

Chapter 5. Distributed Git

- - Distributed Workflows
 - Centralized Workflow
 - Integration-Manager Workflow
 - Dictator and Lieutenants Workflow
 - Workflows Summary
 - Contributing to a Project
 - Commit Guidelines
 - Private Small Team
 - Forked Public Project
 - Public Project over E-Mail
 - Summary
 - Maintaining a Project
 - Working in Topic Branches
 - Applying Patches from E-mail
 - Applying a Patch with apply
 - Applying a Patch with am
 - Checking Out Remote Branches
 - Determining What Is Introduced
 - Integrating Contributed Work
 - Merging Workflows
 - Large-Merging Workflows
 - Rerere
 - Tagging Your Releases
 - Generating a Build Number
 - Preparing a Release
 - The Shortlog
 - Summary

Distributed Workflows

Centralized Workflow

- In centralized systems, there is generally a single collaboration model-the centralized workflow.
- One central hub, or repository, can accept code, and everyone synchronizes their work to it.
- A number of developers are nodes - consumers of that hub - and synchronize to that one place.

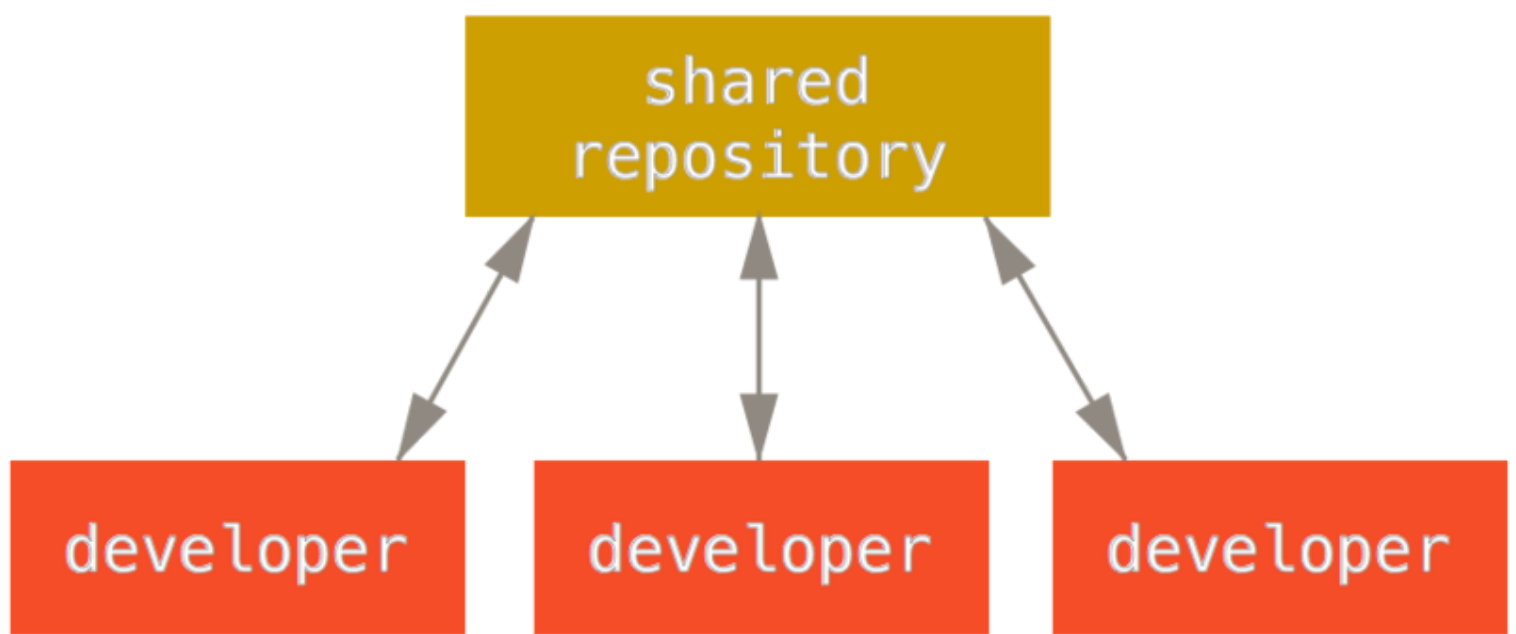


Figure 5-1. Centralized workflow.

- This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems.
- The second developer must merge in the first ones work before pushing changes up, so as not to overwrite the first developers changes.
- This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

Integration-Manager Workflow

- Its possible to have a workflow where each developer has write access to their own public repository and read access to everyone elses.
- The process works as follows:
 1. The project maintainer pushes to their public repository.
 2. A contributor clones that repository and makes changes.
 3. The contributor pushes to their own public copy.
 4. The contributor sends the maintainer an e-mail asking them to pull changes.
 5. The maintainer adds the contributors repo as a remote and merges locally.
 6. The maintainer pushes merged changes to the main repository.

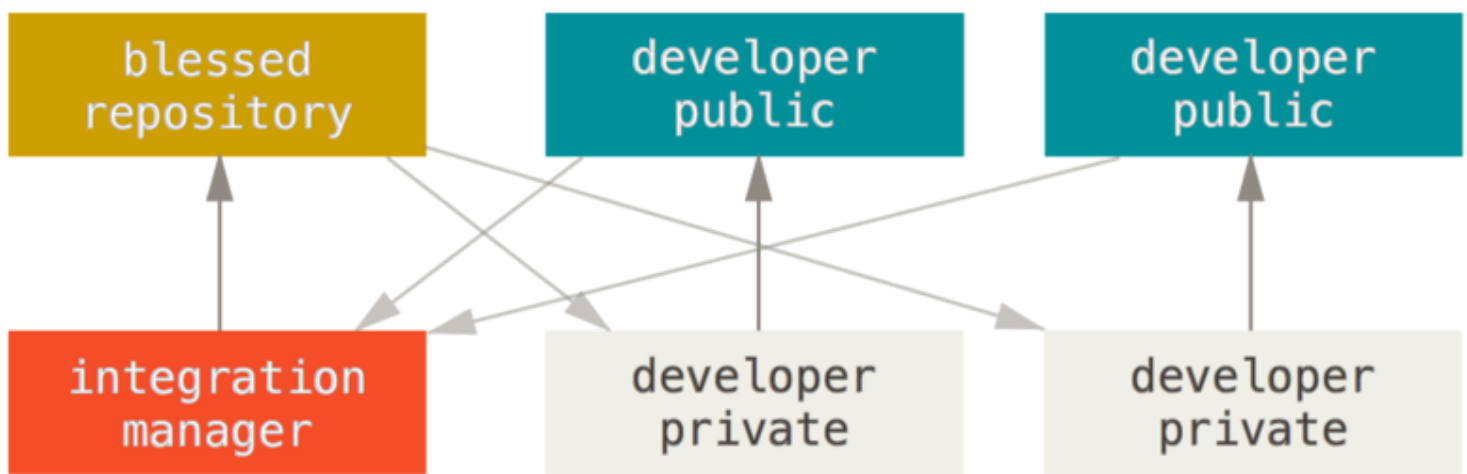


Figure 5-2. Integration-manager workflow.

- One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time.
- Contributors don't have to wait for the project to incorporate their changes - each party can work at their own pace.

Dictator and Lieutenants Workflow

- This is a variant of a multiple-repository workflow.
- Various integration managers are in charge of certain parts of the repository; they're called lieutenants.
- All the lieutenants have one integration manager known as the benevolent dictator.
- The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull.
- The process works like this:
 1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the dictator.
 2. Lieutenants merge the developers topic branches into their `master` branch.
 3. The dictator merges the lieutenants `master` branches into the dictators `master` branch.
 4. The dictator pushes their `master` to the reference repository so the other developers can rebase on it.

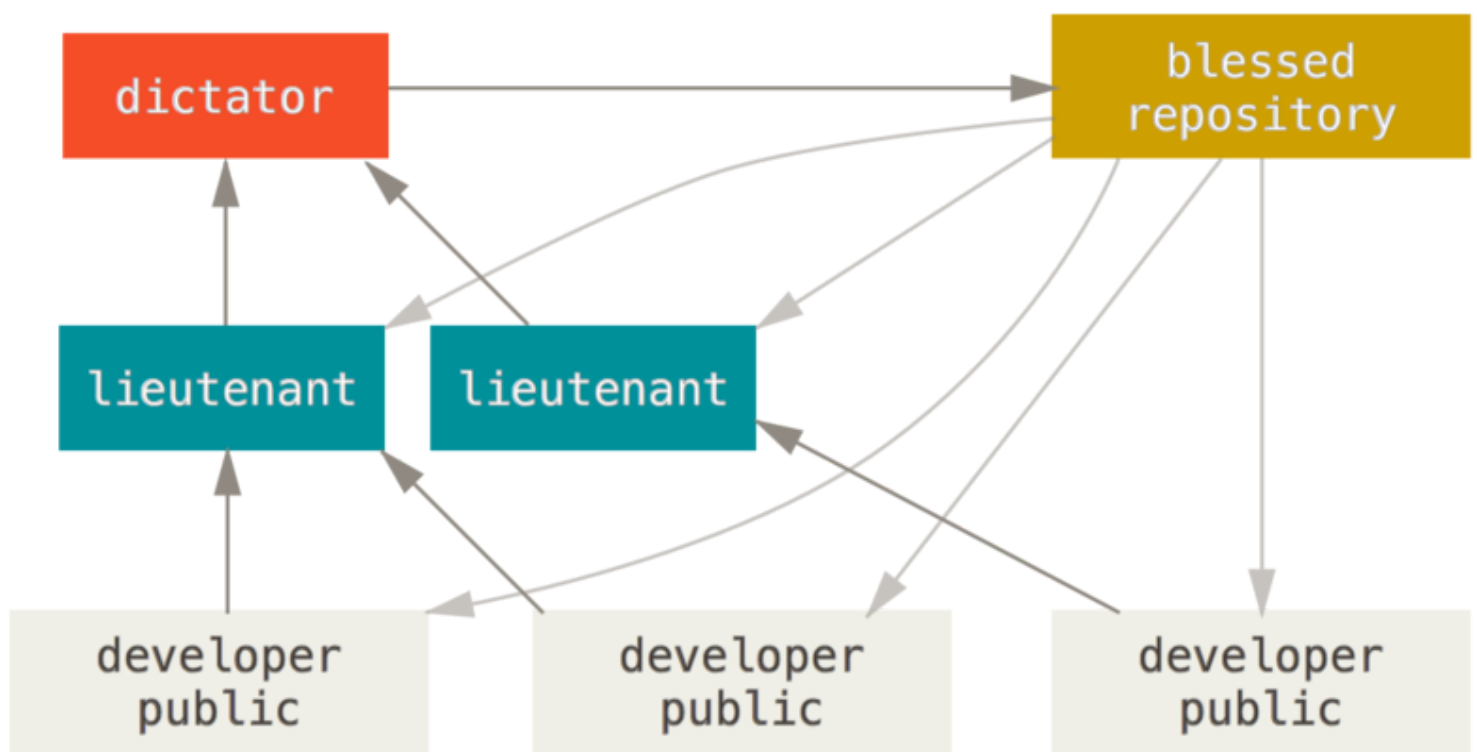


Figure 5-3. Benevolent dictator workflow.

- This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments.
- It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

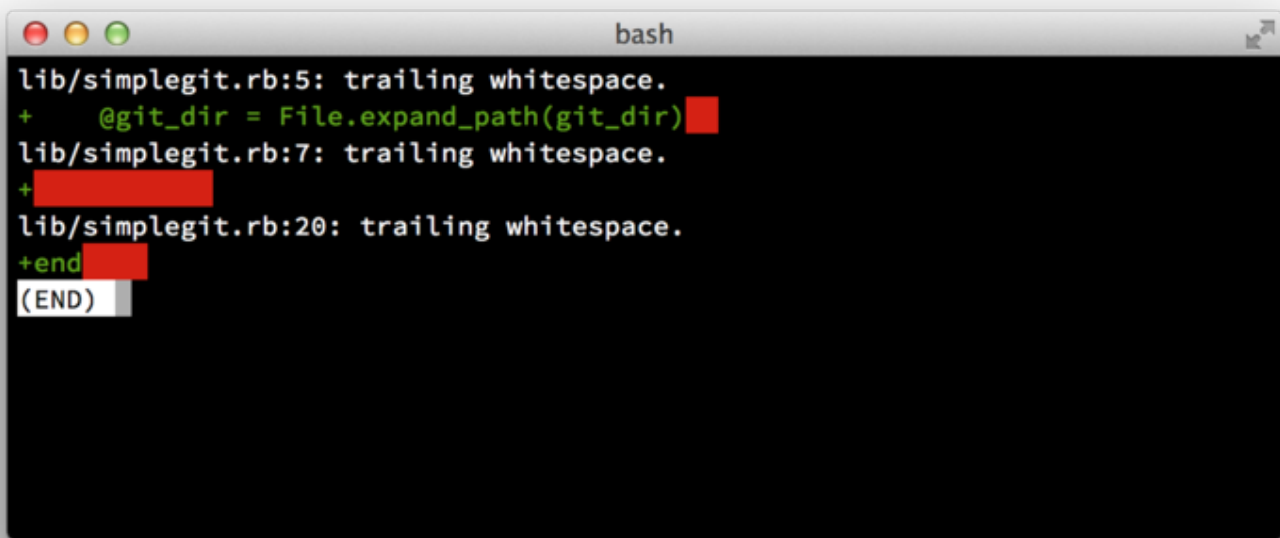
Workflows Summary

Contributing to a Project

- The first variable is active contributor count - how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects.
- The next variable is the workflow in use for the project.
- Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

Commit Guidelines

- you can read it in the Git source code in the `Documentation/SubmittingPatches` file.
- First, you don't want to submit any whitespace errors.
- Git provides an easy way to check for this - before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.

A terminal window titled 'bash' showing the output of the 'git diff --check' command. The output consists of four lines of error messages, each indicating a 'trailing whitespace' issue in a specific file: 'lib/simplegit.rb:5: trailing whitespace.', 'lib/simplegit.rb:7: trailing whitespace.', and 'lib/simplegit.rb:20: trailing whitespace.'. Each message is preceded by a '+' sign. The fourth line is '+end'. The prompt '(END)' is visible at the bottom left of the terminal window.

```
bash
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+ 
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Figure 5-4. Output of `git diff --check`.

- If you run that command before committing, you can tell if you're about to commit whitespace issues that may annoy other developers.
- Next, try to make each commit a logically separate changeset.
- The last thing to keep in mind is the commit message.
- Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier.
- As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation.

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

- If all your commit messages look like this, things will be a lot easier for you and the developers you work with.
- The Git project has well-formatted commit messages - try running `git log --no-merges` there to see what a nicely formatted project-commit history looks like.

- Private, in this context, means closed-source - not accessible to the outside world. You and the other developers all have push access to the repository.
- The first developer, John, clones the repository, makes a change, and commits locally.

```
# John's Machine
$ git clone john@github: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(-)
```

- The second developer, Jessica, does the same thing - clones the repository and commits a change:

```
# Jessica's Machine
$ git clone jessica@github: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(-)
```

- Now, Jessica pushes her work up to the server:

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

- John tries to push his change up, too:

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

- John isn't allowed to push because Jessica has pushed in the meantime.
- This is especially important to understand if you're used to Subversion, because you'll notice that the two developers didn't edit the same file.
- Although Subversion automatically does such a merge on the server if different files are edited, in Git you must merge the commits locally.
- John has to fetch Jessica's changes and merge them in before he will be allowed to push:

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

- At this point, Johns local repository looks something like this:

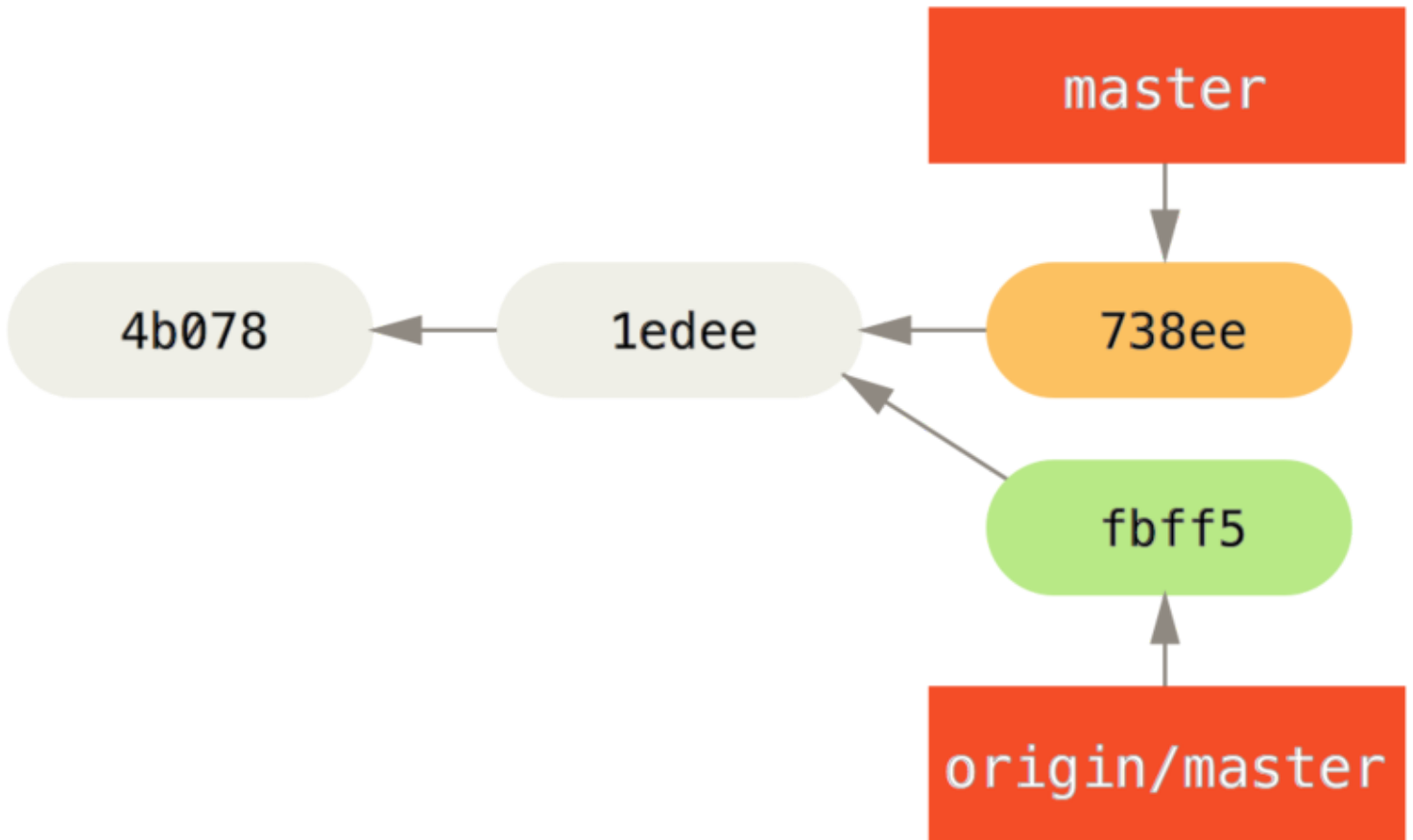


Figure 5-5. Johns divergent history.

- John has a reference to the changes Jessica pushed up, but he has to merge them into his own work before he is allowed to push:

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

The merge goes smoothly - Johns commit history now looks like this:

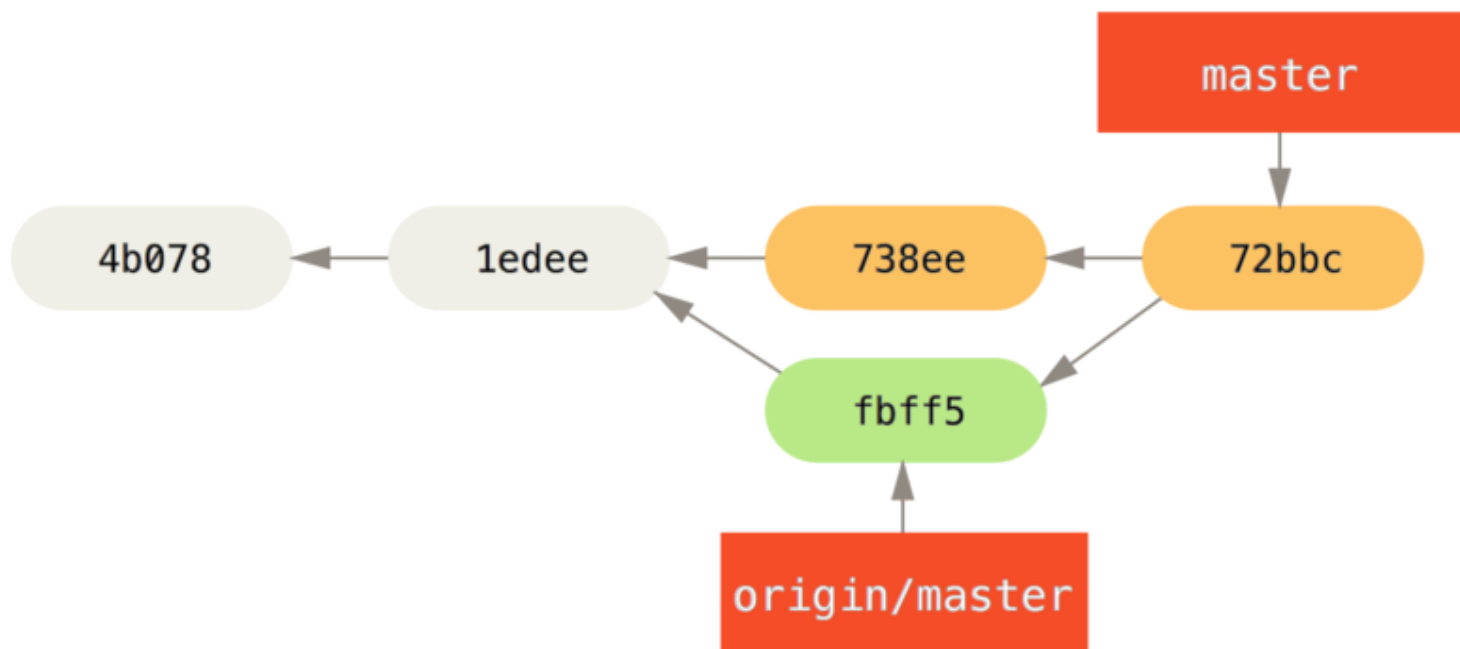


Figure 5-6. Johns repository after merging `origin/master`.

- Now, John can test his code to make sure it still works properly, and then he can push his new merged work up to the server:

```

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

- Finally, Johns commit history looks like this:

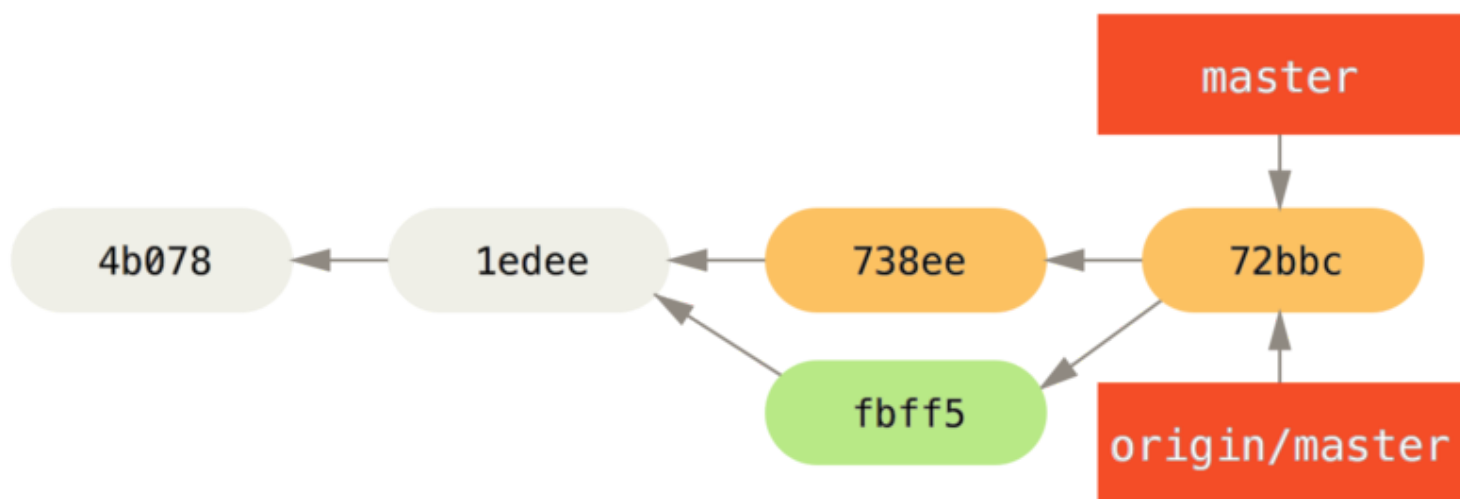


Figure 5-7. Johns history after pushing to the `origin` server.

- In the meantime, Jessica has been working on a topic branch.
- Shes created a topic branch called `issue54` and done three commits on that branch.
- She hasnt fetched Johns changes yet, so her commit history looks like this:

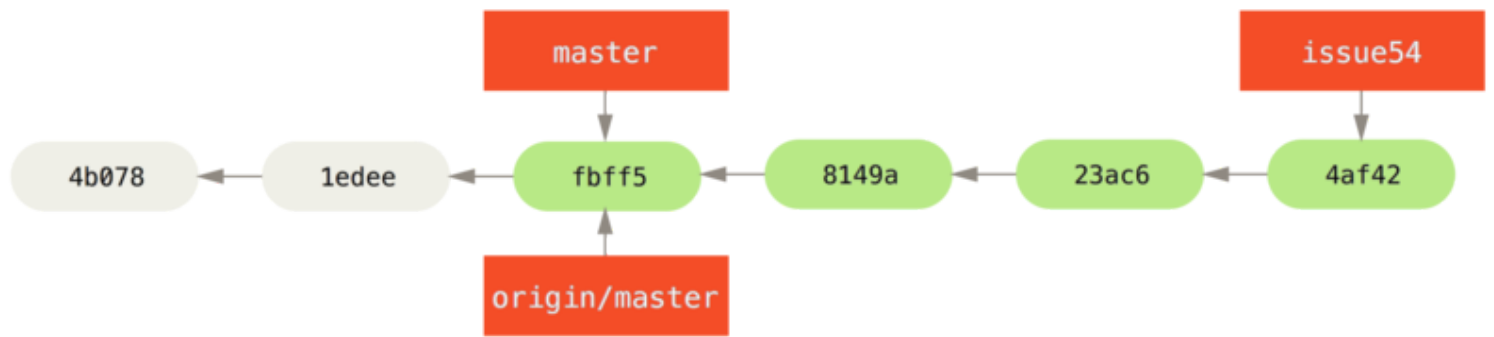


Figure 5-8. Jessica's topic branch.

- Jessica wants to sync up with John, so she fetches:

```

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

- That pulls down the work John has pushed up in the meantime. Jessica's history now looks like this:

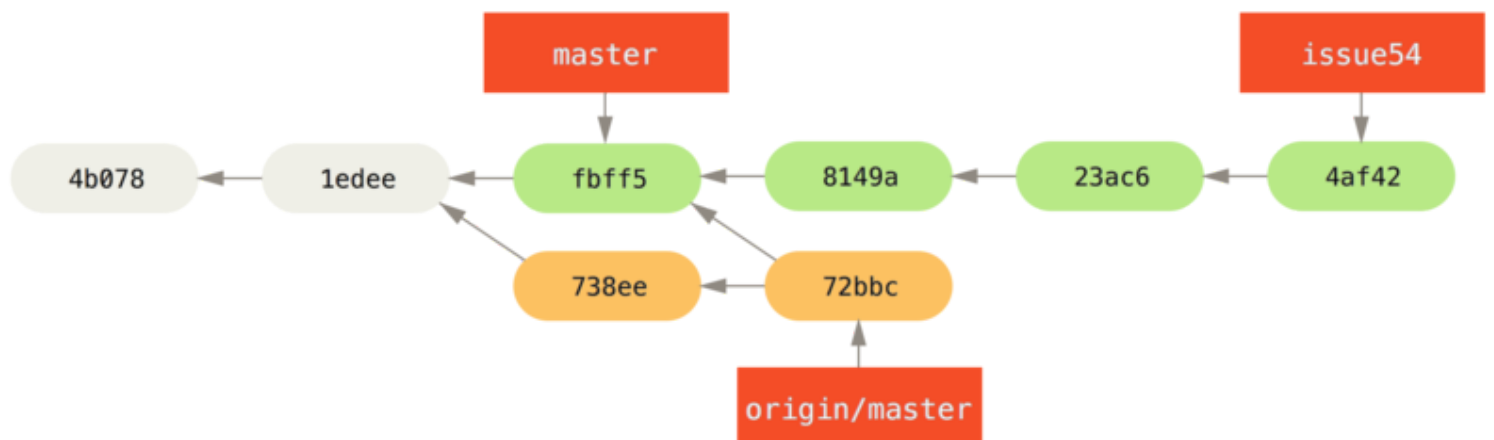


Figure 5-9. Jessica's history after fetching John's changes.

- Jessica thinks her topic branch is ready, but she wants to know what she has to merge into her work so that she can push. She runs `git log` to find out:

```

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

    removed invalid default value
  
```

- The `issue54..origin/master` syntax is a log filter that asks Git to only show the list of commits that are on the latter branch (in this case `origin/master`) that are not on the first branch.
- For now, we can see from the output that there is a single commit that John has made that Jessica has not merged in.

- If she merges `origin/master`, that is the single commit that will modify her local work.
- Now, Jessica can merge her topic work into her master branch, merge Johns work (`origin/master`) into her `master` branch, and then push back to the server again.
- First, she switches back to her master branch to integrate all this work:

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

- She can merge either `origin/master` or `issue54` first - they're both upstream, so the order doesn't matter.
- The end snapshot should be identical no matter which order she chooses; only the history will be slightly different.
- She chooses to merge in `issue54` first:

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

- No problems occur; as you can see it was a simple fast-forward. Now Jessica merges in Johns work (`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(-)
```

- Everything merges cleanly, and Jessicas history looks like this:

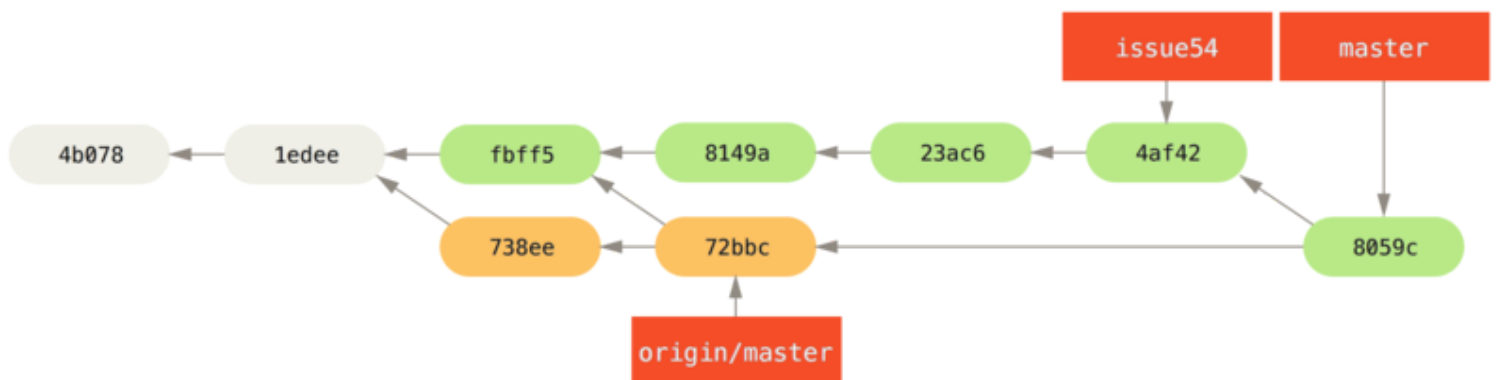


Figure 5-10. Jessicas history after merging Johns changes.

- Now `origin/master` is reachable from Jessicas `master` branch, so she should be able to successfully push (assuming John hasn't pushed again in the meantime):

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

- Each developer has committed a few times and merged each others work successfully.

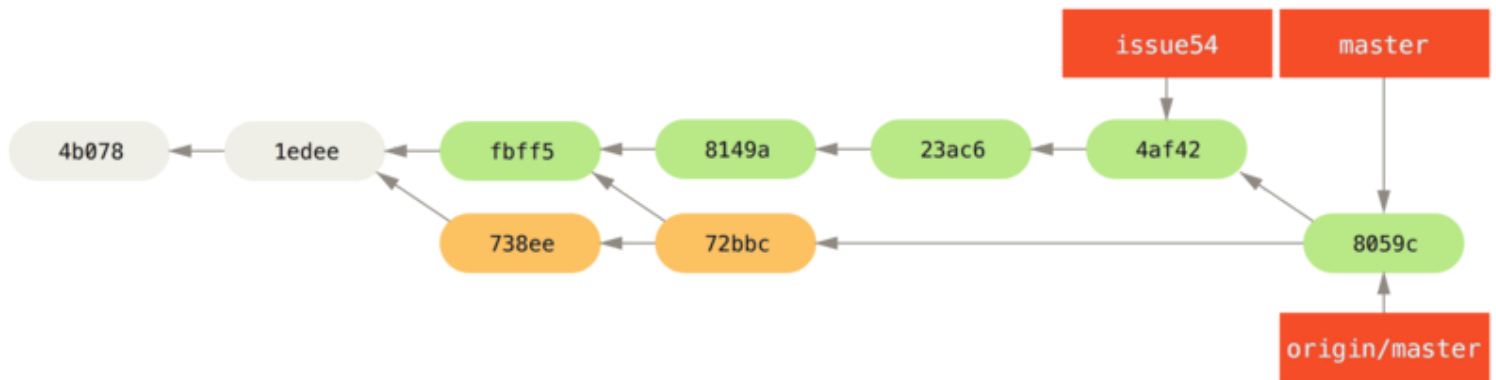


Figure 5-11. Jessicas history after pushing all changes back to th.

When you want to share that work, you merge it into your own master branch, then fetch and merge `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like this:

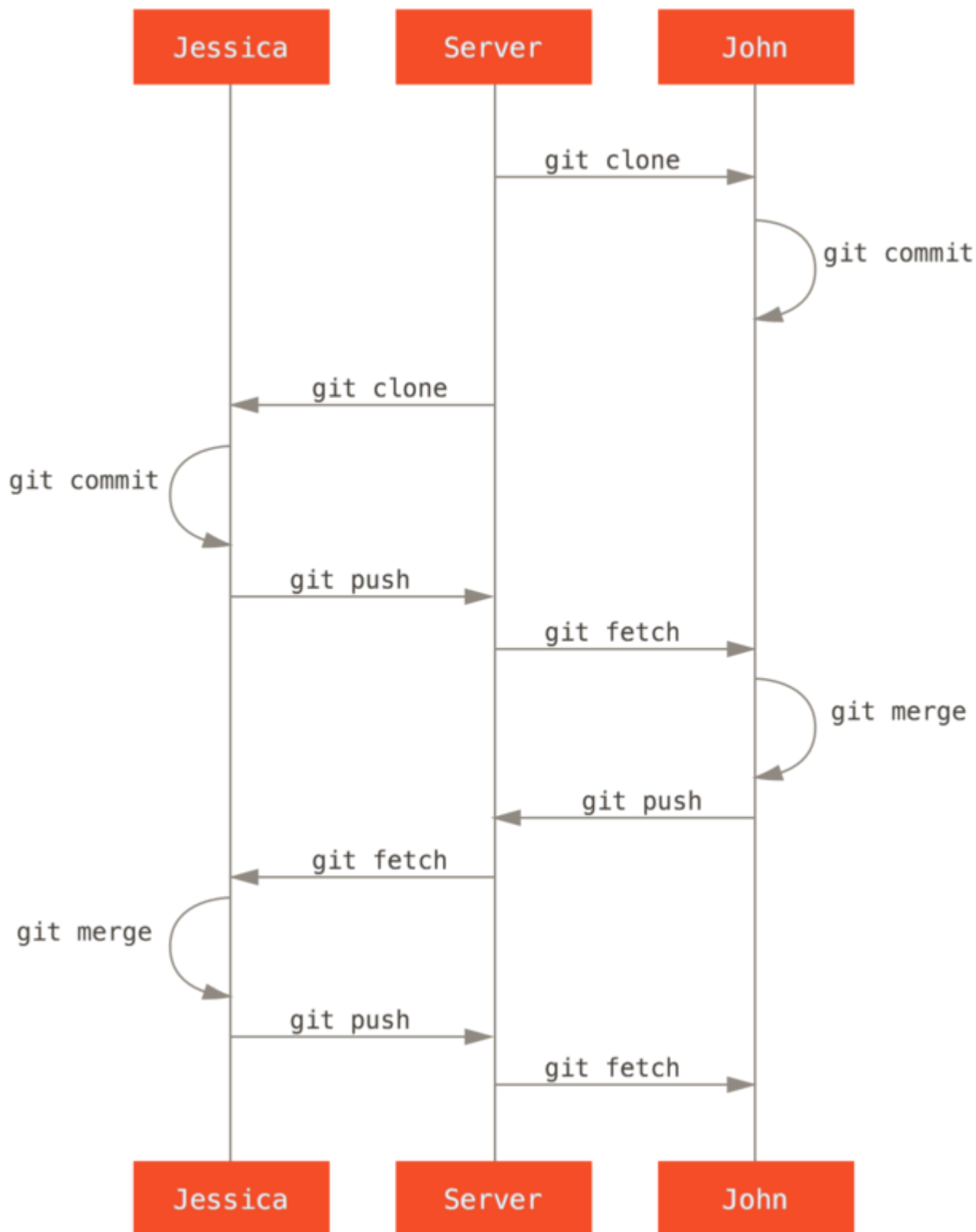


Figure 5-12. General sequence of events for a simple multiple-developer.

Git workflow.

Private Managed Team

In this case, the company is using a type of integration-manager workflow where the work of the individual groups is integrated only by certain engineers, and the `master` branch of the main repo can be updated only by those engineers.

Assuming she already has her repository cloned, she decides to work on `featureA` first. She creates a new branch for the feature and does some work on it there:

```
git # 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(-) featureA
```

Jessica doesn't have push access to the `master` branch - only the integrators do - so she has to push to another branch in order to collaborate with John:

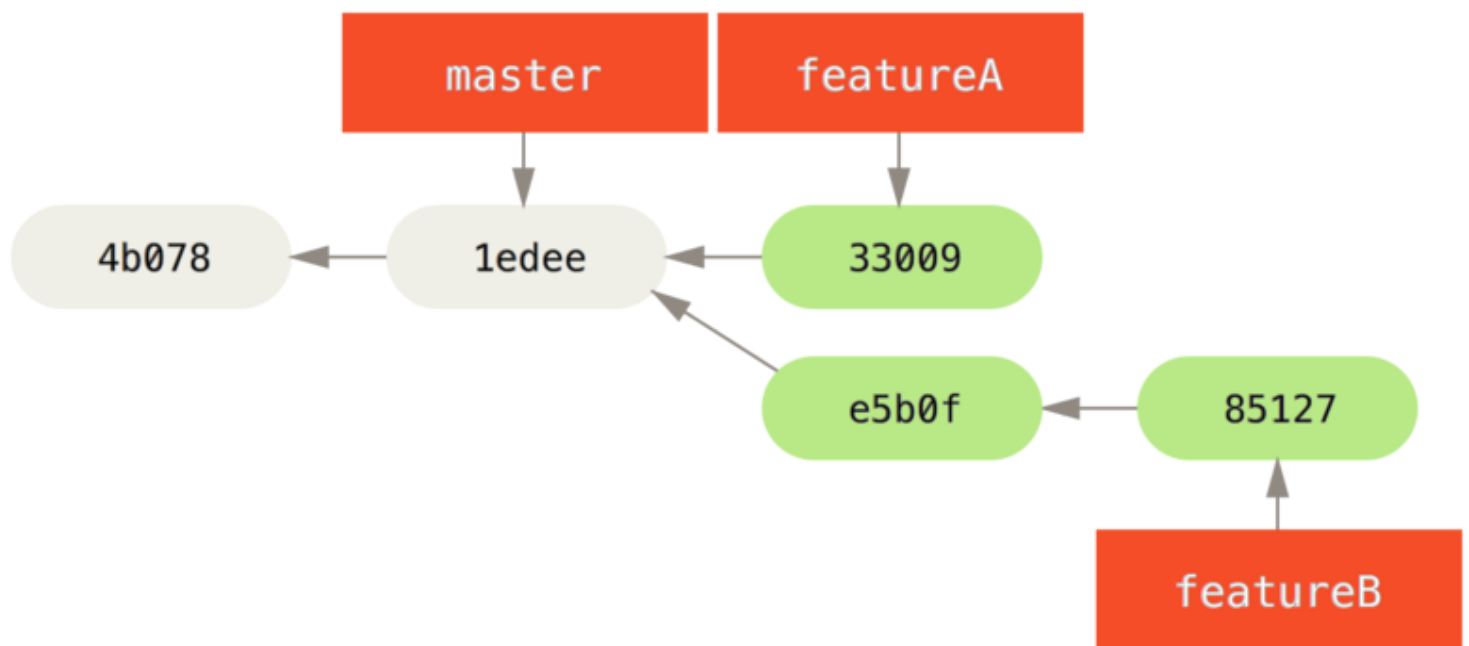
```
`git
$ git push -u origin featureA
...
```

To `jessica@githost:simplegit.git`

```
* Jessica e-mails John to tell him that she's pushed some work into a branch named
`featureA` and he can look at it now. * While she waits for feedback from John,
Jessica decides to start working on `featureB` with Josie. * To begin, she starts a
new feature branch, basing it off the server's `master` branch:
```

```
* Now, Jessica makes a couple of commits on the `featureB` branch:
```

Jessica's repository looks like this:



** Figure 5-13. Jessicas initial commit history.*

- Shes ready to push up her work, but gets an e-mail from Josie that a branch with some initial work on it was already pushed to the server as `featureBee`.
- Jessica first needs to merge those changes in with her own before she can push to the server.
- She can then fetch Josies changes down with `git fetch`:

```

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

```

- Jessica can now merge this into the work she did with `git merge`:

```

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

```

- There is a bit of a problem - she needs to push the merged work in her `featureB` branch to the `featureBee` branch on the server.
- She can do so by specifying the local branch followed by a colon (:) followed by the remote branch to the `git push` command:

```

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

```

- This is called a *refspec*.

- Next, John e-mails Jessica to say hes pushed some changes to the `featureA` branch and asks her to verify them.
- She runs a `git fetch` to pull down those changes:

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

- Then, she can see what has been changed with `git log`:

```
$ git log featureA..origin/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

- Finally, she merges Johns work into her own `featureA` branch:

```
$ 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 wants to tweak something, so she commits again and then pushes this back up to the server:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@github:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

- Jessicas commit history now looks something like this:

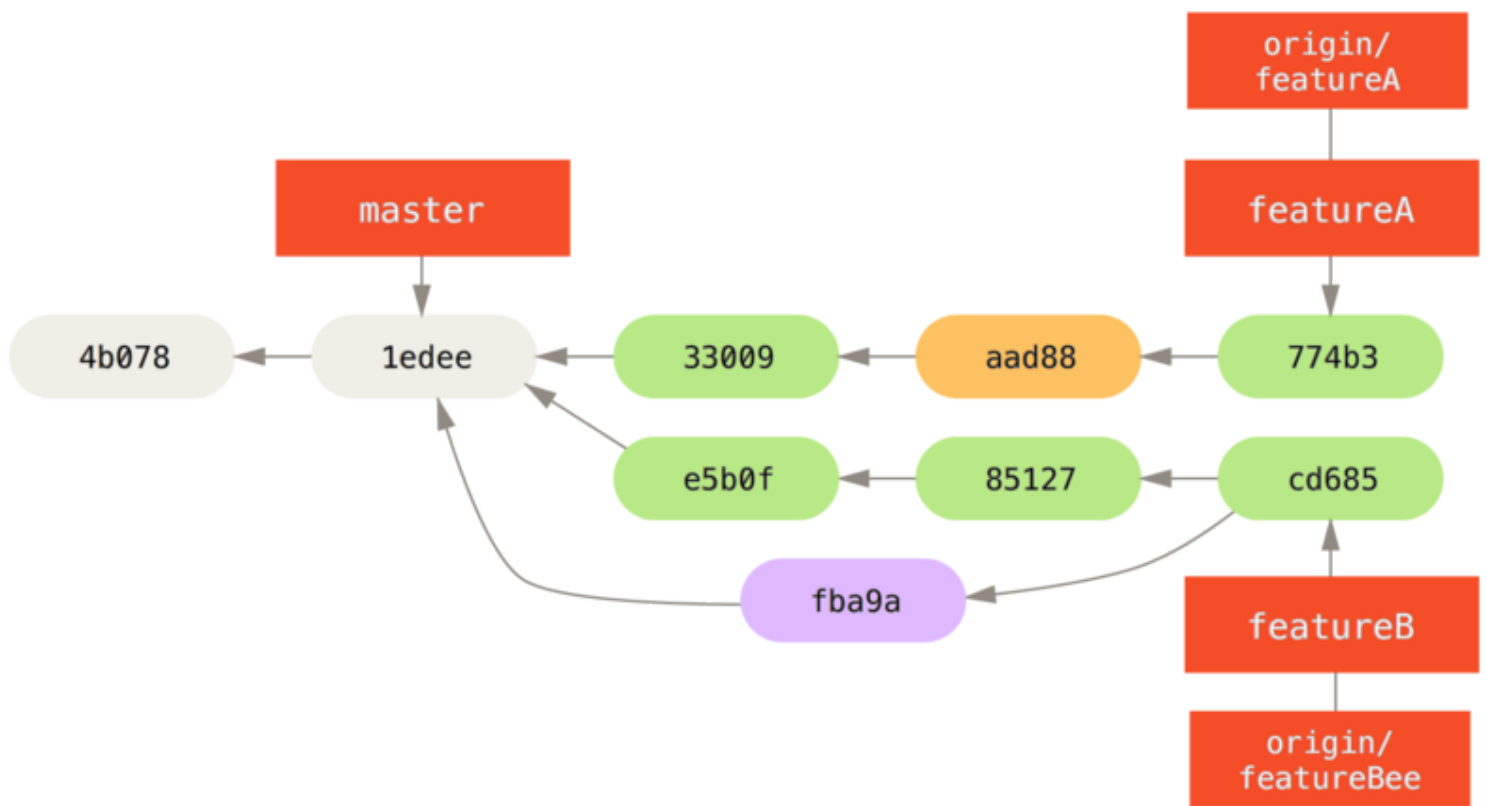


Figure 5-14. Jessicas history after committing on a feature branch.

- Jessica, Josie, and John inform the integrators that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline.
- After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:

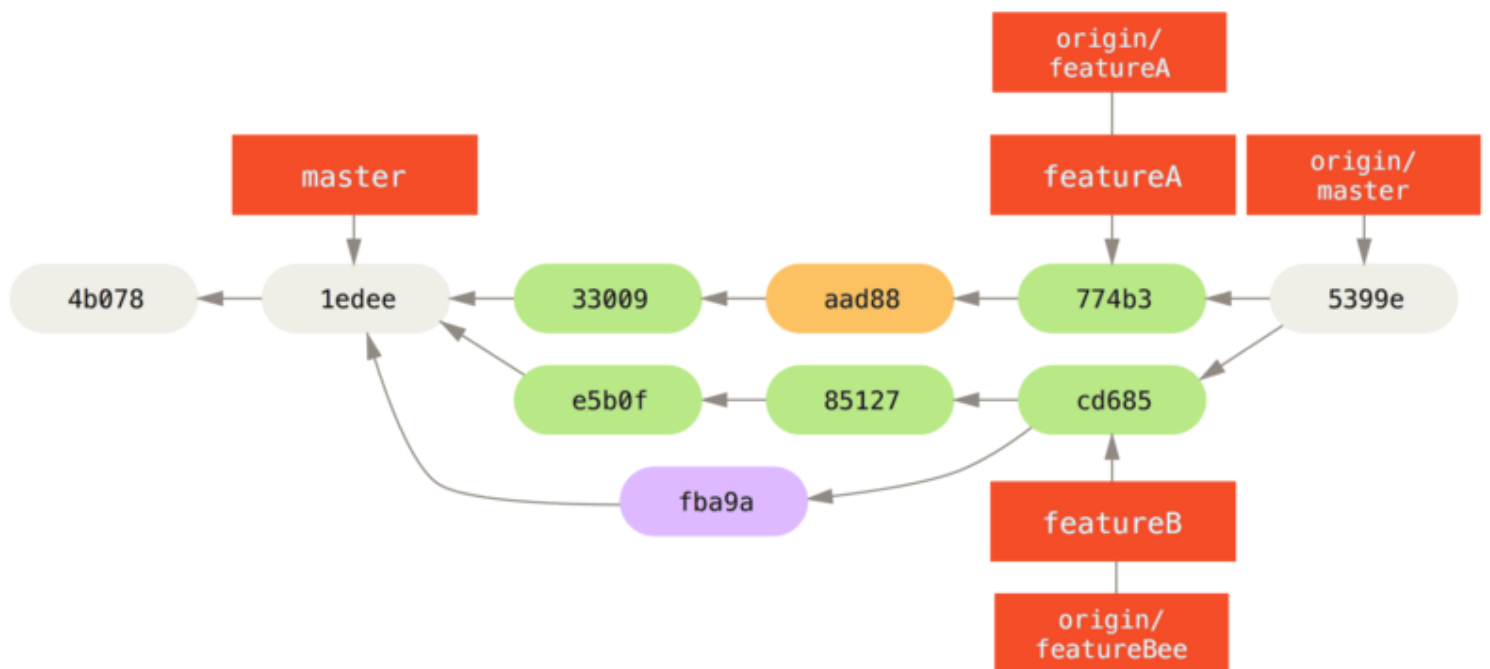


Figure 5-15. Jessicas history after merging both her topic branches.

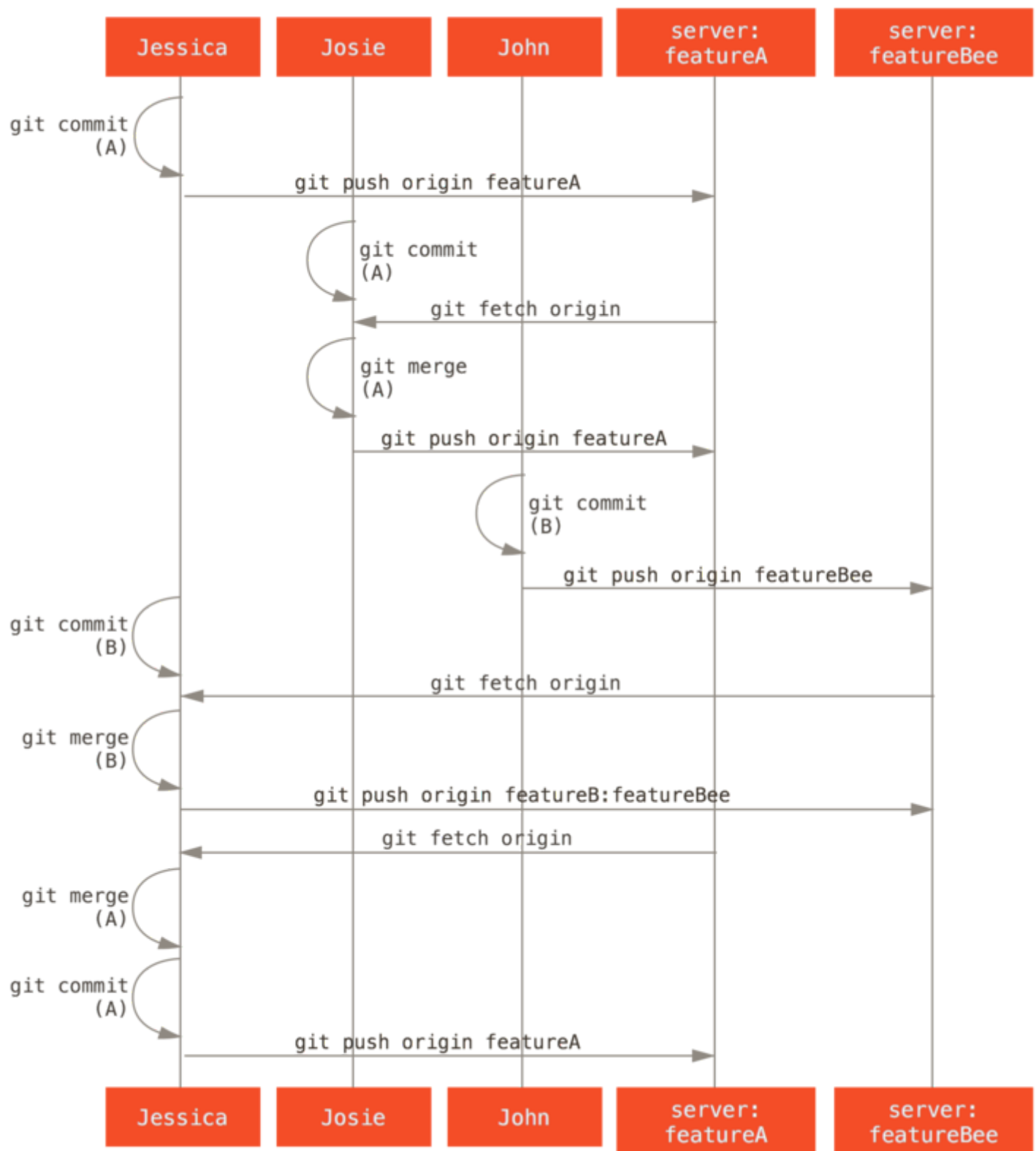


Figure 5-16. Basic sequence of this managed-team workflow.

Forked Public Project

- First, you'll probably want to clone the main repository, create a topic branch for the patch or patch series you're planning to contribute, and do your work there.
- The sequence looks basically like this:

```

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

```

```
$ git commit
# (work)
$ git commit
```

- You may want to use `rebase -i` to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review - see Rewriting History for more information about interactive rebasing.
- When your branch work is finished and youre ready to contribute it back to the maintainers, go to the original project page and click the Fork button, creating your own writable fork of the project.
- You then need to add in this new repository URL as a second remote, in this case named `myfork`:

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

- Then you need to push your work up to it.
- Its easiest to push the topic branch youre working on up to your repository, rather than merging into your master branch and pushing that up.
- The reason is that if the work isnt accepted or is cherry picked, you dont have to rewind your master branch.
- If the maintainers merge, rebase, or cherry-pick your work, youll eventually get it back via pulling from their repository anyhow:

```
$ git push -u myfork featureA
```

- When your work has been pushed up to your fork, you need to notify the maintainer.
- This is often called a pull request, and you can either generate it via the website - GitHub has its own Pull Request mechanism that well go over in.
- Can run the `git request-pull` command and e-mail the output to the project maintainer manually.
- The `request-pull` command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes youre asking to be pulled in.
- For instance, if Jessica wants to send John a pull request, and shes done two commits on the topic branch she just pushed up, she can run this:

```
$ 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(-)
```

- The output can be sent to the maintainer-it tells them where the work was branched from, summarizes the commits, and tells where to pull this work from.
- On a project for which youre not the maintainer, its generally easier to have a branch like `master` always track `origin/master` and to do your work in topic branches that you can easily discard if theyre rejected.

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

- Now, each of your topics is contained within a silo - similar to a patch queue - that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:

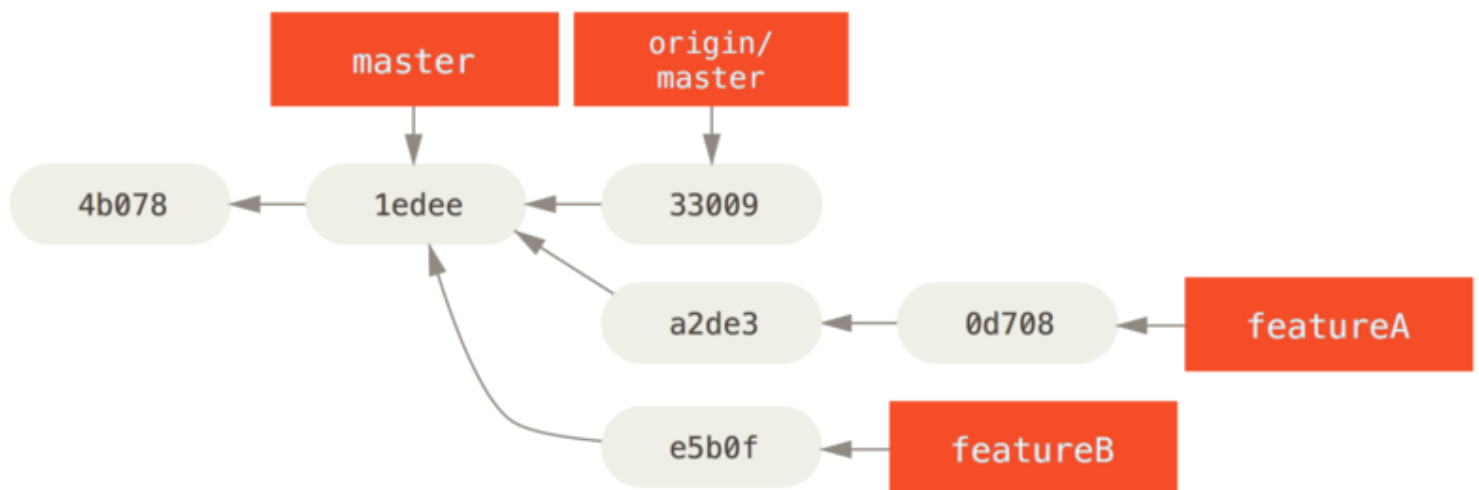


Figure 5-17. Initial commit history with `featureB` work.

- Lets say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges.
- In this case, you can try to rebase that branch on top of `origin/master`, resolve the conflicts for the maintainer, and then resubmit your changes:

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

This rewrites your history to now look like Figure 5-18.

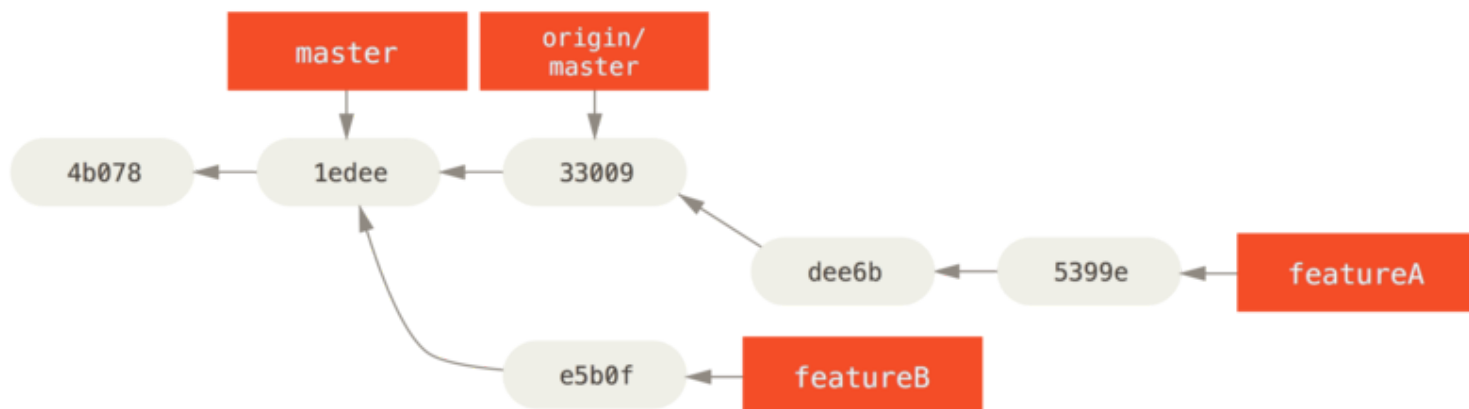


Figure 5-18. Commit history after `featureA` work.

- Because you rebased the branch, you have to specify the `-f` to your push command in order to be able to replace the `featureA` branch on the server with a commit that isn't a descendant of it.
- An alternative would be to push this new work to a different branch on the server (perhaps called `featureAv2`).

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

- The `--squash` option takes all the work on the merged branch and squashes it into one changeset producing the repository state as if a real merge happened, without actually making a merge commit.
- This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit.

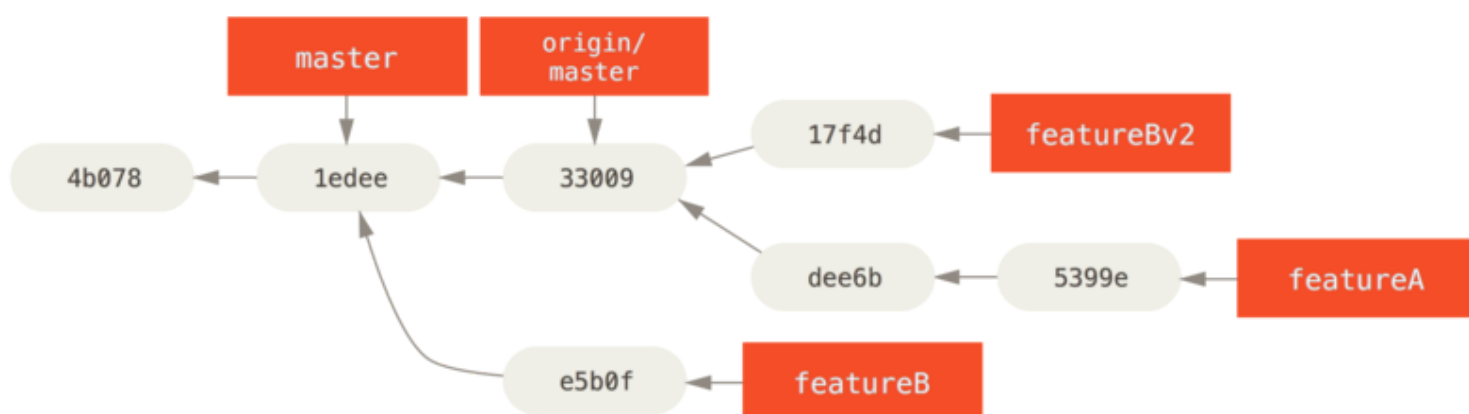


Figure 5-19. Commit history after `featureBv2` work.

Public Project over E-Mail

- The workflow is similar to the previous use case - you create topic branches for each patch series you work on.
- The difference is how you submit them to the project.
- Instead of forking the project and pushing to your own writable version, you generate e-mail versions of each commit series and e-mail them to the developer mailing list:

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

- You use `git format-patch` to generate the mbox-formatted files that you can e-mail to the list - it turns each commit into an e-mail message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body.

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

- The `format-patch` command prints out the names of the patch files it creates.
- The `-M` switch tells Git to look for renames.
- The files end up looking like this:

```
$ 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')
--
2.1.0
```

- If you add text between the `\---` line and the beginning of the patch (the `diff --git` line), then developers can read it; but applying the patch excludes it.
- Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you.

- First, you need to set up the imap section in your `~/.gitconfig` file.
- You can set each value separately with a series of `git config` commands, or you can add them manually, but in the end your config file should look something like this:

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

- If your IMAP server doesn't use SSL, the last two lines probably aren't necessary, and the host value will be `imap://` instead of `imaps://`.
- When that is set up, you can use `git imap-send` to place the patch series in the Drafts folder of the specified IMAP server:

```
$ 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
```

- At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you're sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.
- You can also send the patches through an SMTP server.
- As before, you can set each value separately with a series of `git config` commands, or you can add them manually in the sendemail section in your `~/.gitconfig` file:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

- After this is done, you can use `git send-email` to send your patches:

```
$ 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
```

- Then, Git spits out a bunch of log information looking something like this for each patch you're sending:

```
(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
```

Summary

- This section has covered a number of common workflows for dealing with several very different types of Git projects you're likely to encounter, and introduced a couple of new tools to help you manage this process.
- Next, you'll see how to work the other side of the coin: maintaining a Git project.
- You'll learn how to be a benevolent dictator or integration manager.

Maintaining a Project

Working in Topic Branches

- When you're thinking of integrating new work, it's generally a good idea to try it out in a topic branch - a temporary branch specifically made to try out that new work.
- As you'll remember, you can create the branch based off your master branch like this:

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

- Or, if you want to also switch to it immediately, you can use the `checkout -b` option:

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

- Now you're ready to add your contributed work into this topic branch and determine if you want to merge it into your longer-term branches.

Applying Patches from E-mail

- There are two ways to apply an e-mailed patch: with `git apply` or with `git am`.

Applying a Patch with apply

- If you received the patch from someone who generated it with the `git diff` or a Unix `diff` command (which is not recommended; see the next section), you can apply it with the `git apply` command.
- Assuming you saved the patch at `/tmp/patch-ruby-client.patch`, you can apply the patch like this:

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

- This modifies the files in your working directory.
- Its almost identical to running a `patch -p1` command to apply the patch, although its more paranoid and accepts fewer fuzzy matches than patch.
- It also handles file adds, deletes, and renames if theyre described in the `git diff` format, which `patch` wont do.
- Finally, `git apply` is an apply all or abort all model where either everything is applied or nothing is, whereas `patch` can partially apply patchfiles, leaving your working directory in a weird state.
- `git apply` is overall much more conservative than `patch`.
- It wont create a commit for you - after running it, you must stage and commit the changes introduced manually.
- You can also use `git apply` to see if a patch applies cleanly before you try actually applying it - you can run `git apply --check` with the patch:

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

- If there is no output, then the patch should apply cleanly.
- This command also exits with a non-zero status if the check fails, so you can use it in scripts if you want.

Applying a Patch with `am`

- If the contributor is a Git user and was good enough to use the `format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message for you.
- If you can, encourage your contributors to use `format-patch` instead of `diff` to generate patches for you.
- You should only have to use `git apply` for legacy patches and things like that.
- To apply a patch generated by `format-patch`, you use `git am`.
- Technically, `git am` is built to read an mbox file, which is a simple, plain-text format for storing one or more e-mail messages in one text file. It looks something like this:

```
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

- However, if someone uploaded a patch file generated via `format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

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

- You can see that it applied cleanly and automatically created the new commit for you.
- The author information is taken from the e-mails `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the e-mail.

For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ 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
```

- The `Commit` information indicates the person who applied the patch and the time it was applied.
- The `Author` information is the individual who originally created the patch and when it was originally created.

```
$ 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".
```

- This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation.
- You solve this issue much the same way - edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch:

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

- You can pass a `-3` option to it, which makes Git attempt a three-way merge.
- This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository.
- If you do have that commit - if the patch was based on a public commit - then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ 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.
```

```
$ 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
```

Checking Out Remote Branches

- For instance, if Jessica sends you an e-mail saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

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

- If she e-mails you again later with another branch containing another great feature, you can fetch and check out because you already have the remote setup.
- The other advantage of this approach is that you get the history of the commits as well.
- Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.
- If you aren't working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command.
- This does a one-time pull and doesn't save the URL as a remote reference:

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

Determining What Is Introduced

- You can exclude commits in the master branch by adding the `\|--not` option before the branch name.
- If your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ 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
```

- To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.
- To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results.
- You may think to run this:

```
$ git diff master
```

- This command gives you a diff, but it may be misleading.
- If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results.
- This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch.
- If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.
- Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

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

- However, that isn't convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax.

- In the context of the `diff` command, you can put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with another branch:

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

Integrating Contributed Work

Merging Workflows

- One simple workflow merges your work into your `master` branch.
- If we have a repository with work in two branches named `ruby_client` and `php_client` merge `ruby_client` first and then `php_client` next.

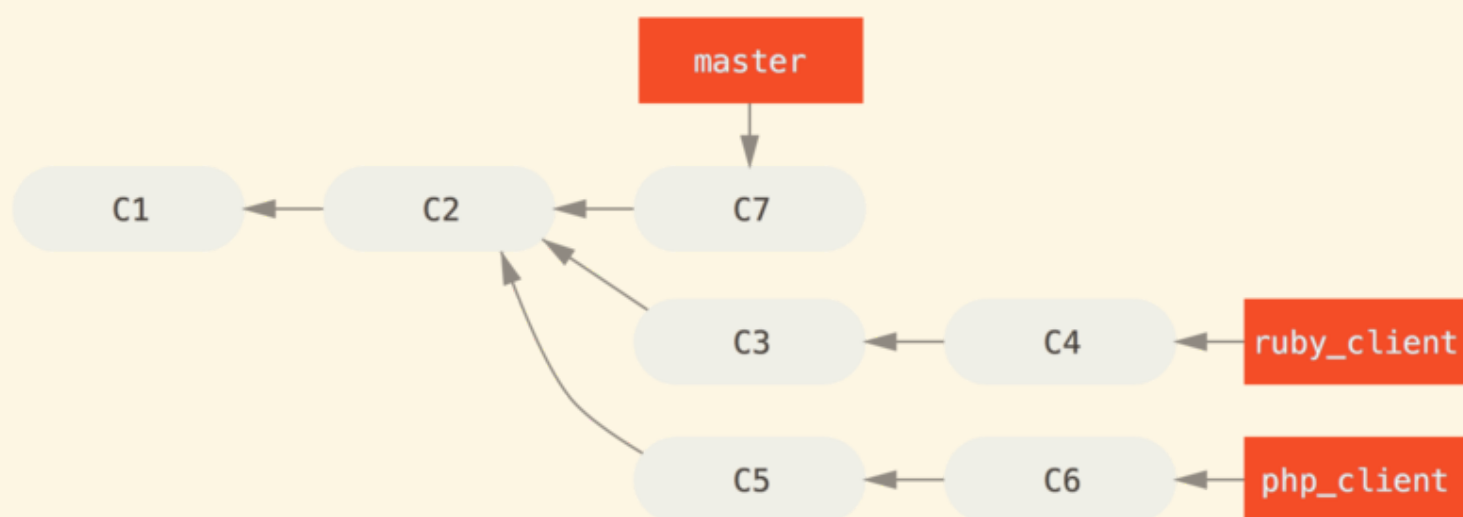


Figure 5-20. History with several topic branches.

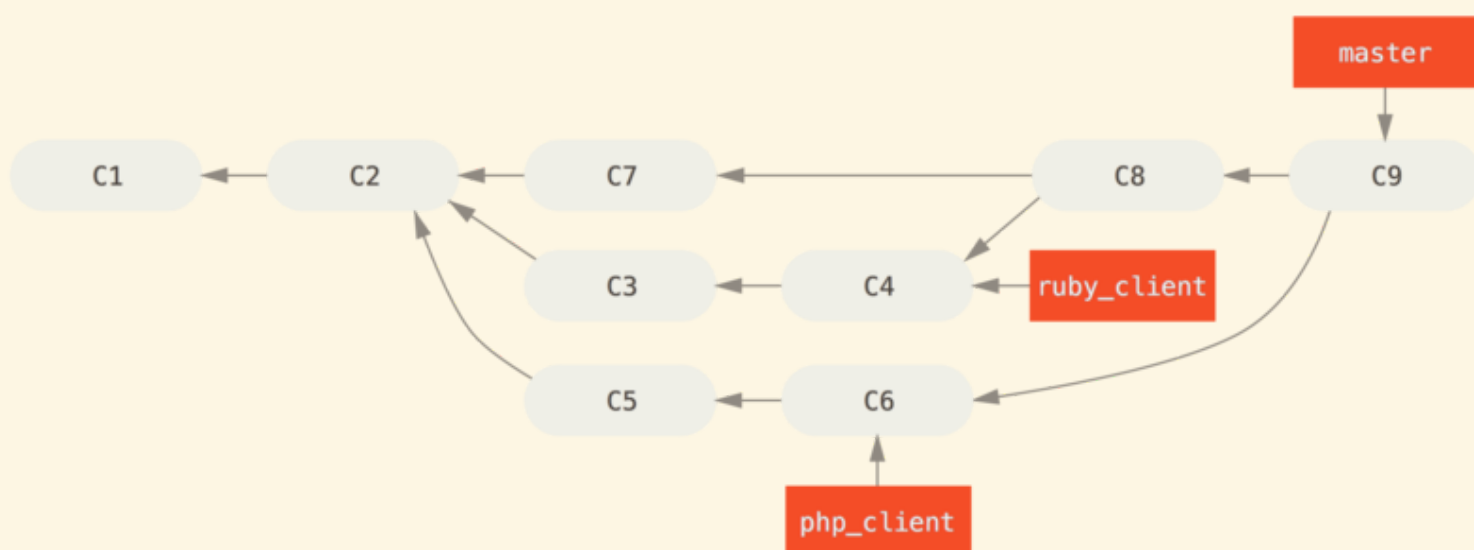


Figure 5-21. After a topic branch merge.

- If you have a more important project, you might want to use a two-phase merge cycle.
- In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is

integrated into the `develop` branch.

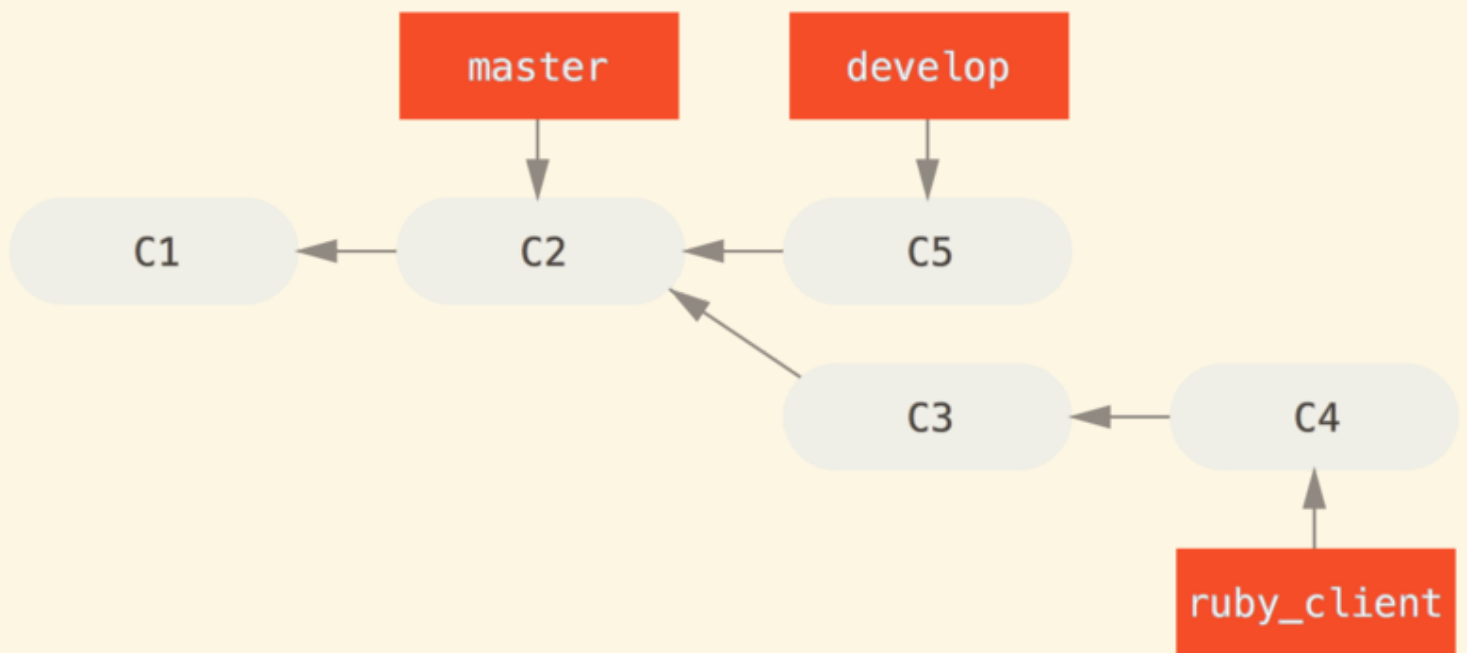


Figure 5-22. Before a topic branch merge.

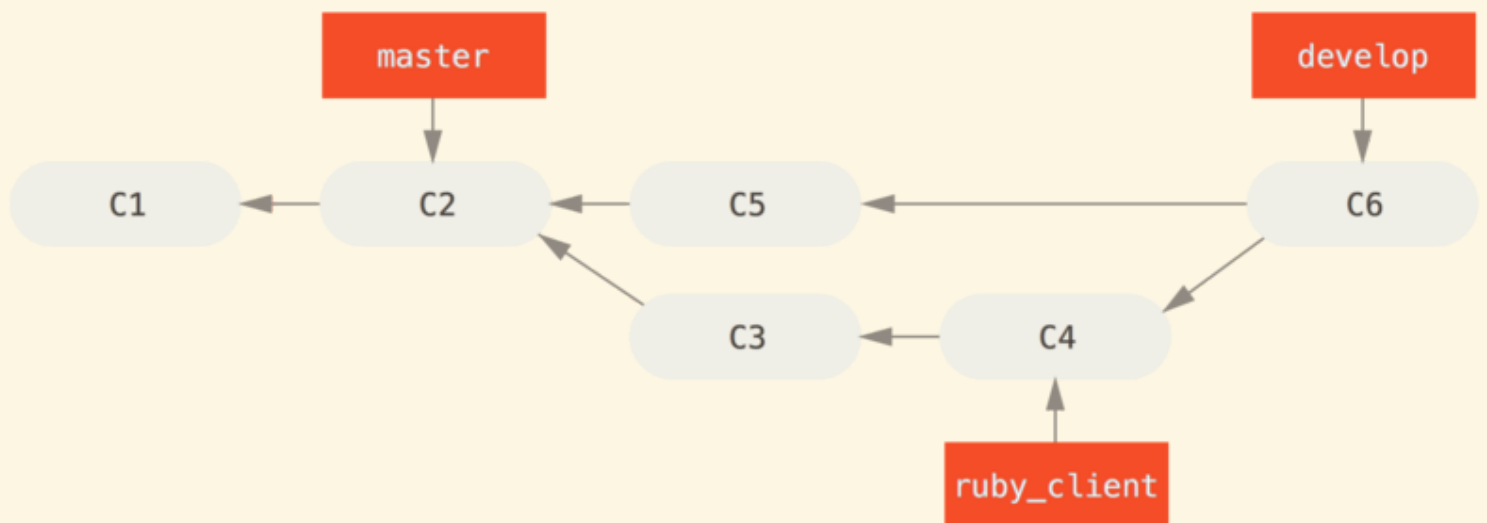


Figure 5-23. After a topic branch merge.

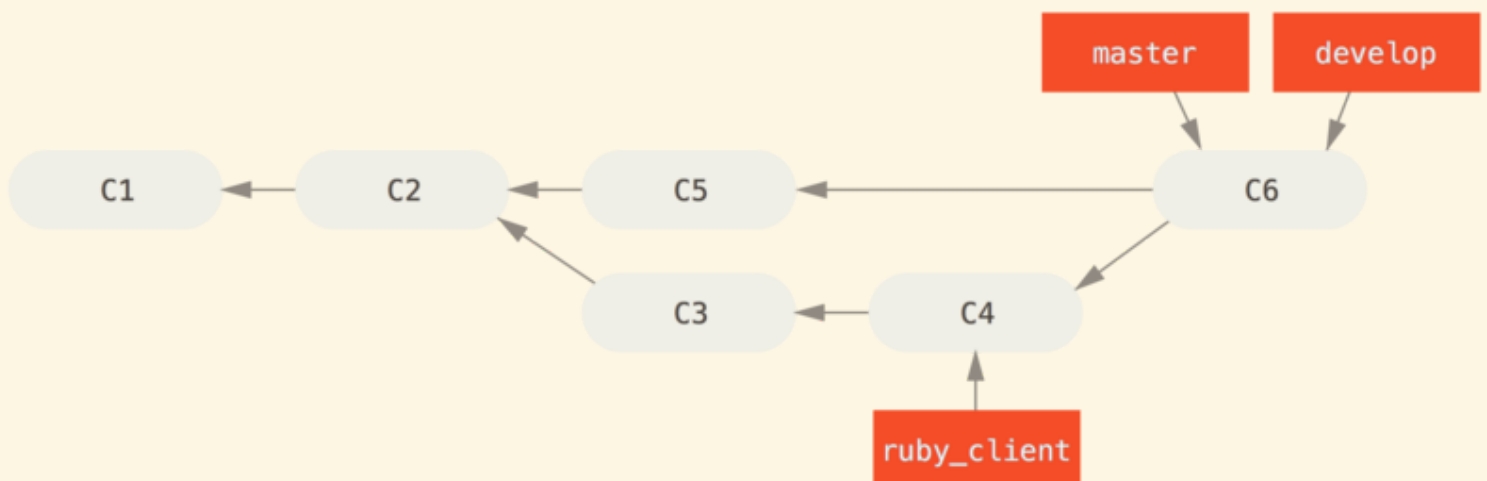


Figure 5-24. After a project release.

- This way, when people clone your projects repository, they can either check out master to build the latest stable version and keep up to date on that easily, or they can check out develop, which is the more cutting-edge stuff.

Large-Merging Workflows

- The Git project has four long-running branches: `master`, `next`, and `pu` (proposed updates) for new work, and `maint` for maintenance backports.
- When new work is introduced by contributors, its collected into topic branches in the maintainers repository in a manner similar to what weve described.
- At this point, the topics are evaluated to determine whether theyre safe and ready for consumption or whether they need more work.
- If theyre safe, theyre merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.

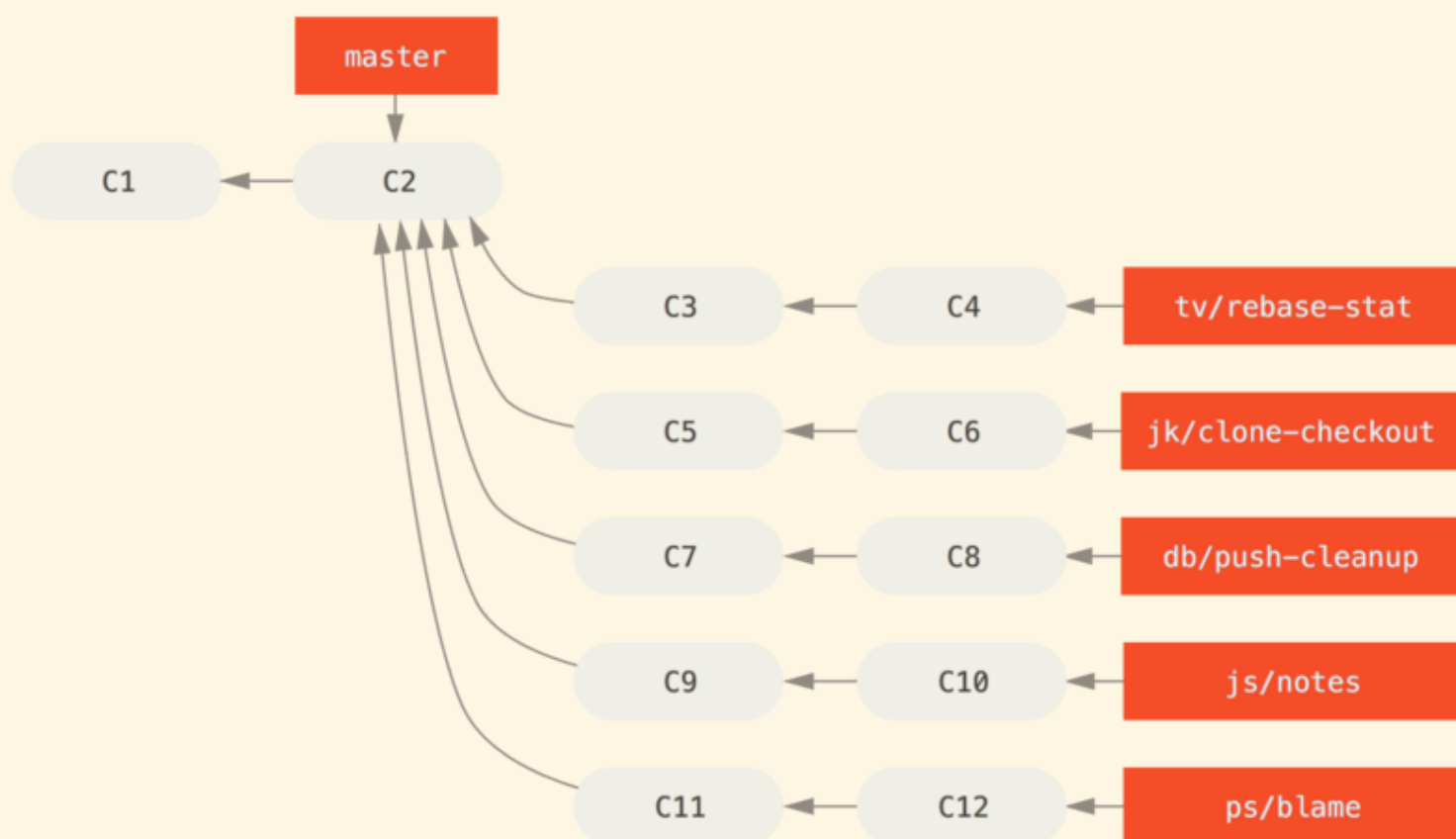


Figure 5-25. Managing a complex series of parallel contributed topics.

If the topics still need work, they're merged into `pu` instead. `master` `next` `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `pu` is rebased even more often:

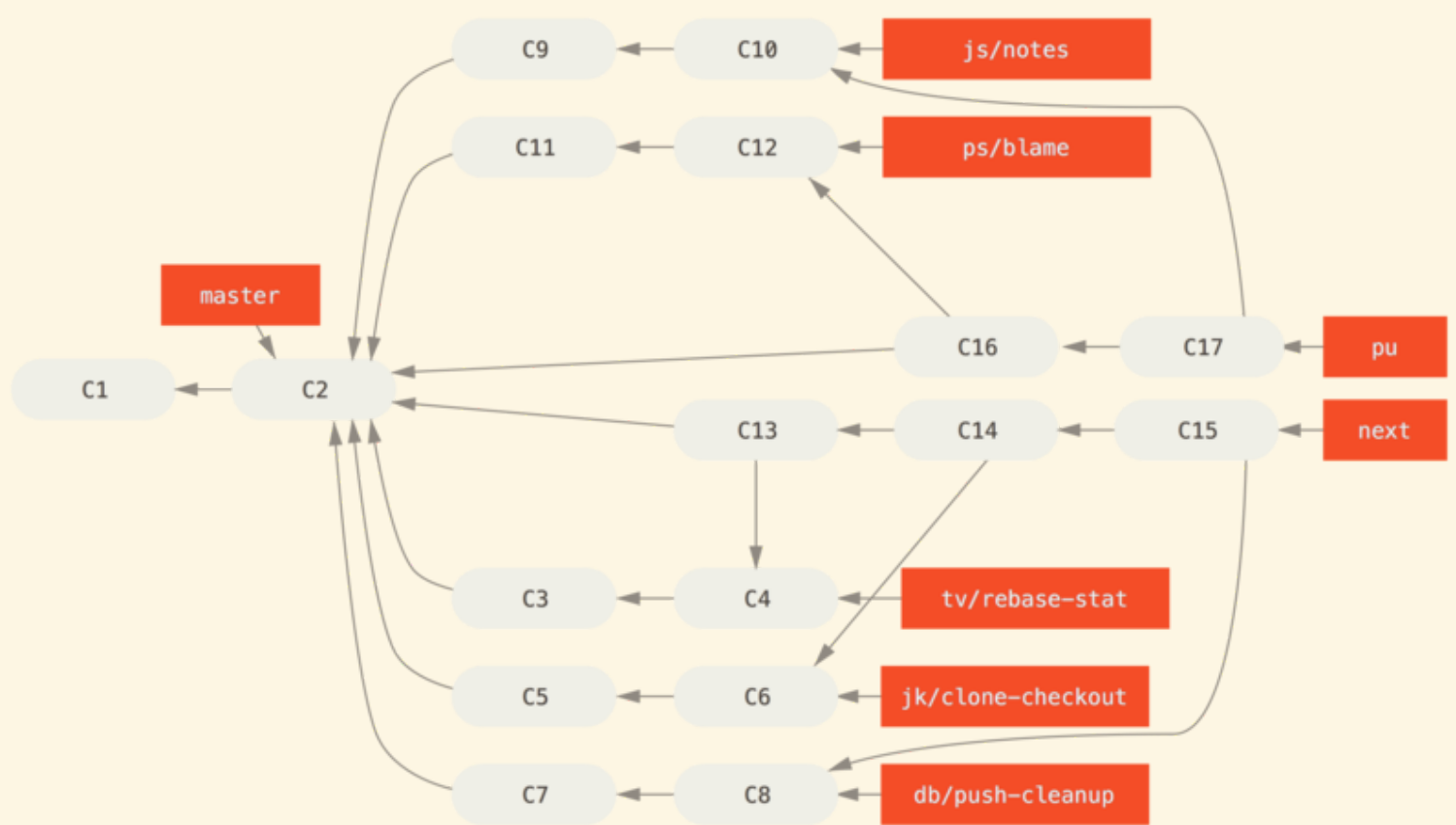


Figure 5-26. Merging contributed topic branches into long-ter.

integration branches. `master`

The Git project also has a `maint` branch that is forked off from the last release to provide backported patches in case a maintenance release is required.

Other maintainers prefer to rebase or cherry-pick contributed work on top of their master branch, rather than merging it in, to keep a mostly linear history. `develop`

If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase.

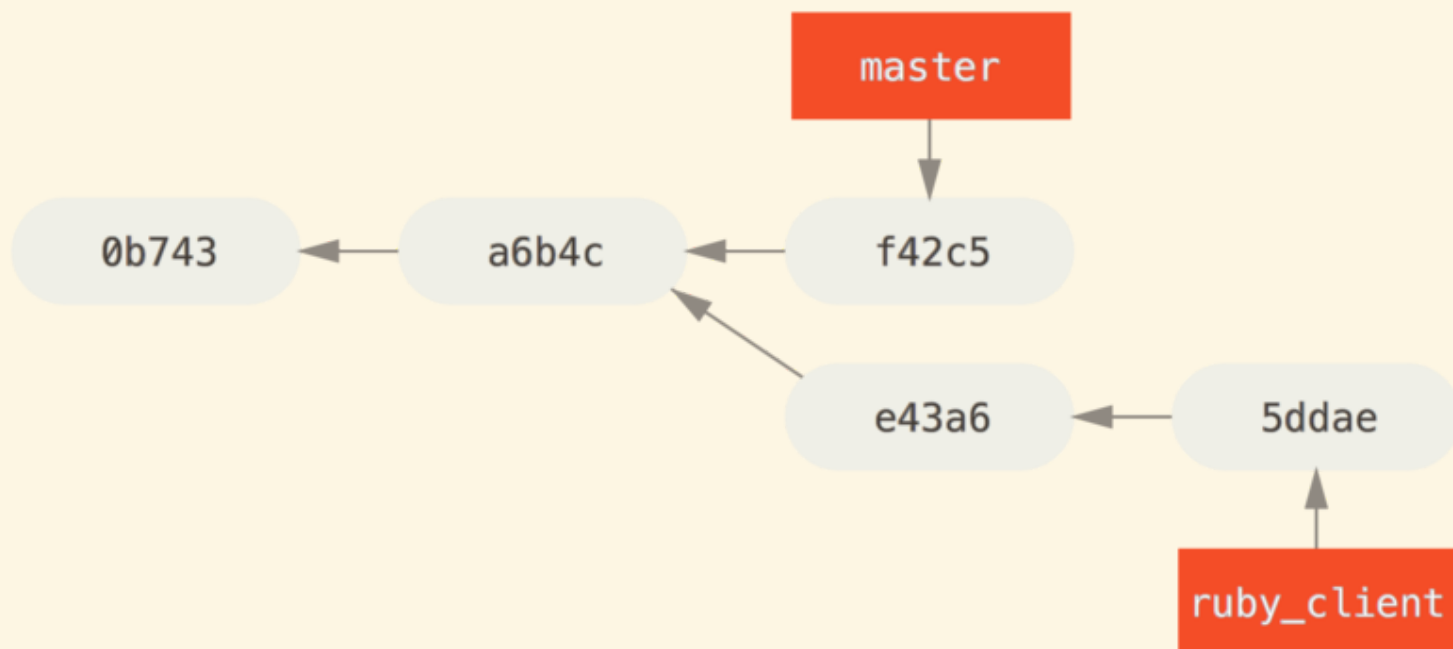


Figure 5-27. Example history before a cherry-pick.

- If you want to pull commit `e43a6` into your master branch, you can run

```

$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcd
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
  
```

- This pulls the same change introduced in `e43a6`, but you get a new commit SHA-1 value, because the date applied is different.
- Now your history looks like this:

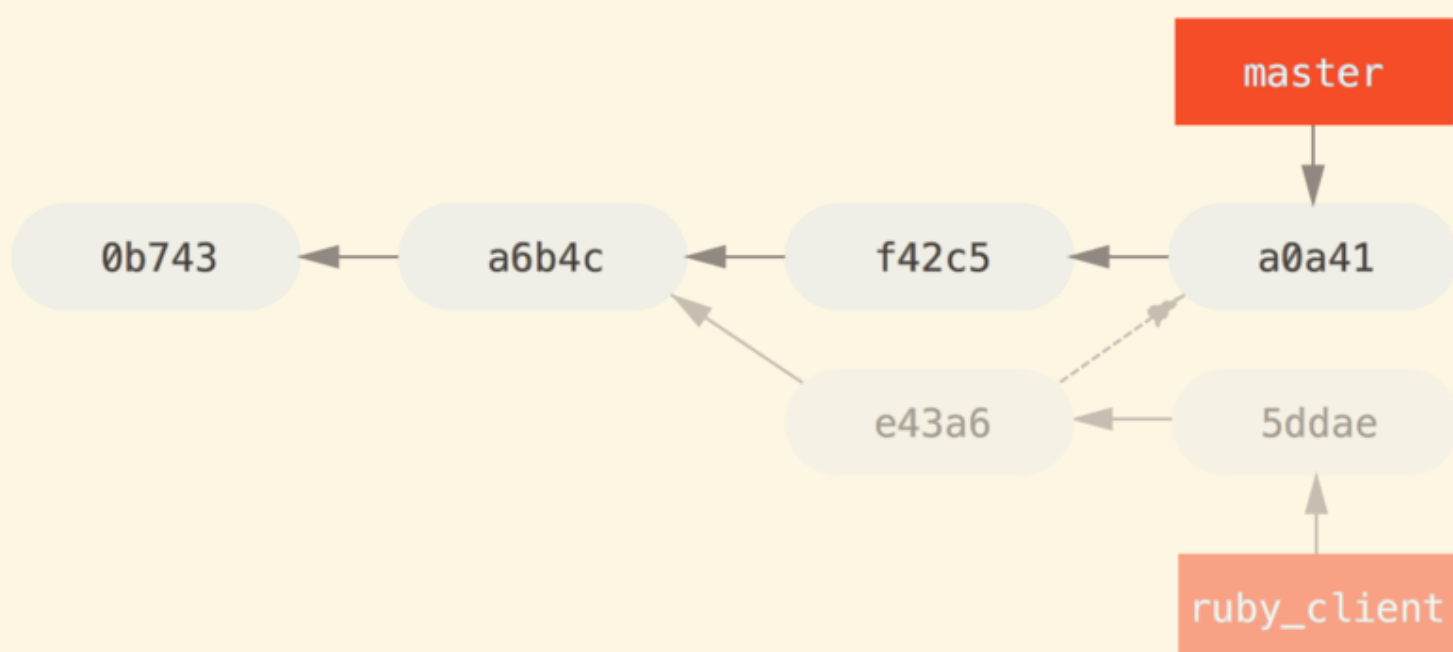


Figure 5-28. History after cherry-picking a commit on a topic branch.

- Now you can remove your topic branch and drop the commits you didn't want to pull in.

Rerere

- If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called rerere that can help.
- Rerere stands for reuse recorded resolution - it's a way of shortcutting manual conflict resolution.
- When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.
- This feature comes in two parts: a configuration setting and a command.
- The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

```
$ git config --global rerere.enabled true
```

- Whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.
- You can interact with the rerere cache using the `git rerere` command.

Tagging Your Releases

- The tag as the maintainer, the tagging may look something like this:

```
$ 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
```

- If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags.
- The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content.
- To do this, you can figure out which key you want by running `gpg --list-keys`:

```
$ 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]
```

- Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

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

- You can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

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

- If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone.
- If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

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

Generating a Build Number

- If you want to have a human-readable name to go with a commit, you can run `git describe` on that commit.
- Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit youre describing:

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

- This way, you can export a snapshot or build and name it something understandable to people.
- The `git describe` command favors annotated tags (tags created with the `-a` or `-s` flag), so release tags should be created this way if youre using `git describe`, to ensure the commit is named properly when described.

Preparing a Release

- The command to do this is `git archive`:

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

- If someone opens that tarball, they get the latest snapshot of your project under a project directory.
- You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

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

- You now have a nice tarball and a zip archive of your project release that you can upload to your website or e-mail to people.

The Shortlog

- A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or e-mail is to use the `git shortlog` command.

```
$ 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
```

- You get a clean summary of all the commits since v1.0.1, grouped by author, that you can e-mail to your list.

Summary

- You should feel fairly comfortable contributing to a project in Git as well as maintaining your own project or integrating other users contributions.
- Congratulations on being an effective Git developer!