `rebase -i` 将所有更新先变作单个提交，又或者想重新安排提交之间的差异补丁，以方便项目维护者审阅 -- 有关交互式衍合操作的细节见第六章。

在完成了特性分支开发，提交给项目维护者之前，先到原始项目的页面上点击“Fork”按钮，创建一个自己可写的公共仓库（译注：即下面的 url 部分，参照后续的例子，应该是 `git://githost/simplegit.git`）。然后将此仓库添加为本地的第二个远端仓库，姑且称为 `myfork`：

```
$ git remote add myfork (url)
```

你需要将本地更新推送到这个仓库。要是将远端 master 合并到本地再推回去，还不如把整个特性分支推上去来得干脆直接。而且，假若项目维护者未采纳你的贡献的话（不管是直接合并还是 cherry pick），都不用回退（rewind）自己的 master 分支。但若维护者合并或 cherry-pick 了你的工作，最后总还可以从他们的更新中同步这些代码。好吧，现在先把 featureA 分支整个推上去：

```
$ git push myfork featureA
```

然后通知项目管理员，让他来抓取你的代码。通常我们把这件事叫做 pull request。可以直接用 GitHub 等网站提供的 “pull request” 按钮自动发送请求通知；或手工把 `git request-pull` 命令输出结果电邮给项目管理员。

`request-pull` 命令接受两个参数，第一个是本地特性分支开始前的原始分支，第二个是请求对方来抓取的 Git 仓库 URL（译注：即下面 `myfork` 所指的，自己可写的公共仓库）。比如现在Jessica 准备要给 John 发一个 pull requst，她之前在自己的特性分支上提交了两次更新，并把分支整个推到了服务器上，所以运行该命令会看到：

```
$ git request-pull origin/master myfork
    The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
    John Smith (1):
    added a new function

    are available in the git repository at:

    git://githost/simplegit.git featureA

    Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25
```

```
    lib/simplegit.rb | 10 +++++++++-
    1 files changed, 9 insertions(+), 1 deletions(-)
```

输出的内容可以直接发邮件给管理者，他们就会明白这是从哪次提交开始旁支出去的，该到哪里去抓取新的代码，以及新的代码增加了哪些功能等等。

像这样随时保持自己的 `master` 分支和官方 `origin/master` 同步，并将自己的工作限制在特性分支上的做法，既方便又灵活，采纳和丢弃都轻而易举。就算原始主干发生变化，我们也能重新衍合提供新的补丁。比如现在要开始第二项特性的开发，不要在原来已推送的特性分支上继续，还是按原始 `master` 开始：

```
$ git checkout -b featureB origin/master
  $ (work)
  $ git commit
  $ git push myfork featureB
  $ (email maintainer)
  $ git fetch origin
```

现在，A、B 两个特性分支各不相扰，如同竹筒里的两颗豆子，队列中的两个补丁，你随时都可以分别从头写过，或者衍合，或者修改，而不用担心特性代码的交叉混杂。如图 5-16 所示：

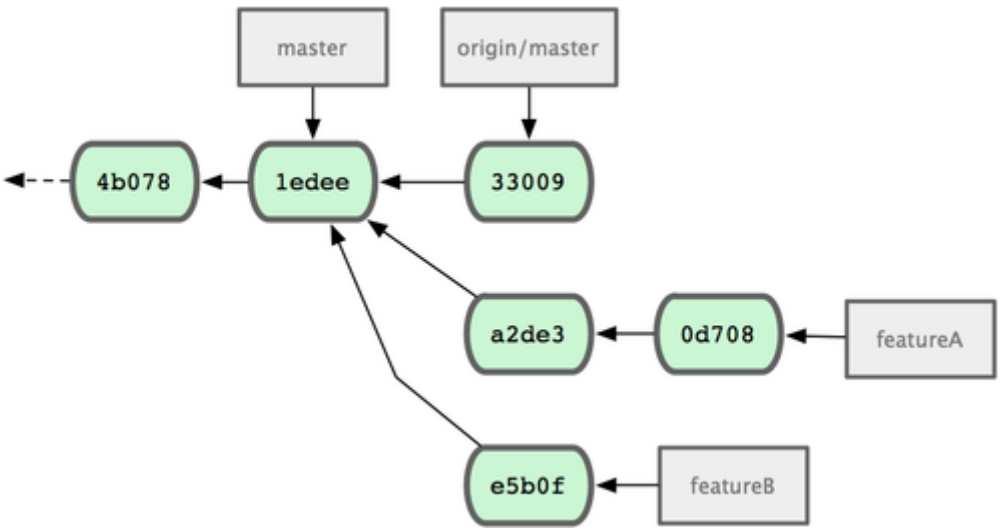

图 5-16. featureB 以后的提交历史

假设项目管理员接纳了许多别人提交的补丁后，准备要采纳你提交的第一个分支，却发现因为代码基准不一致，合并工作无法正确干净地完成。这就需要你再次衍合到最新的 `origin/master`，解决相关冲突，然后重新提交你的修改：

```
$ git checkout featureA
  $ git rebase origin/master
  $ git push -f myfork featureA
```

自然，这会重写提交历史，如图 5-17 所示：

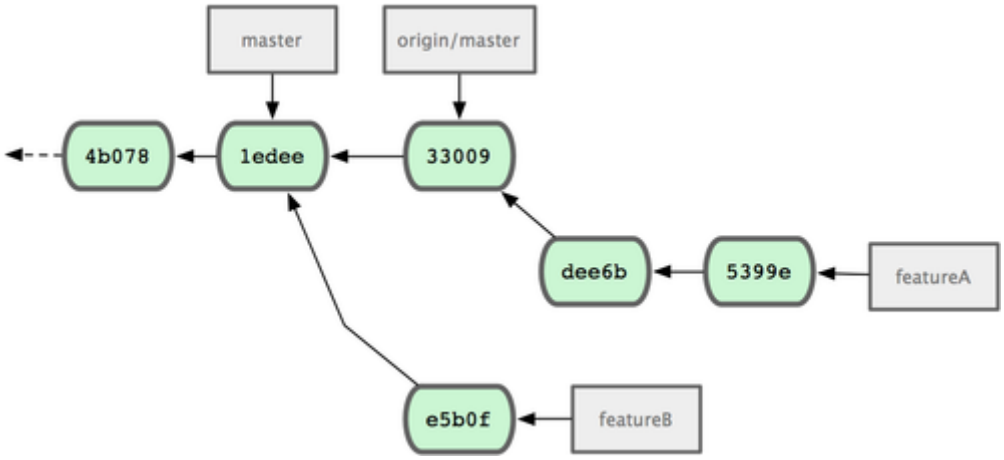

图 5-17. featureA 重新衍合后的提交历史

注意，此时推送分支必须使用 `-f` 选项（译注：表示 force，不作检查强制重写）替换远程已有的 `featureA` 分支，因为新的 commit 并非原来的后续更新。当然你也可以直接推送到另一个新的分支上去，比如称作 `featureAv2`。

再考虑另一种情形：管理员看过第二个分支后觉得思路新颖，但想请你改下具体实现。我们只需以当前 `origin/master` 分支为基准，开始一个新的特性分支 `featureBv2`，然后把原来的 `featureB` 的更新拿过来，解决冲突，按要求重新实现部分代码，然后将此特性分支推送上去：

```
$ git checkout -b featureBv2 origin/master
    $ git merge --no-commit --squash featureB
    $ (change implementation)
    $ git commit
    $ git push myfork featureBv2
```

这里的 `--squash` 选项将目标分支上的所有更改全拿来应用到当前分支上，而 `--no-commit` 选项告诉 Git 此时无需自动生成和记录（合并）提交。这样，你就可以在原来代码基础上，继续工作，直到最后一起提交。

好了，现在可以请管理员抓取 `featureBv2` 上的最新代码了，如图 5-18 所示：

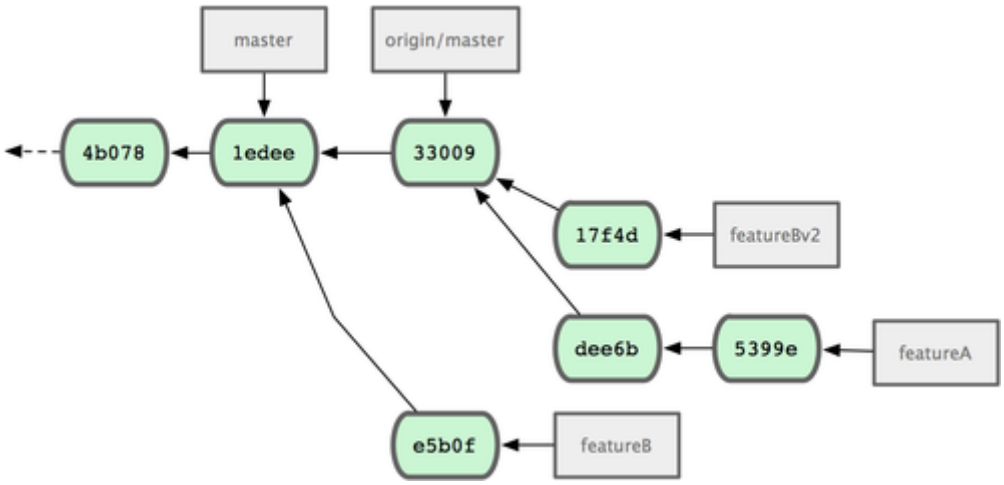

图 5-18. featureBv2 之后的提交历史

公开的大型项目

许多大型项目都会立有一套自己的接受补丁流程，你应该注意下其中细节。但多数项目都允许通过开发者邮件列表接受补丁，现在我们来看具体例子。

整个工作流程类似上面的情形：为每个补丁创建独立的特性分支，而不同之处在于如何提交这些补丁。不需要创建自己可写的公共仓库，也不用将自己的更新推送到自己的服务器，你只需将每次提交的差异内容以电子邮件的方式依次发送到邮件列表中即可。

```
$ git checkout -b topicA
    $ (work)
    $ git commit
    $ (work)
    $ git commit
```

如此一番后，有了两个提交要发到邮件列表。我们可以用 `git format-patch` 命令来生成 mbox 格式的文件然后作为附件发送。每个提交都会封装为一个 `.patch` 后缀的 mbox 文件，但其中只包含一封邮件，邮件标题就是提交消息（译注：额外有前缀，看例子），邮件内容包含补丁正文和 Git 版本号。这种方式的妙处在于接受补丁时仍可保留原来的提交消息，请看接下来的例子：

```
$ git format-patch -M origin/master
    0001-add-limit-to-log-function.patch
    0002-changed-log-output-to-30-from-25.patch
```

`format-patch` 命令依次创建补丁文件，并输出文件名。上面的 `-M` 选项允许 Git 检查是否有对文件重命名的提交。我们来看看补丁文件的内容：

```
$ cat 0001-add-limit-to-log-function.patch
    From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
    From: Jessica Smith <jessica@example.com>
    Date: Sun, 6 Apr 2008 10:17:23 -0700
    Subject: [PATCH 1/2] add limit to log function

    Limit log functionality to the first 20

    ---
    lib/simplegit.rb | 2 +-
    1 files changed, 1 insertions(+), 1 deletions(-)

    diff --git a/lib/simplegit.rb b/lib/simplegit.rb
    index 76f47bc..f9815f1 100644
    --- a/lib/simplegit.rb
    +++ b/lib/simplegit.rb
    @@ -14,7 +14,7 @@ class SimpleGit
    end

    def log(treeish = 'master')
    - command("git log #{treeish}")
    + command("git log -n 20 #{treeish}")
    end

    def ls_tree(treeish = 'master')
    --
    1.6.2.rc1.20.g8c5b.dirty
```

如果有额外信息需要补充，但又不想放在提交消息中说明，可以编辑这些补丁文件，在第一个 `---` 行之前添加说明，但不要修改下面的补丁正文，比如例子中的 `Limit log functionality to the first 20` 部分。这样，其它开发者能阅读，但在采纳补丁时不会将此合并进来。

你可以用邮件客户端软件发送这些补丁文件，也可以直接在命令行发送。有些所谓智能的邮件客户端软件会自作主张帮你调整格式，所以粘贴补丁到邮件正文时，有可能会丢失换行符和若干空格。

Git 提供了一个通过 IMAP 发送补丁文件的工具。接下来我会演示如何通过 Gmail 的 IMAP 服务器发送。另外，在 Git 源代码中有个 `Documentation/SubmittingPatches` 文件，可以仔细读读，看看其它邮件程序的相关导引。

首先在 `~/.gitconfig` 文件中配置 imap 项。每个选项都可用 `git config` 命令分别设置，当然直接编辑文件添加以下内容更便捷：

```
[imap]
    folder = "[Gmail]/Drafts"
    host = imaps://imap.gmail.com
    user = user@gmail.com
    pass = p4ssw0rd
    port = 993
    sslverify = false
```

如果你的 IMAP 服务器没有启用 SSL，就无需配置最后那两行，并且 host 应该以 `imap://` 开头而不再是有 `s` 的 `imaps://`。保存配置文件后，就能用 `git send-email` 命令把补丁作为邮件依次发送到指定的 IMAP 服务器上的文件夹中（译注：这里就是 Gmail 的 `[Gmail]/Drafts` 文件夹。但如果你的语言设置不是英文，此处的文件夹 Drafts 字样会变为对应的语言。）：

```
$ git send-email *.patch
    0001-added-limit-to-log-function.patch
    0002-changed-log-output-to-30-from-25.patch
    Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
    Emails will be sent from: Jessica Smith <jessica@example.com>
    Who should the emails be sent to? jessica@example.com
    Message-ID to be used as In-Reply-To for the first email? y
```

接下来，Git 会根据每个补丁依次输出类似下面的日志：

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
    \line 'From: Jessica Smith <jessica@example.com>'
    OK. Log says:
    Sendmail: /usr/sbin/sendmail -i jessica@example.com
    From: Jessica Smith <jessica@example.com>
    To: jessica@example.com
    Subject: [PATCH 1/2] added limit to log function
    Date: Sat, 30 May 2009 13:29:15 -0700
    Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
    X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
    In-Reply-To: <y>
    References: <y>

    Result: OK
```

最后，到 Gmail 上打开 Drafts 文件夹，编辑这些邮件，修改收件人地址为邮件列表地址，另外给要抄送的人也加到 Cc 列表中，最后发送。

小结

本节主要介绍了常见 Git 项目协作的工作流程，还有一些帮助处理这些工作的命令和工具。接下来我们要看看如何维护 Git 项目，并成为一个合格的项目管理员，或是集成经理。

5.3 项目的管理

既然是相互协作，在贡献代码的同时，也免不了要维护管理自己的项目。像是怎么处理别人用 `format-patch` 生成的补丁，或是集成远端仓库上某个分支上的变化等等。但无论是管理代码仓库，还

是帮忙审核收到的补丁，都需要同贡献者约定某种长期可持续的工作方式。

使用特性分支进行工作

如果想要集成新的代码进来，最好局限在特性分支上做。临时的特性分支可以让你随意尝试，进退自如。比如碰上无法正常工作的补丁，可以先搁在那边，直到有时间仔细核查修复为止。创建的分支可以用相关的主题关键字命名，比如 `ruby_client` 或者其它类似的描述性词语，以帮助将来回忆。Git 项目本身还时常把分支名称分置于不同命名空间下，比如 `sc/ruby_client` 就说明这是 `sc` 这个人贡献的。现在从当前主干分支为基础，新建临时分支：

```
$ git branch sc/ruby_client master
```

另外，如果你希望立即转到分支上去工作，可以用 `checkout -b`：

```
$ git checkout -b sc/ruby_client master
```

好了，现在已经准备妥当，可以试着将别人贡献的代码合并进来了。之后评估一下有没有问题，最后再决定是不是真的要并入主干。

采纳来自邮件的补丁

如果收到一个通过电邮发来的补丁，你应该先把它应用到特性分支上进行评估。有两种应用补丁的方法：`git apply` 或者 `git am`。

使用 apply 命令应用补丁

如果收到的补丁文件是用 `git diff` 或由其它 Unix 的 `diff` 命令生成，就该用 `git apply` 命令来应用补丁。假设补丁文件存在 `/tmp/patch-ruby-client.patch`，可以这样运行：

```
$ git apply /tmp/patch-ruby-client.patch
```

这会修改当前工作目录下的文件，效果基本与运行 `patch -p1` 打补丁一样，但它更为严格，且不会出现混乱。如果是 `git diff` 格式描述的补丁，此命令还会相应地添加，删除，重命名文件。当然，普通的 `patch` 命令是不会这么做的。另外请注意，`git apply` 是一个事务性操作的命令，也就是说，要么所有补丁都打上去，要么全部放弃。所以不会出现 `patch` 命令那样，一部分文件打上了补丁而另一部分却没有，这样一种不上不下的修订状态。所以总的来说，`git apply` 要比 `patch` 严谨许多。因为仅仅是更新当前的文件，所以此命令不会自动生成提交对象，你得手工缓存相应文件的更新状态并执行提交命令。

在实际打补丁之前，可以先用 `git apply --check` 查看补丁是否能够干净顺利地应用到当前分支中：

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
    error: patch failed: ticgit.gemspec:1
    error: ticgit.gemspec: patch does not apply
```

如果没有任何输出，表示我们可以顺利采纳该补丁。如果有问题，除了报告错误信息之外，该命令还会返回一个非零的状态，所以在 shell 脚本里可用于检测状态。

使用 am 命令应用补丁

如果贡献者也用 Git，且擅于制作 format-patch 补丁，那你的合并工作将会非常轻松。因为这些补丁中除了文件内容差异外，还包含了作者信息和提交消息。所以请鼓励贡献者用 format-patch 生成补丁。对于传统的 diff 命令生成的补丁，则只能用 git apply 处理。

对于 format-patch 制作的新式补丁，应当使用 git am 命令。从技术上来说，git am 能够读取 mbox 格式的文件。这是种简单的纯文本文件，可以包含多封电邮，格式上用 From 加空格以及随便什么辅助信息所组成的行作为分隔行，以区分每封邮件，就像这样：

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
    From: Jessica Smith <jessica@example.com>
    Date: Sun, 6 Apr 2008 10:17:23 -0700
    Subject: [PATCH 1/2] add limit to log function

    Limit log functionality to the first 20
```

这是 format-patch 命令输出的开头几行，也是一个有效的 mbox 文件格式。如果有人用 git send-email 给你发了一个补丁，你可以将此邮件下载到本地，然后运行 git am 命令来应用这个补丁。如果你的邮件客户端能将多封电邮导出为 mbox 格式的文件，就可以用 git am 一次性应用所有导出的补丁。

如果贡献者将 format-patch 生成的补丁文件上传到类似 Request Ticket 一样的任务处理系统，那么可以先下载到本地，继而使用 git am 应用该补丁：

```
$ git am 0001-limit-log-function.patch
    Applying: add limit to log function
```

你会看到它被干净地应用到本地分支，并自动创建了新的提交对象。作者信息取自邮件头 From 和 Date，提交消息则取自 Subject 以及正文中补丁之前的内容。来看具体实例，采纳之前展示的那个 mbox 电邮补丁后，最新的提交对象为：

```
$ git log --pretty=fuller -1
    commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
    Author: Jessica Smith <jessica@example.com>
    AuthorDate: Sun Apr 6 10:17:23 2008 -0700
    Commit: Scott Chacon <schacon@gmail.com>
    CommitDate: Thu Apr 9 09:19:06 2009 -0700

    add limit to log function

    Limit log functionality to the first 20
```

Commit 部分显示的是采纳补丁的人，以及采纳的时间。而 Author 部分则显示的是原作者，以及创建补丁的时间。

有时，我们也会遇到打不上补丁的情况。这多半是因为主干分支和补丁的基础分支相差太远，但也可能是因为某些依赖补丁还未应用。这种情况下，git am 会报错并询问该怎么做：

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
    Applying: seeing if this helps the gem
    error: patch failed: ticgit.gemspec:1
    error: ticgit.gemspec: patch does not apply
    Patch failed at 0001.
    When you have resolved this problem run "git am --resolved".
    If you would prefer to skip this patch, instead run "git am --skip".
    To restore the original branch and stop patching run "git am --abort".
```

Git 会在有冲突的文件里加入冲突解决标记，这同合并或衍合操作一样。解决的办法也一样，先编辑文件消除冲突，然后暂存文件，最后运行 `git am --resolved` 提交修正结果：

```
$ (fix the file)
    $ git add ticgit.gemspec
    $ git am --resolved
    Applying: seeing if this helps the gem
```

如果想让 Git 更智能地处理冲突，可以用 `-3` 选项进行三方合并。如果当前分支未包含该补丁的基础代码或其祖先，那么三方合并就会失败，所以该选项默认为关闭状态。一般来说，如果该补丁是基于某个公开的提交制作而成的话，你总是可以通过同步来获取这个共同祖先，所以用三方合并选项可以解决很多麻烦：

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
    Applying: seeing if this helps the gem
    error: patch failed: ticgit.gemspec:1
    error: ticgit.gemspec: patch does not apply
    Using index info to reconstruct a base tree...
    Falling back to patching base and 3-way merge...
    No changes -- Patch already applied.
```

像上面的例子，对于打过的补丁我又再打一遍，自然会产生冲突，但因为加上了 `-3` 选项，所以它很聪明地告诉我，无需更新，原有的补丁已经应用。

对于一次应用多个补丁时所用的 mbox 格式文件，可以用 `am` 命令的交互模式选项 `-i`，这样就会在打每个补丁前停住，询问该如何操作：

```
$ git am -3 -i mbox
    Commit Body is:
    --------------------------
    seeing if this helps the gem
    --------------------------
    Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

在多个补丁要打的情况下，这是个非常好的办法，一方面可以预览下补丁内容，同时也可以有选择性的接纳或跳过某些补丁。

打完所有补丁后，如果测试下来新特性可以正常工作，那就可以安心地将当前特性分支合并到长期分支中去了。

检出远程分支

如果贡献者有自己的 Git 仓库，并将修改推送到此仓库中，那么当你拿到仓库的访问地址和对应分支的名称后，就可以加为远程分支，然后在本地进行合并。

比如，Jessica 发来一封邮件，说在她代码库中的 `ruby-client` 分支上已经实现了某个非常棒的新功能，希望我们能帮忙测试一下。我们可以先把她的仓库加为远程仓库，然后抓取数据，完了再将她所说的分支检出到本地来测试：

```
$ git remote add jessica git://github.com/jessica/myproject.git
    $ git fetch jessica
    $ git checkout -b rubyclient jessica/ruby-client
```

若是不久她又发来邮件，说还有个很棒的功能实现在另一分支上，那我们只需重新抓取下最新数据，然后检出那个分支到本地就可以了，无需重复设置远程仓库。

这种做法便于同别人保持长期的合作关系。但前提是要求贡献者有自己的服务器，而我们也需要为每个人建一个远程分支。有些贡献者提交代码补丁并不是很频繁，所以通过邮件接收补丁效率会更高。同时我们自己也不会希望建上百来个分支，却只从每个分支取一两个补丁。但若是用脚本程序来管理，或直接使用代码仓库托管服务，就可以简化此过程。当然，选择何种方式取决于你和贡献者的喜好。

使用远程分支的另外一个好处是能够得到提交历史。不管代码合并是不是会有问题，至少我们知道该分支的历史分叉点，所以默认会从共同祖先开始自动进行三方合并，无需 `-3` 选项，也不用像打补丁那样祈祷存在共同的基准点。

如果只是临时合作，只需用 `git pull` 命令抓取远程仓库上的数据，合并到本地临时分支就可以了。一次性的抓取动作自然不会把该仓库地址加为远程仓库。

```
$ git pull git://github.com/onetimeguy/project.git
    From git://github.com/onetimeguy/project
    * branch HEAD -> FETCH_HEAD
    Merge made by recursive.
```

决断代码取舍

现在特性分支上已合并好了贡献者的代码，是时候决断取舍了。本节将回顾一些之前学过的命令，以看清将要合并到主干的是哪些代码，从而理解它们到底做了些什么，是否真的要并入。

一般我们会先看下，特性分支上都有哪些新增的提交。比如在 `contrib` 特性分支上打了两个补丁，仅查看这两个补丁的提交信息，可以用 `--not` 选项指定要屏蔽的分支 `master`，这样就会剔除重复的提交历史：

```
$ git log contrib --not master
    commit 5b6235bd297351589efc4d73316f0a68d484f118
    Author: Scott Chacon <schacon@gmail.com>
    Date: Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

    commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
    Author: Scott Chacon <schacon@gmail.com>
    Date: Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

还可以查看每次提交的具体修改。请牢记，在 `git log` 后加 `-p` 选项将展示每次提交的内容差异。

如果想看当前分支同其他分支合并时的完整内容差异，有个小窍门：

```
$ git diff master
```

虽然能得到差异内容，但请记住，结果有可能和我们的预期不同。一旦主干 `master` 在特性分支创建之后有所修改，那么通过 `diff` 命令来比较的，是最新主干上的提交快照。显然，这不是我们所要的。比方在 `master` 分支中某个文件里添了一行，然后运行上面的命令，简单的比较最新快照所得到的结论只能是，特性分支中删除了这一行。

这个很好理解：如果 `master` 是特性分支的直接祖先，不会产生任何问题；如果它们的提交历史在不同的分叉上，那么产生的内容差异，看起来就像是增加了特性分支上的新代码，同时删除了 `master` 分支上的新代码。

实际上我们真正想要看的，是新加入到特性分支的代码，也就是合并时会并入主干的代码。所以，准确地讲，我们应该比较特性分支和它同 `master` 分支的共同祖先之间的差异。

我们可以手工定位它们的共同祖先，然后与之比较：

```
$ git merge-base contrib master
    36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
    $ git diff 36c7db
```

但这么做很麻烦，所以 Git 提供了便捷的 `...` 语法。对于 `diff` 命令，可以把 `...` 加在原始分支（拥有共同祖先）和当前分支之间：

```
$ git diff master...contrib
```

现在看到的，就是实际将要引入的新代码。这是一个非常有用的命令，应该牢记。

代码集成

一旦特性分支准备停当，接下来的问题就是如何集成到更靠近主线的分支中。此外还要考虑维护项目的总体步骤是什么。虽然有很多选择，不过我们这里只介绍其中一部分。

合并流程

一般最简单的情形，是在 `master` 分支中维护稳定代码，然后在特性分支上开发新功能，或是审核测试别人贡献的代码，接着将它并入主干，最后删除这个特性分支，如此反复。来看示例，假设当前代码库中有两个分支，分别为 `ruby_client` 和 `php_client`，如图 5-19 所示。然后先把 `ruby_client` 合并进主干，再合并 `php_client`，最后的提交历史如图 5-20 所示。

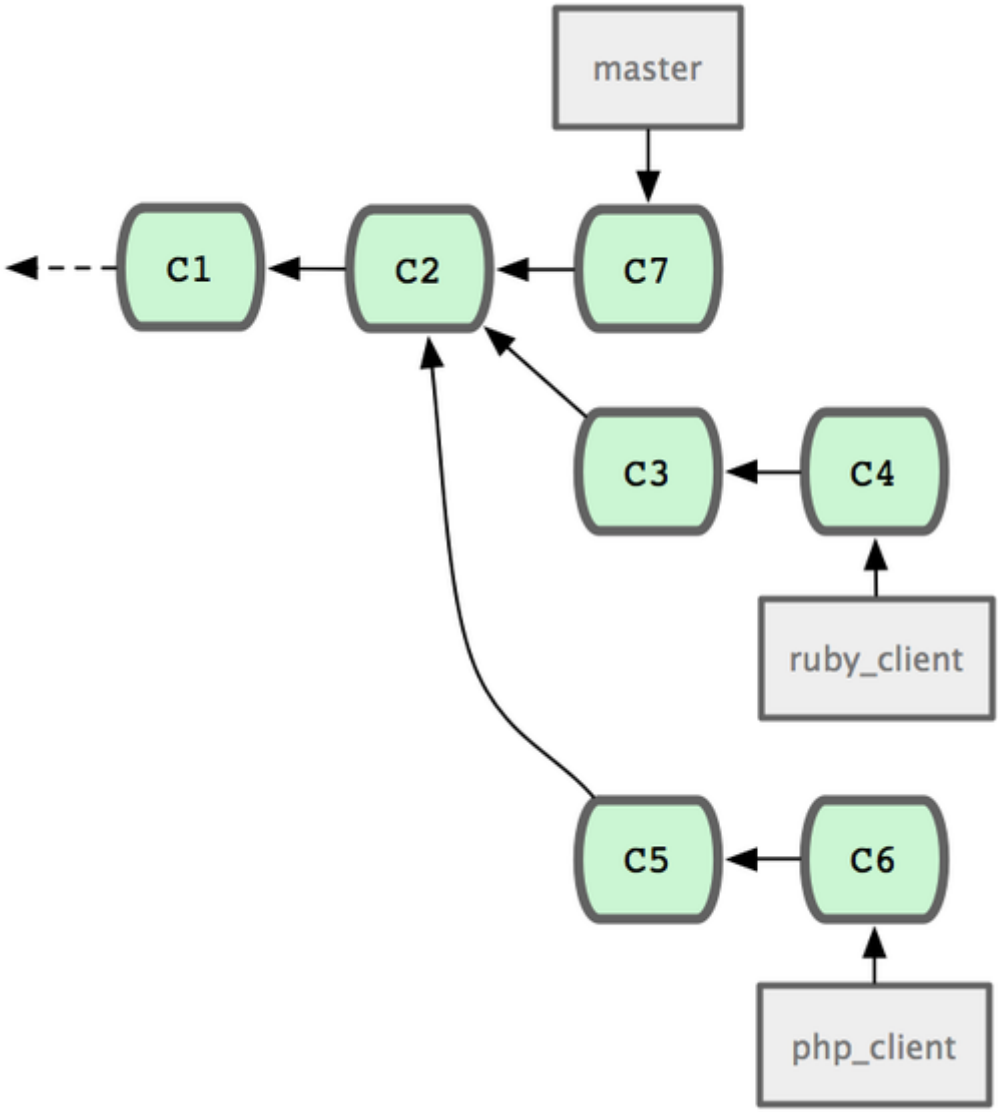

图 5-19. 多个特性分支

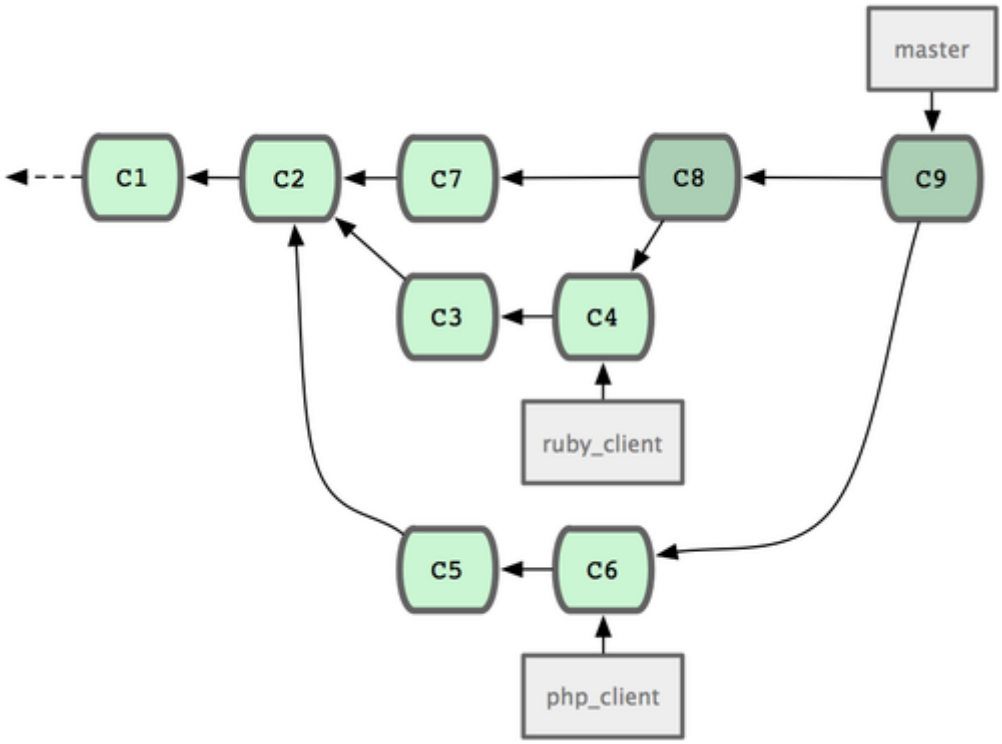

图 5-20. 合并特性分支之后

这是最简单的流程，所以在处理大一些的项目时可能会有问题。

对于大型项目，至少需要维护两个长期分支 `master` 和 `develop`。新代码（图 5-21 中的 `ruby_client`）将首先并入 `develop` 分支（图 5-22 中的 `C8`），经过一个阶段，确认 `develop` 中的代码已稳定到可发行时，再将 `master` 分支快进到稳定点（图 5-23 中的 `C8`）。而平时这两个分支都会被推送到公开的代码库。

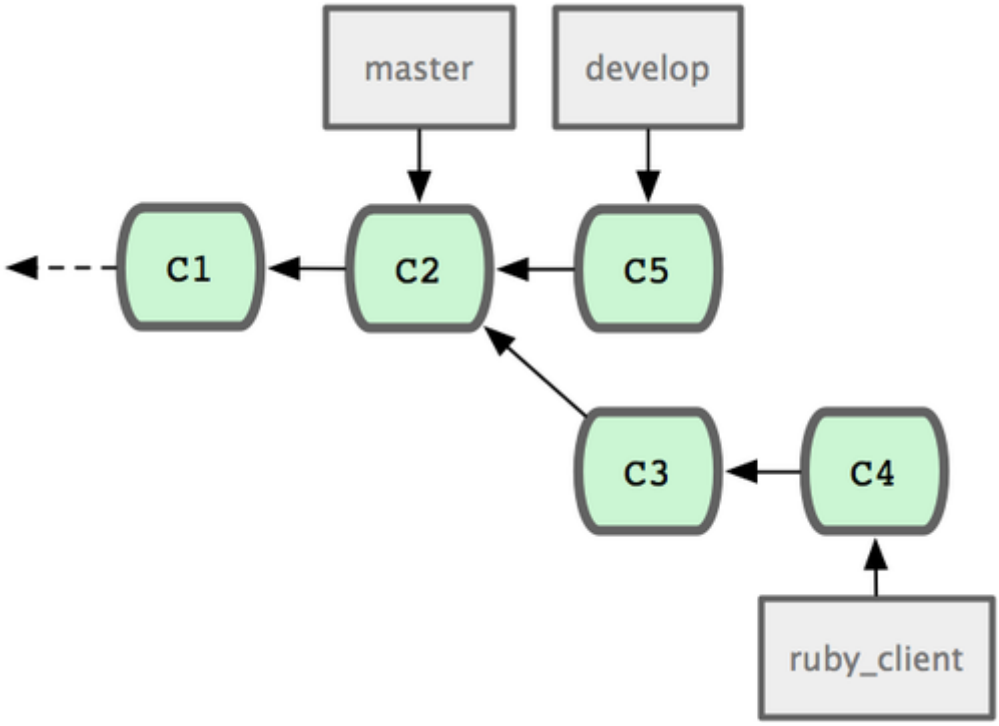

图 5-21. 特性分支合并前

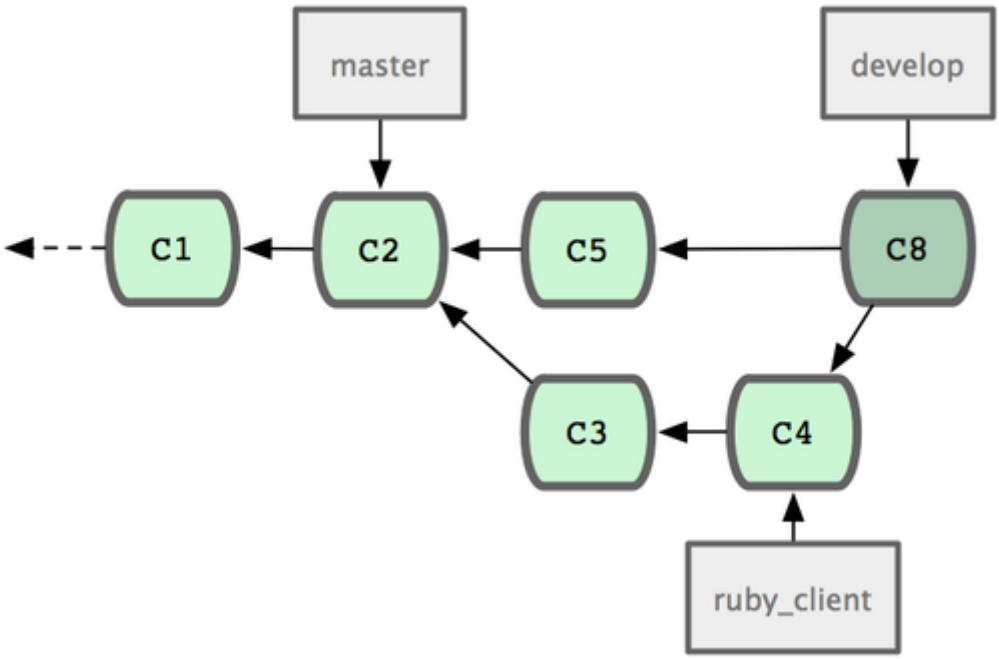

图 5-22. 特性分支合并后

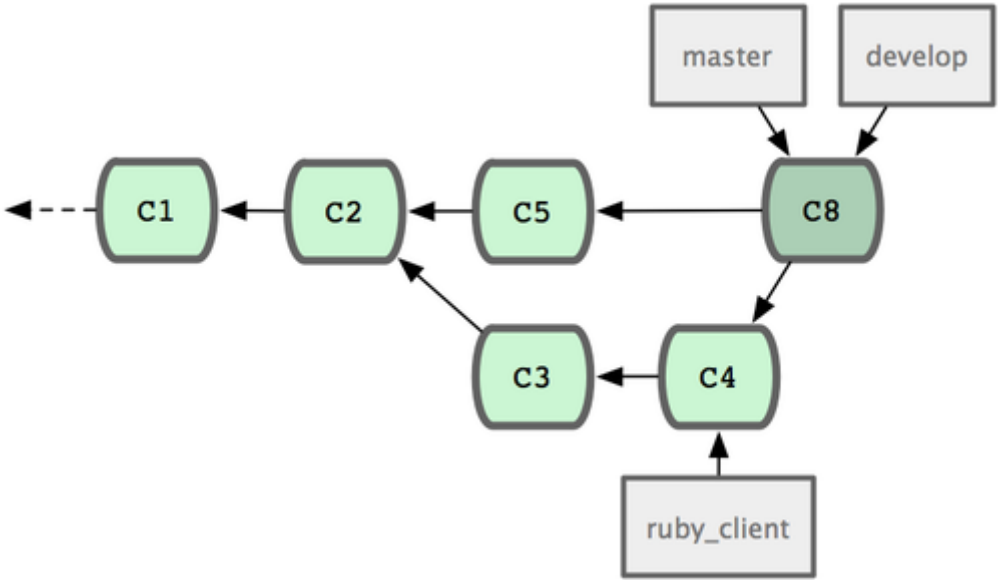

图 5-23. 特性分支发布后

这样，在人们克隆仓库时就有两种选择：既可检出最新稳定版本，确保正常使用；也能检出开发版本，试用最前沿的新特性。你也可以扩展这个概念，先将所有新代码合并到临时特性分支，等到该分支稳定下来并通过测试后，再并入 `develop` 分支。然后，让时间检验一切，如果这些代码确实可以正常工作相当长一段时间，那就有理由相信它已经足够稳定，可以放心并入主干分支发布。

大项目的合并流程

Git 项目本身有四个长期分支：用于发布的 `master` 分支、用于合并基本稳定特性的 `next` 分支、用于合并仍需改进特性的 `pu` 分支（pu 是 proposed updates 的缩写），以及用于除错维护的 `maint` 分支（maint 取自 maintenance）。维护者可以按照之前介绍的方法，将贡献者的代码引入为不同的特性分支（如图 5-24 所示），然后测试评估，看哪些特性能稳定工作，哪些还需改进。稳定的特性可以并入 `next` 分支，然后再推送到公共仓库，以供其他人试用。

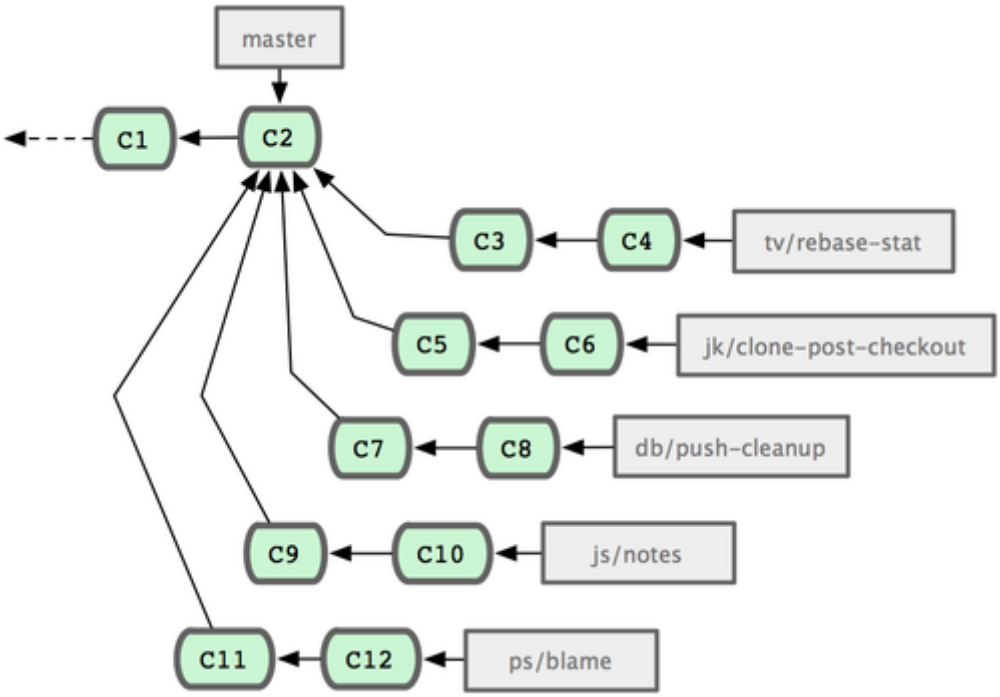

图 5-24. 管理复杂的并行贡献

仍需改进的特性可以先并入 `pu` 分支。直到它们完全稳定后再并入 `master`。同时一并检查下 `next` 分支，将足够稳定的特性也并入 `master`。所以一般来说，`master` 始终是在快进，`next` 偶尔做下衍合，而 `pu` 则是频繁衍合，如图 5-25 所示：

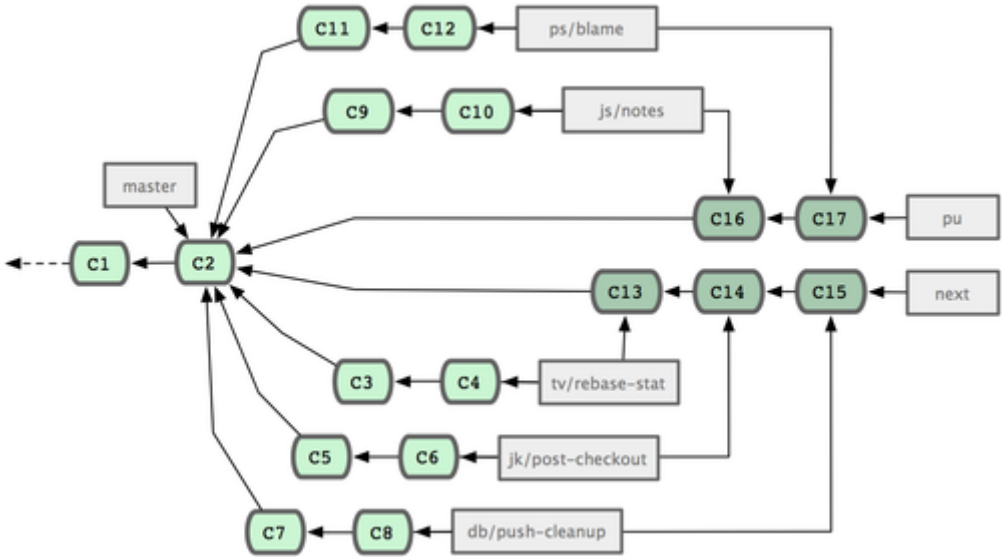

图 5-25. 将特性并入长期分支

并入 `master` 后的特性分支，已经无需保留分支索引，放心删除好了。Git 项目还有一个 `maint` 分支，它是以最近一次发行版为基础分化而来的，用于维护除错补丁。所以克隆 Git 项目仓库后会得到这四个分支，通过检出不同分支可以了解各自进展，或是试用前沿特性，或是贡献代码。而维护者则通过管理这些分支，逐步有序地并入第三方贡献。

衍合与挑拣（cherry-pick）的流程

一些维护者更喜欢衍合或者挑拣贡献者的代码，而不是简单的合并，因为这样能够保持线性的提交历史。如果你完成了一个特性的开发，并决定将它引入到主干代码中，你可以转到那个特性分支然后执行衍合命令，好在你的主干分支上（也可能是`develop`分支之类的）重新提交这些修改。如果这些代码工作得很好，你就可以快进`master`分支，得到一个线性的提交历史。

另一个引入代码的方法是挑拣。挑拣类似于针对某次特定提交的衍合。它首先提取某次提交的补丁，然后试着应用在当前分支上。如果某个特性分支上有多个提交，但你只想引入其中之一就可以使用这种方法。也可能仅仅是因为你喜欢用挑拣，讨厌衍合。假设你有一个类似图 5-26 的工程。

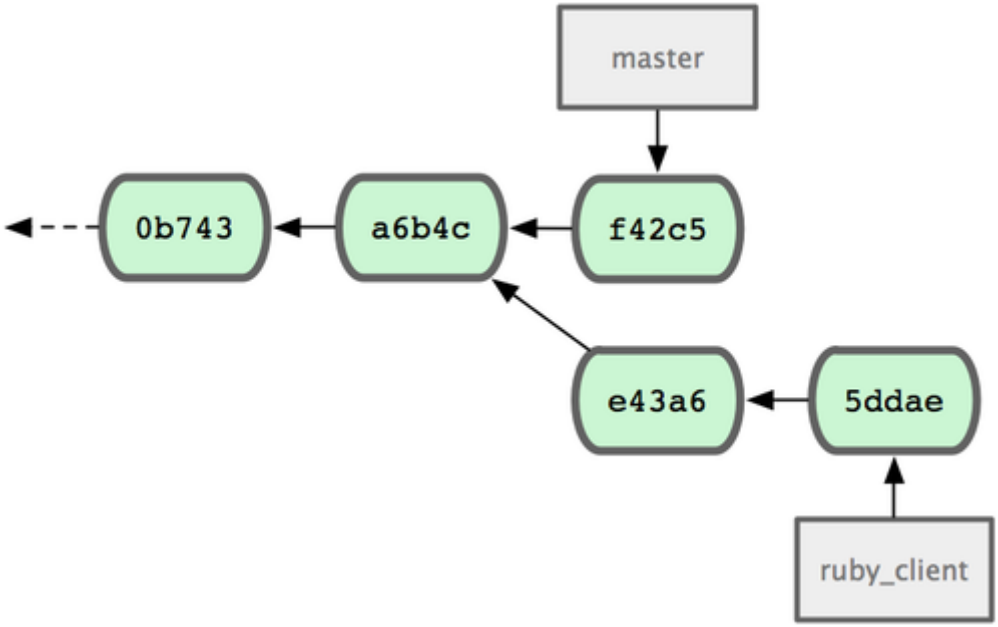

图 5-26. 挑拣（cherry-pick）之前的历史

如果你希望拉取e43a6到你的主干分支，可以这样：

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
    Finished one cherry-pick.
    [master]: created a0a41a9: "More friendly message when locking the index fails."
    3 files changed, 17 insertions(+), 3 deletions(-)
```

这将会引入e43a6的代码，但是会得到不同的SHA-1值，因为应用日期不同。现在你的历史看起来像图 5-27.

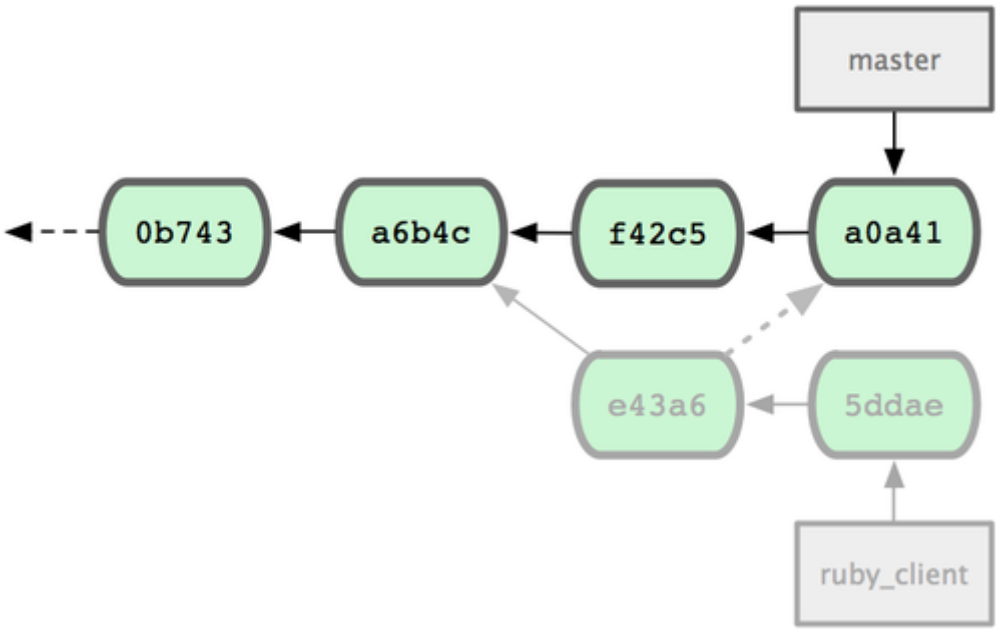

图 5-27. 挑拣（cherry-pick）之后的历史

现在，你可以删除这个特性分支并丢弃你不想引入的那些commit。

给发行版签名

你可以删除上次发布的版本并重新打标签，也可以像第二章所说的那样建立一个新的标签。如果你决定以维护者的身份给发行版签名，应该这样做：

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
    You need a passphrase to unlock the secret key for
    user: "Scott Chacon <schacon@gmail.com>"
    1024-bit DSA key, ID F721C45A, created 2009-02-09
```

完成签名之后，如何分发PGP公钥（public key）是个问题。（译者注：分发公钥是为了验证标签）。还好，Git的设计者想到了解决办法：可以把key（既公钥）作为blob变量写入Git库，然后把它的内容直接写在标签里。`gpg --list-keys`命令可以显示出你所拥有的key：

```
$ gpg --list-keys
    /Users/schacon/.gnupg/pubring.gpg
    ---------------------------------
    pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
    uid Scott Chacon <schacon@gmail.com>
    sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

然后，导出key的内容并经由管道符传递给`git hash-object`，之后钥匙会以blob类型写入Git中，最后返回这个blob量的SHA-1值：

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
    659ef797d181633c87ec71ac3f9ba29fe5775b92
```

现在你的Git已经包含了这个key的内容了，可以通过不同的SHA-1值指定不同的key来创建标签。

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

在运行`git push --tags`命令之后，`maintainer-pgp-pub`标签就会公布给所有人。如果有人想要校验标签，他可以使用如下命令导入你的key：

```
$ git show maintainer-pgp-pub | gpg --import
```

人们可以用这个key校验你签名的所有标签。另外，你也可以在标签信息里写入一个操作向导，用户只需要运行`git show <tag>`查看标签信息，然后按照你的向导就能完成校验。

生成内部版本号

因为Git不会为每次提交自动附加类似'v123'的递增序列，所以如果你想要得到一个便于理解的提交号可以运行`git describe`命令。Git将会返回一个字符串，由三部分组成：最近一次标定的版本号，加上自那次标定之后的提交次数，再加上一段SHA-1值of the commit you’re describing：

```
$ git describe master
    v1.6.2-rc1-20-g8c5b85c
```

这个字符串可以作为快照的名字，方便人们理解。如果你的Git是你自己下载源码然后编译安装的，你会发现`git --version`命令的输出和这个字符串差不多。如果在一个刚刚打完标签的提交上运行`describe`命令，只会得到这次标定的版本号，而没有后面两项信息。

`git describe`命令只适用于有标注的标签（通过`-a`或者`-s`选项创建的标签），所以发行版的标签都应该是带有标注的，以保证`git describe`能够正确的执行。你也可以把这个字符串作为`checkout`或者`show`命令的目标，因为他们最终都依赖于一个简短的SHA-1值，当然如果这个SHA-1值失效他们也跟着失效。最近Linux内核为了保证SHA-1值的唯一性，将位数由8位扩展到10位，这就导致扩展之前的`git describe`输出完全失效了。

准备发布

现在可以发布一个新的版本了。首先要将代码的压缩包归档，方便那些可怜的还没有使用Git的人们。可以使用`git archive`：

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
    $ ls *.tar.gz
    v1.6.2-rc1-20-g8c5b85c.tar.gz
```

这个压缩包解压出来的是一个文件夹，里面是你项目的最新代码快照。你也可以用类似的方法建立一个zip压缩包，在`git archive`加上`--format=zip`选项：

```
$ git archive master --prefix='project/' --format=zip > `git describe
    master`.zip
```

现在你有了一个tar.gz压缩包和一个zip压缩包，可以把他们上传到你网站上或者用e-mail发给别人。

制作简报

是时候通知邮件列表里的朋友们来检验你的成果了。使用`git shortlog`命令可以方便快捷的制作一份修改日志（changelog），告诉大家上次发布之后又增加了哪些特性和修复了哪些bug。实际上这个命令能够统计给定范围内的所有提交;假如你上一次发布的版本是v1.0.1，下面的命令将给出自从上次发布之后的所有提交的简介：

```
$ git shortlog --no-merges master --not v1.0.1
    Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils

    Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

这就是自从v1.0.1版本以来的所有提交的简介，内容按照作者分组，以便你能快速的发e-mail给他们。

5.4 小结

你学会了如何使用Git为项目做贡献，也学会了如何使用Git维护你的项目。恭喜！你已经成为一名高效的开发者。在下一章你将学到更强大的工具来处理更加复杂的问题，之后你会变成一位Git大师。

下一节上一节首页（目录）| 返回 码云