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：

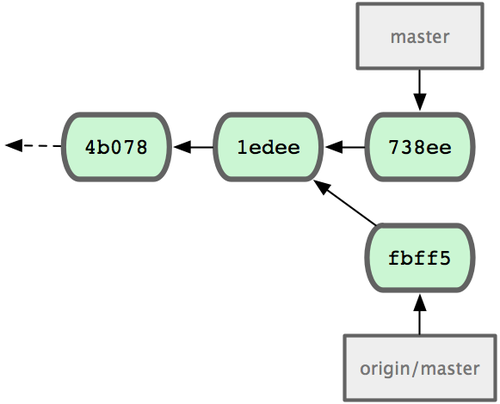
$ git fetch origin

...

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：

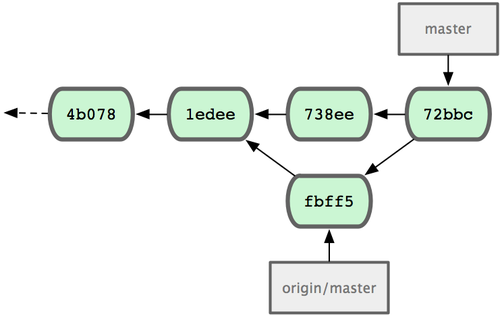
$ git merge origin/master

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：

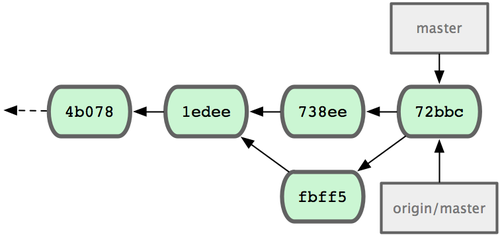
$ git push origin master

...

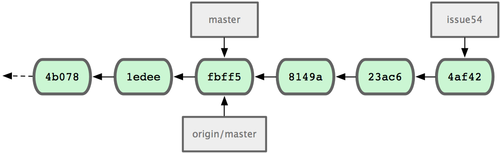
To john@githost:simplegit.git

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

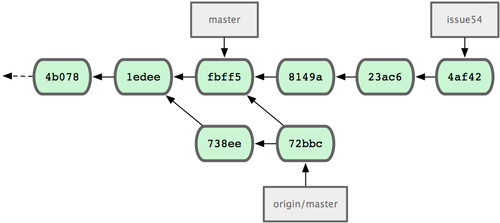
$ git fetch origin

...

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

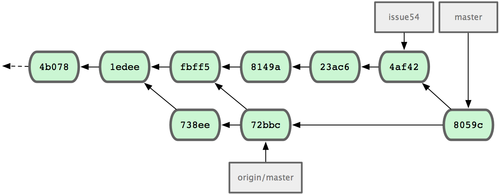
Auto-merging lib/simplegit.rb

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）：

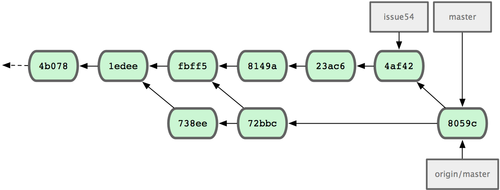
$ git push origin master

...

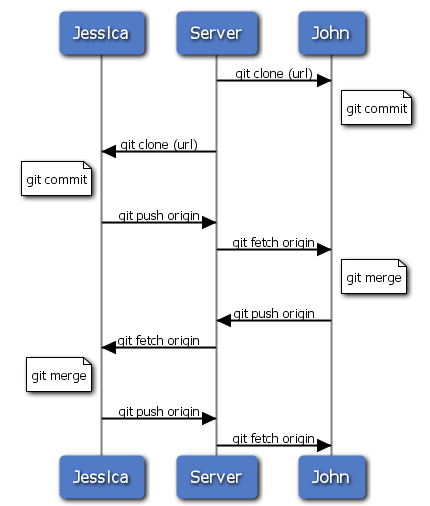
To jessica@githost:simplegit.git

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(-)

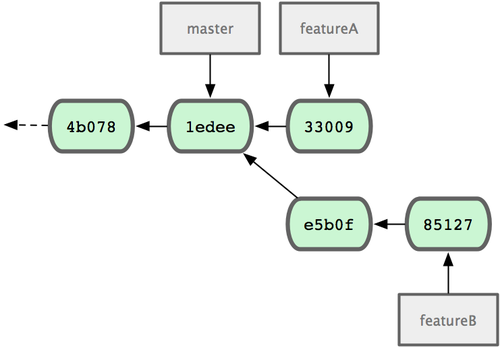
$ vim lib/simplegit.rb

$ 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 774b3ed] small tweak

1 files changed, 1 insertions(+), 1 deletions(-)

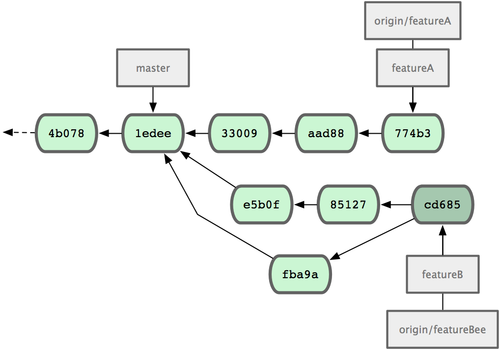
$ git push origin featureA

...

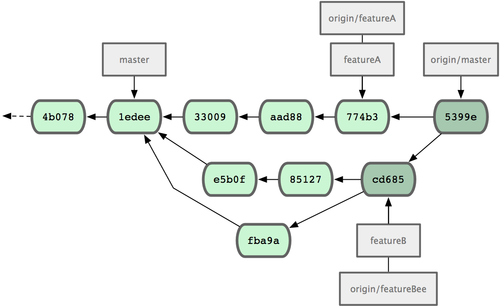
To jessica@githost:simplegit.git

3300904..774b3ed featureA -> featureA

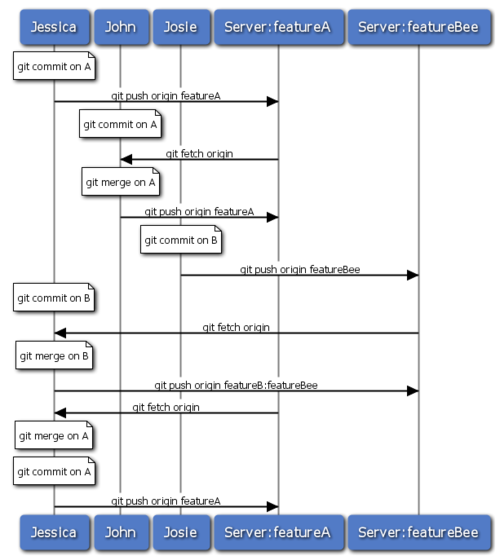
现在的 Jessica 提交历史如图 5-13 所示：

  
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)

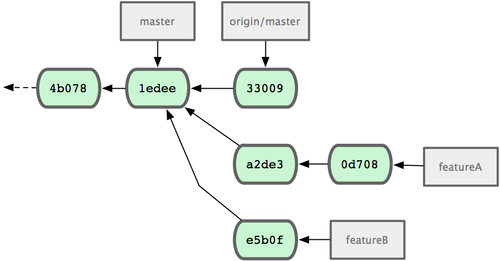
$ git commit

$ 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

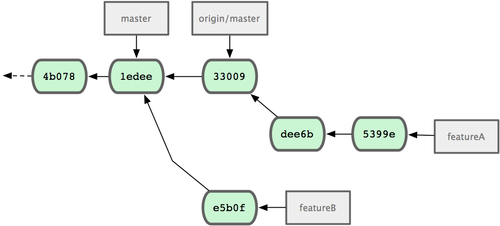
假设项目管理员接纳了许多别人提交的补丁后，准备要采纳你提交的第一个分支，却发现因为代码基准不一致，合并工作无法正确干净地完成。这就需要你再次衍合到最新的 origin/master，解决相关冲突，然后重新提交你的修改：

$ 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

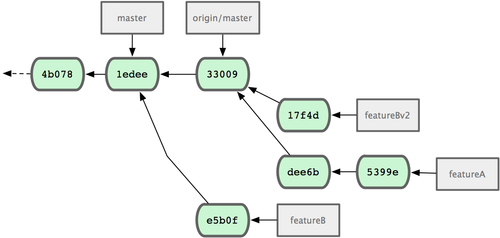
$ (change implementation)

$ 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。）：

$ cat \*.patch |git imap-send

Resolving imap.gmail.com... ok

Connecting to [74.125.142.109]:993... ok

Logging in...

sending 2 messages

100% (2/2) done

然后，你应该去你到草稿箱去更改你要发送的补丁的收件人信息，以及需要抄送的人，然后发送它。

您也可以通过SMTP服务器发送补丁。和上面一样，你可以通过git config命令单独设置每个参数，也可以在你的~/.gitconfig文件中的sendemail节点手动添加它们。

[sendemail]

smtpencryption = tls

smtpserver = smtp.gmail.com

smtpuser = user@gmail.com

smtpserverport = 587

配置完成后，您可以使用git send-email来发送你的补丁：

$ 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

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。