Лекция 5. Распределённый Git

Распределённый Git

Теперь, когда вы обзавелись настроенным удалённым Git-репозиторием, являющимся местом, где разработчики могут обмениваться своим кодом, а также познакомились с основными командами Git'а для локальной работы, мы рассмотрим как задействовать некоторые распределённые рабочие процессы, предлагаемые Git'ом.

В этой главе мы рассмотрим работу с Git'ом в распределённой среде как в роли рядового разработчика, так и в роли системного интегратора. То есть вы научитесь успешно вносить свой код в проект, делая это как можно более просто и для вас, и для владельца проекта, а также научитесь тому, как сопровождать проекты, в работе над которыми участвует множество человек.

Распределённые рабочие процессы

В отличие от централизованных систем управления версиями, распределённая природа Git'а позволяет вам быть гораздо более гибким в отношении участия разработчиков в работе над проектами. В централизованных системах все разработчики являются узлами сети, более или менее одинаково работающими на центральном хабе. Однако, в Git каждый разработчик потенциально является и узлом, и хабом. То есть каждый разработчик может как вносить код в другие репозитории, так и содержать публичный репозиторий, на основе которого работают

другие разработчики, и в который они вносят свои изменения. Это даёт вашей команде возможность осуществлять любой из множества различных способов осуществления рабочего процесса в ваших проектах, поэтому мы рассмотрим несколько распространённых подходов, пользующихся гибкостью Git'а. Мы рассмотрим сильные стороны и возможные недостатки каждого подхода;

вы можете выбрать для себя один из них, а можете совместить особенности сразу нескольких подходов.

Централизованный рабочий процесс

В централизованных системах существует, как правило, одна модель совместной разработки

— централизованный рабочий процесс. Один центральный хаб, или репозиторий, может принимать код, а все остальные синхронизируют свою работу с ним. Некоторое число разработчиков являются узлами — клиентами этого хаба — и синхронизируются с ним

одним (смотри Рисунок 5-1).

Это значит, что если два разработчика выполняют клонирование с хаба, и оба делают

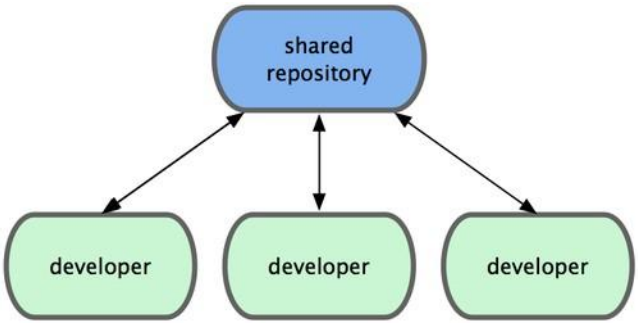

Рисунок 5.1: Централизованный рабочий процесс.

изменения в проекте, то первый из них, кто отправит свои изменения обратно на хаб, сделает это без проблем. Второй разработчик должен взять наработки первого и выполнить слияние перед тем, как отправить свои изменения, так чтобы не перезаписать изменения первого разработчика. Этот принцип справедлив для Git точно также, как и для Subversion (или любой другой ЦСУВ), и в Git такая модель работает отлично.

Если у вас небольшая команда или вас полностью устраивает рабочий процесс централизованного типа, применяемый в вашей компании, вы можете просто продолжить использовать такой рабочий процесс и в Git. Просто настройте один репозиторий и дайте каждому в вашей команде права на отправку изменений; Git не позволит пользователям перезаписывать наработки друг- друга. Если какой-то разработчик склонирует репозиторий, сделает в нём изменения, а затем попытается выложить эти изменения, в то время как другой разработчик уже успел отправить свои, сервер отклонит изменения этого разработчика. Ему будет сказано, что он пытается выложить изменения, для которых невозможно выполнить перемотку (fast-forward), и что надо сначала извлечь данные с сервера, выполнить слияние, а уже потом отправлять свои изменения. Такой рабочий процесс привлекателен для большого количества людей, так как это та модель, с которой многие знакомы, и которая многим понятна.

Рабочий процесс с менеджером по интеграции

Так как Git позволяет иметь несколько удалённых репозиториев, существует возможность ведения такого рабочего процесса, при котором каждый разработчик имеет права на запись в свой собственный публичный репозиторий и права на чтение для всех остальных. Этот сценарий часто подразумевает существование канонического репозитория, который представляет собой «официальный» проект. Чтобы принять участие в работе над

этим проектом, надо создать свою собственную публичную копию проекта и выложить туда свои изменения. Потом вы можете отправить запрос владельцу основного проекта на внесение в него ваших изменений. Он может добавить ваш репозиторий в качестве удалённого, протестировать локально ваши изменения, слить их со своей веткой и затем отправить обратно в публичный репозиторий.

Этот процесс осуществляется следующим образом (смотри Рисунок 5-2):

1. Владелец проекта выкладывает файлы в публичный репозиторий.
2. Участники проекта клонируют этот репозиторий и делают изменения.
3. Участники выкладывают изменения в свои собственные публичные репозитории.
4. Участник проекта отправляет владельцу письмо с просьбой включения его изменений.
5. Владелец проекта добавляет репозиторий участника как удалённый и локально выполняет слияние..
6. Владелец отправляет слитые изменения в основной репозиторий.

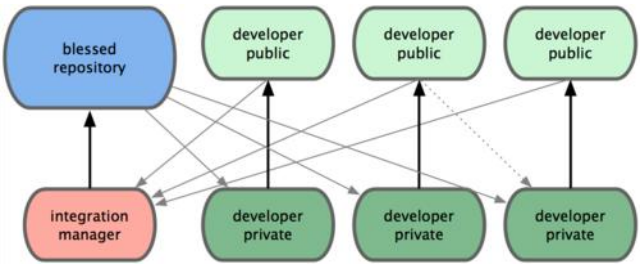

Рисунок 5.2: Рабочий процесс с менеджером по интеграции.

Это очень распространённый тип рабочего процесса для сайтов вроде GitHub, где можно легко форкнуть проект и выложить свои изменения на всеобщее обозрение в собственную копию. Одно из главных преимуществ такого подхода — возможность продолжать работать, в то время, как владелец основного репозитория может включить себе ваши изменения, когда ему угодно. Участникам проекта не придётся ждать, пока их изменения не будут включены в проект — каждый может работать в своём собственном ритме.

Рабочий процесс с диктатором и его помощниками

Это одна из разновидностей рабочего процесса с множеством репозиториев. В основном он используется в огромных проектах с сотнями участников; ядро Linux яркий тому пример. Несколько менеджеров по интеграции заведуют разными частями

репозитория; этих людей называют помощниками. У всех этих помощников есть только один менеджер по интеграции, которого называют благосклонным диктатором. Репозиторий благосклонного диктатора служит эталонным репозиторием, откуда все участники проекта должны брать изменения. Этот процесс происходит так (смотри Рисунок 5-3):

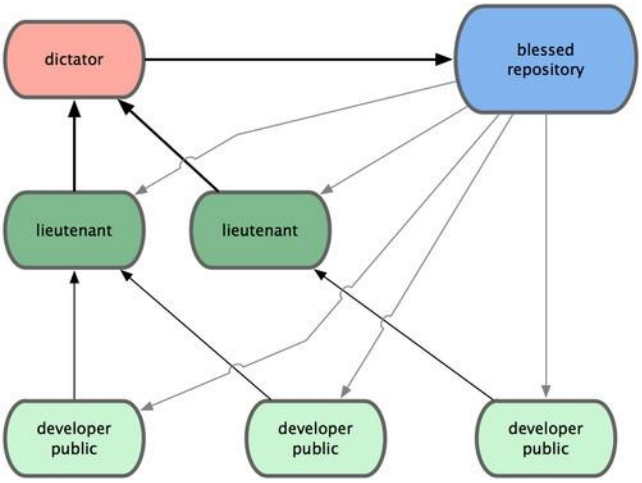

Рисунок 5.3: Рабочий процесс с благосклонным диктатором.

Этот тип рабочего процесса не является распространённым, но он может быть полезен в очень больших проектах или в сильно иерархическом окружении, так как он позволяет лидеру проекта (диктатору) передать другим полномочия по выполнению большой части работ и собирать код большими порциями с нескольких мест перед его интеграцией.

Мы рассмотрели несколько широко используемых типов рабочих процессов доступных при работе с распределёнными системами вроде Git, но, как видите, возможны различные вариации для подгонки под ваш конкретный тип рабочего процесса. Теперь, когда вы в состоянии определить, какая комбинация рабочих процессов сработает для вас лучше, мы рассмотрим несколько более специфичных примеров действий, выполняемых основными ролями участников различных процессов.

Содействие проекту

Мы узнали, что представляют собой различные рабочие процессы, также у вас должно быть достаточно хорошее понимание основ использования Git. В этом разделе вы узнаете о нескольких типичных способах внести свой вклад в проект.

Главная трудность в описании этого процесса состоит в том, что существует огромное количество вариаций того, как он организован. Так как Git очень гибок, люди могут осуществлять совместную работу по-разному, и проблематично описать то, как вы должны содействовать проекту — все проекты немного разные. Многое зависит от количества активных участников, от выбранного типа рабочего процесса, от ваших прав доступа к репозиториям, и, возможно, от метода принятия изменений от внешних разработчиков.

Первый фактор — это количество активных участников. Как много пользователей активно вносят свой вклад в проект и как часто? Во многих случаях это два-три разработчика с несколькими коммитами в день, возможно, меньше, для вялотекущих проектов. В по-настоящему больших компаниях или проектах число разработчиков может измеряться тысячами, с десятками или даже сотнями ежедневно поступающих патчей. Это важно, поскольку с увеличением числа разработчиков вам становится труднее убедиться, что ваши изменения можно будет чисто применить или беспрепятственно слить. Изменения, которые вы отправляете, могут оказаться устаревшими или частично сломанными той работой, которая была влита, пока вы работали, или пока ваши изменения ожидали утверждения или применения. Как сохранить свой код согласованным, а патчи применимыми?

Следующий фактор — это рабочий процесс, используемый в проекте. Он централизован, и каждый разработчик имеет равные права на запись в главный репозиторий? Есть у проекта мейнтейнер или менеджер по интеграции, который проверяет патчи? Все ли патчи проверяются и утверждаются экспертами? Вы вовлечены в этот процесс? Присутствует ли система помощников и должны ли вы сначала отправлять свою работу им?

Следующий пункт — это доступ на отправку изменений. Рабочий процесс, требуемый для внесения вклада в проект сильно отличается в зависимости от того, имеете ли вы доступ на запись или нет. Если у вас нет доступа на запись, то как в проекте принято принимать вклад в работу? Вообще, существует ли какая-либо политика? Какой объём работы вы вносите за раз? Как часто вы это делаете?

Все эти вопросы могу повлиять на то, как эффективно вы будете вносить вклад в проект и какой рабочий процесс предпочтителен или доступен вам. Я расскажу об аспектах каждого из них на серии примеров, продвигаясь от простых к более сложным; на основе этих примеров вы сможете создать специфический нужный вам в вашей работе тип рабочего процесса.

Рекомендации по созданию коммитов

Прежде чем мы приступим к рассмотрению специфичных примеров использования, сделаем короткое замечание о сообщениях коммитов. Обладание хорошим руководством по созданию коммитов и следование ему значительно облегчает работу с Git’ом и сотрудничество с другими разработчиками. У проекта Git имеется документ с хорошими советами по созданию коммитов, из которых делаются патчи — прочитать его можно в исходном коде Git в файле Documen- tation/SubmittingPatches.

Во-первых, не стоит отсылать ничего с ошибками в пробельных символах. Git предоставляет простой способ их обнаружения — перед коммитом, запустите git diff --

check, это определит возможные проблемы и перечислит их вам. Вот пример, в котором я заменил красный цвет терминала символами X:

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

Если выполните эту команду перед коммитом, то сможете понять, собираетесь ли вы сделать коммит с раздражающими разработчиков ошибками в пробельных символах.

Далее, старайтесь делать так, чтобы каждый коммит был логически отдельным набором изменений. Если можете, старайтесь делать ваши изменения обозримыми — не стоит писать код все выходные, работая над пятью задачами, а затем отправлять их все в понедельник одним массивным коммитом. Даже если вы не делали коммитов в течение выходных, воспользуйтесь индексом, чтобы разбить свою работу на части, как минимум по одному коммиту для каждой проблемы с полезным сообщением к каждому. Если некоторые из изменений затрагивают один и тот же файл, попробуйте использовать git add --patch для индексирования файла по частям (это подробно рассмотрено в Главе 6). Снимок состояния проекта на верхушке ветки будет идентичным, сделаете ли вы один коммит или пять, покуда все ваши изменения добавлены в какой-то момент, так что попытайтесь облегчить жизнь вашим коллегам разработчикам, когда они будут просматривать ваши изменения. При таком подходе будет проще выделить или отменить одно из изменений, если возникнет такая необходимость. В главе 6 описано множество полезных ухищрений для переписывания истории и интерактивного индексирования файлов — пользуйтесь этими инструментами для изготовления ясной и понятной истории.

Последняя вещь, которую стоит иметь в виду, — это сообщение коммита. Написание качественных сообщений коммитов должно войти в привычку, это сделает сотрудничество с использованием Git’а гораздо проще. По общему правилу, ваши сообщения должны начинаться с одной строки не длиннее 50 символов, лаконично описывающей набор изменений, затем

пустая строка, затем более детальное описание. Проект Git требует, чтобы детальное объяснение включало в себя мотивацию на изменения и противопоставляло вашу реализацию с предыдущим поведением — это хорошее руководство к действию. Если вы пишите сообщения к коммитам на английском языке, то хорошей идеей является использование

повелительного наклонения глаголов в настоящем времени. Другими словами, пишите команды. Вместо «I added tests for» или «Adding tests for» используйте «Add tests for».

Вот шаблон, изначально написанный Тимом Поупом на сайте tpope.net:

```
Краткое (до 50 символов) описание изменений

Более детальное объяснение, если необходимо. Перенос на 72 символе
или около того. В некоторых контекстах, первая строка рассматривается
как тема письма, а остальное телом. Пустая строка, отделяющая сводку
от тела важна (если вы не опустили тело целиком); если вы оставите их
вместе, инструменты такие, как rebase, могут воспринять это неправильно.

Дальнейшие параграфы идут после пустых строк

- также можно применять маркеры списков
```

Если все ваши сообщения о коммитах будут выглядеть как это, всё будет намного проще для вас и для разработчиков, с которыми вы работаете. Проект Git содержит хорошо отформатированные сообщения о коммитах — я советую вам запустить git log --no-merges там, чтобы увидеть, как выглядит хорошо отформатированная история коммитов проекта.

В последующих примерах и почти везде в этой книге для краткости я не форматирую сообщения так красиво, как это; вместо этого я использую опцию -mдля команды git com- mit. Делайте, как я говорю, а не как я делаю.

Отдельная маленькая команда

Наиболее простой тип организации, с которой вы легко можете столкнуться — частный проект с одним или двумя другими разработчиками. Под термином частный я подразумеваю закрытый код, недоступный для чтения остальному миру. Вы и все остальные разработчики имеете право записи в репозиторий.

В этой среде вы можете придерживаться рабочего процесса, похожего на тот, который вы бы использовали в Subversion или другой централизованной системе. Вы по-прежнему получите такие преимущества, как локальные коммиты (коммиты в offline) и возможность гораздо более простого ветвления и слияния, но сам рабочий процесс может оставаться очень похожим; главное отличие — во время выполнения коммита слияние происходит на стороне клиента, а не на сервере. Давайте посмотрим, как выглядел бы процесс, когда два разработчика начинают работать вместе с общим репозиторием. Первый

разработчик, Джон, клонирует репозиторий, делает изменения и создаёт локальный коммит. (Я заменяю служебные сообщения знаком ...в этих примерах, чтобы немного их сократить.)

```
# Машина Джона

$ 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
```

Второй разработчик, Джессика, выполняет то же самое — клонирует репозиторий и делает коммит с изменениями:

```
# Машина Джессики

$ git clone jessica@githost:simplegit.git

Initialized empty Git repository in /home/jessica/simplegit/.git/

...

$ cd simplegit/
```

Теперь Джессика отправляет свою работу на сервер:

```
# Машина Джессики

$ git push origin master

...
```

Джон также пытается выложить свои изменения:

```
# Машина Джона

$ 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'
```

Джон не может выполнить отправку изменений, так как за это время Джессика уже отправила свои. Это очень важно понять, особенно если вы привыкли к Subversion, так как

мы видим, что эти два разработчика не редактировали один и тот же файл. Хотя Subversion и выполняет автоматическое слияние на сервере, если редактировались разные файлы, при использовании Git вы должны слить коммиты локально. Прежде чем Джон сможет отправить свои изменения на сервер, он должен извлечь наработки Джессики и выполнить слияние:

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

На этот момент, локальный репозиторий Джона выглядит так, как показано на Рисунке 5.4

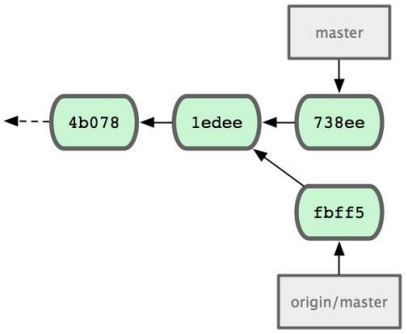

Рисунок 5.4: Исходный репозиторий Джона.

У Джона есть ссылка на изменения, выложенные Джессикой, и он должен слить их со своей работой перед тем, как ему разрешат её отправить:

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

Слияние прошло без проблем — история коммитов Джона теперь выглядит как на Рисунке 5.5

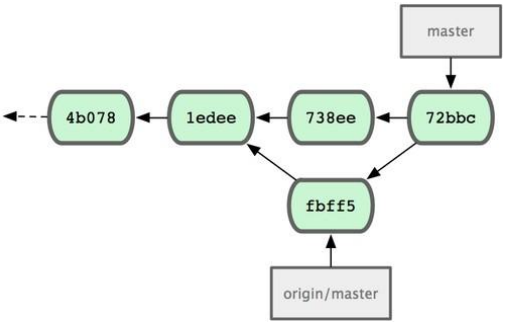

Рисунок 5.5: Репозиторий Джона после слияния с origin/master.

Теперь Джон может протестировать свой код, дабы удостовериться, что он по-прежнему работает нормально, а затем выложить свою работу, уже объединённую с работой Джессики, на сервер:

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

В результате история коммитов Джона выглядит, как показано на Рисунке 5-6.

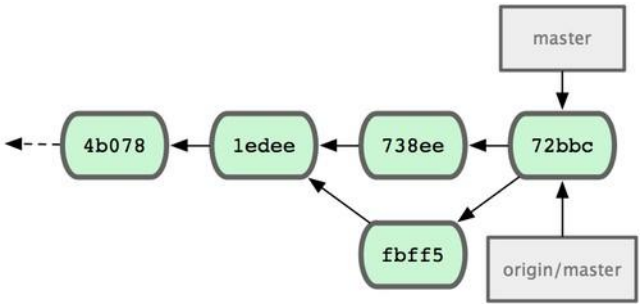

Рисунок 5.6: История коммитов Джона после отправки изменений на сервер.

Тем временем, Джессика работала над тематической веткой. Она создала тематическую ветку с названием issue54 и сделала три коммита в этой ветке. Она ещё не извлекала изменения Джона, так что её история коммитов выглядит, как показано на Рисунке 5-7.

Джессика хочет синхронизировать свою работу с Джоном, так что она извлекает изменения с сервера:

```
# Машина Джессики
$ git fetch origin
...
From jessica@githost:simplegit
```

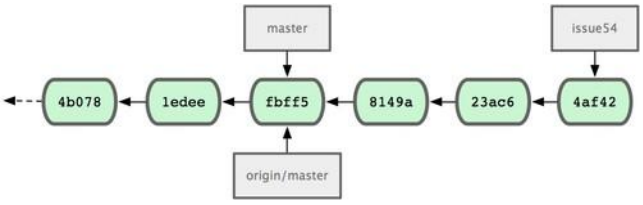

Рисунок 5.7: Исходная история коммитов Джессики.

```
fbff5bc..72bbc59  master     -> origin/master
```

Эта команда извлекает наработки Джона, которые он успел выложить. История коммитов Джессики теперь выглядит как на Рисунке 5-8.

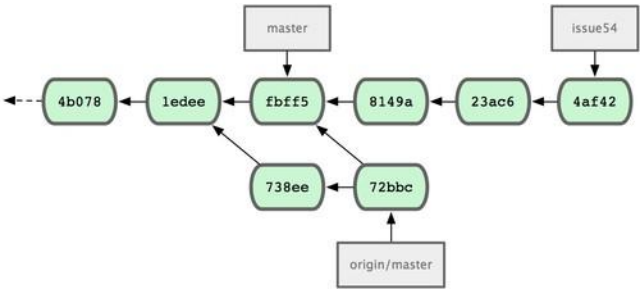

Рисунок 5.8: История коммитов Джессики после извлечения изменений Джона.

Джессика полагает, что её тематическая ветка закончена, но она хочет узнать, с чем ей нужно слить свою работу, чтобы она могла выложить её на сервер. Она запускает git log, чтобы выяснить это:

```
$ 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
```

Теперь Джессика может слить свою тематическую ветку в ветку master, слить работу Джона (origin/master) в свою ветку masterи затем отправить изменения на сервер. Сначала она переключается на свою основную ветку, чтобы объединить всю эту работу:

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

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

Она может слить сначала ветку origin/master, а может и issue54 — обе они находятся выше в истории коммитов, так что не важно какой порядок слияния она выберет. Конечное состояние репозитория должно получиться идентичным независимо от того, какой порядок слияния она выберет; только история коммитов будет немного разная. Она решает слить ветку issue54первой:

```
$ git merge issue54

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

Никаких проблем не возникло; как видите, это был обычная перемотка. Теперь Джессика сливает работу Джона (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(-)
```

Слияние проходит нормально, и теперь история коммитов Джессики выглядит так, как показано на Рисунке 5-9.

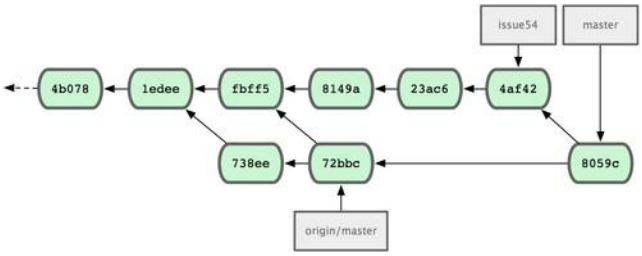

Рисунок 5.9: История коммитов Джессики после слияния с изменениями Джона.

Теперь указатель origin/masterдоступен из ветки masterДжессики, так что она может спокойно выполнить git push(полагая, что Джон не выкладывал свои изменения за это время):

```
                            $ git push origin master

                                        ...

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

Каждый разработчик несколько раз выполнял коммиты и успешно сливал свою работу с работой другого; смотри Рисунок 5-10.

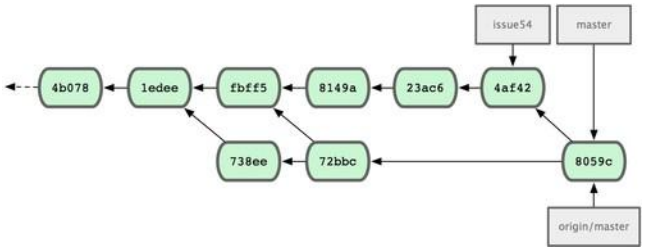

Рисунок 5.10: История коммитов Джессики после отправки всех изменений обратно на сервер.

Это один из простейших рабочих процессов. Вы работаете некоторое время, преимущественно в тематической ветке, и, когда приходит время, сливаете её в свою ветку master. Когда вы готовы поделиться этой работой с другими, вы сливаете её в ветку master, извлекаете изменения с сервера и сливаете origin/master(если за это время произошли изменения), и, наконец, отправляете свои изменения в ветку master на сервер. Общая последовательность действий выглядит так, как показано на Рисунке 5-11.

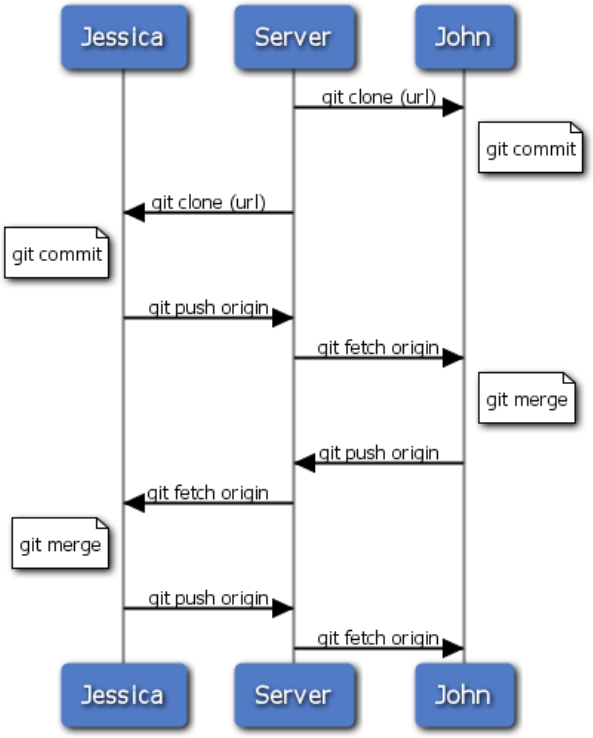

Рисунок 5.11: Общая последовательность событий для простого рабочего процесса с несколькими разработчиками в Git'е.

Отдельная команда с менеджером

В этом сценарии мы рассмотрим роли участников проекта в закрытых группах большего размера. Вы научитесь работе в окружении, где маленькие группы совместно работают над задачами, а затем результаты их деятельности интегрируются отдельным субъектом.

Давайте представим, что Джон и Джессика работают вместе над одной задачей, в то время как Джессика с Джози работают над другой. В этом случае компания использует рабочий процесс с менеджером по интеграции, при котором работа частных групп объединяется только

определёнными инженерами (обновление ветки master главного репозитория может осуществляться только этими инженерами). В этом случае вся работа выполняется в ветках отдельных команд разработчиков и впоследствии объединяется воедино менеджерами по интеграции.

Давайте проследим за рабочим процессом Джессики, которая работает над двумя задачами, сотрудничая одновременно с двумя разными разработчиками. Положим, что она уже имеет свою собственную копию репозитория. Джессика решает сначала взяться за задачу fea- tureA. Для этого она создаёт новую ветку и выполняет в ней некоторую работу:

```
                                    # Машина Джессики

                                  $ 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(-)
```

На этом этапе ей требуется поделиться своей работой с Джоном, так что она отправляет коммиты, выполненные на ветке featureA, на сервер. Так как Джессика не имеет право на изменение ветки master на сервере — только менеджеры по интеграции могут делать это она вынуждена отправлять свои изменения в другую ветку, чтобы обмениваться работой с Джоном:

```
                                  $ git push origin featureA

                                             ...

                               To jessica@githost:simplegit.git

                             * [new branch]      featureA -> featureA
```

Джессика сообщает по электронной почте Джону, что она выложила некоторые наработки в ветку featureA, и что он может проверить их. Пока Джессика ждёт ответа от Джона, она решает начать работу над веткой featureBвместе с Джози. Для начала она создаёт новую ветку для этой задачи, используя в качестве основы ветку masterна сервере:

```
                                    # Машина Джессики

                                   $ git fetch origin

$ git checkout -b featureB origin/master
  Switched to a new branch "featureB"
```

Теперь Джессика делает пару коммитов в ветке featureB:

```
                                   $ 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(-)
```

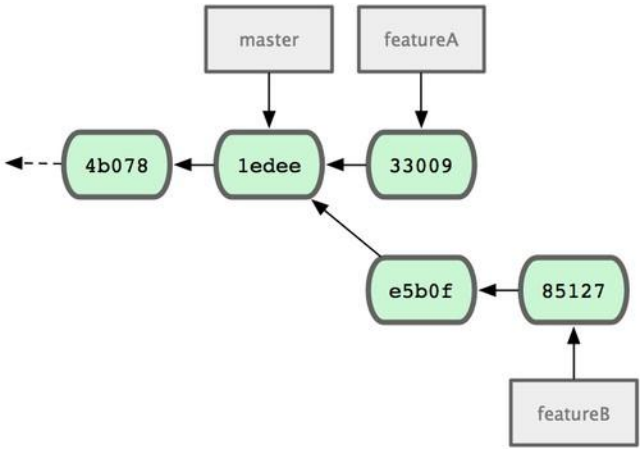

Рисунок 5.12: Исходная история коммитов у Джессики.

Репозиторий Джессики выглядит, как показано на Рисунке 5-12.

Джессика уже готова отправить свою работу на сервер, но получает от Джози сообщение о том, что некоторые наработки уже были выложены на сервер в ветку featureBee. Поэтому Джессика должна сначала слить эти изменения со своими, прежде чем она сможет отправить свою работу на сервер. Она может извлечь изменения Джози командой git fetch:

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

Теперь Джессика может слить эти изменения в свои наработки командой 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(-)
```

Есть небольшая проблема — ей нужно выложить изменения из своей ветки featureB в ветку featureBee на сервере. Она может сделать это при помощи команды git push, указав название локальной и удалённой ветки, разделённые двоеточием:

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

Это называется *refspec*. Смотри Главу 9, где более детально обсуждаются спецификации ссылок и различные вещи, которые вы можете делать с ними.

Далее, Джон сообщает Джессике по почте, что он добавил некоторые изменения в ветку featureA и просит её проверить их. Она выполняет git fetch, чтобы получить внесённые Джоном изменения:

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

Затем, используя команду git log, она смотрит, что же было изменено:

```
$ 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
```

Наконец, она сливает работу Джона в свою собственную ветку featureA:

```
   $ 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(-)
```

Джессика хочет кое-что подправить, так что она опять делает коммит и затем отправляет изменения на сервер:

```
$ 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
```

История коммитов Джессики теперь выглядит так, как показано на Рисунке 5-13.

Джессика, Джози и Джон информируют менеджеров по интеграции, что ветки featureA и featureBee на сервере готовы к интеграции в основную ветку разработки. После того, как они интегрируют эти ветки в основную версию, извлечение данных с сервера приведёт к появлению новых коммитов слияния. Таким образом, история коммитов станет выглядеть так, как на Рисунке 5-14.

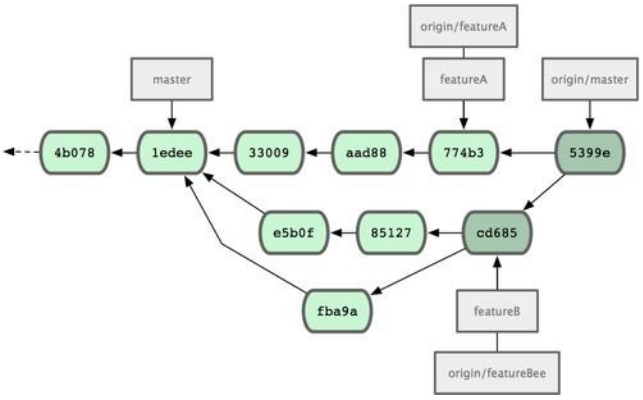

Рисунок 5.14: История коммитов Джессики после слияния двух тематических веток.

Множество групп переходят на Git именно из-за возможности параллельной работы нескольких команд с последующим объединением разных линий разработки. Огромное преимущество

Git'a — возможность маленьких подгрупп большой команды работать вместе через удалённые ветки, не мешая при этом всей команде. Последовательность событий в рассмотренном здесь рабочем процессе представлена на Рисунке 5-15.

Небольшой открытый проект

Внести вклад в открытый проект — это немного другое. Из-за того, что у вас нет прав на прямое изменение веток проекта, требуется какой-нибудь другой путь для обмена наработками с мейнтейнерами. Первый пример описывает участие в проекте через разветвление (fork) на Git-хостингах, на которых это делается достаточно просто. Сайты repo.or.cz и GitHub оба поддерживают такую возможность, и большая часть мейнтейнеров проектов придерживаются такого способа сотрудничества. В следующем разделе рассматриваются проекты, которые предпочитают принимать патчи по e-mail.

Сначала вы скорее всего захотите склонировать основной репозиторий, создать тематическую ветку для одного или нескольких патчей, которые вы собираетесь внести в проект, и выполнить свою работу в ней. Последовательность действий выглядит следующим образом:

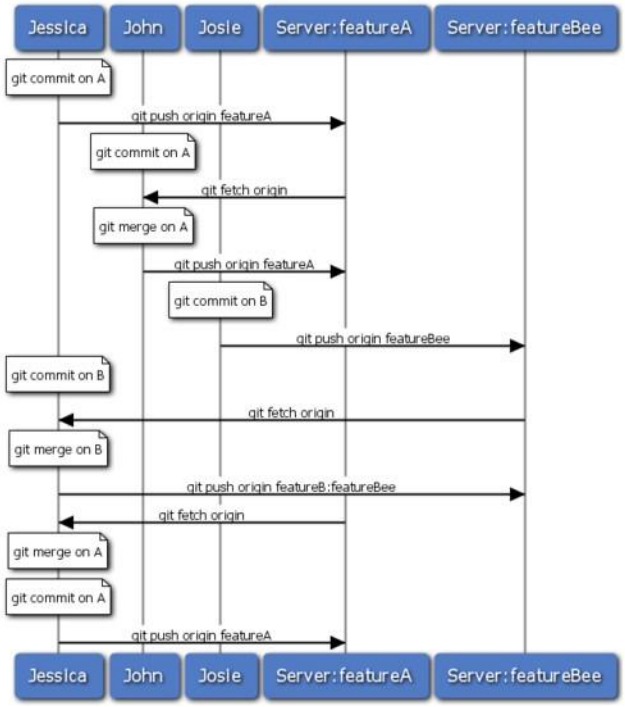

Рисунок 5.15: Основная последовательность действий для рабочего процесса в команде с менеджером по интеграции.

```
$ git clone (url)

$ cd project

$ git checkout -b featureA

$ (выполнение работы)

$ git commit

$ (выполнение работы)
```

Возможно, у вас возникнет желание воспользоваться rebase-i, чтобы сплющить (squash) свои наработки в единый коммит, или реорганизовать наработки в коммитах таким образом, чтобы их было проще воспринимать мейнтейнерам проекта — об интерактивном перемещении будет рассказано в Главе 6.

Если вы закончили работу со своей веткой и готовы поделиться наработками с мейнтейнерами, перейдите на страницу исходного проекта и нажмите кнопку «Fork», создав таким образом свою собственную копию проекта доступную на запись. Затем вам нужно добавить URL этого нового репозитория в список удалённых репозиториев, в нашем случае мы назовём его my- fork:

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

Вам нужно отправить свои наработки в этот репозиторий. Проще всего будет отправить в удалённый репозиторий ту ветку, над которой вы работаете, а не сливать её в ветку master и отправлять потом его. Это объясняется следующим образом — если ваша

работа не будет принята или будет принята только частично, вам не придётся откатывать назад свою ветку master. Если мейнтейнеры сольют, переместят или частично включат вашу работу, вы, в конечном счёте, получите её обратно при получении изменений из их репозитория:

```
$ git push myfork featureA
```

Когда ваши наработки будут отправлены в ваш форк, вам нужно будет послать уведомление мейнтейнеру. Его часто называют запросом на включение (pull request), вы можете либо сгенерировать его через сайт — на GitHub'е есть кнопка «pull request», автоматически уведомляющая мейнтейнера, либо выполнить команду gitrequest-pullи вручную отправить её вывод по почте мейнтейнеру.

Команда request-pull принимает в качестве аргумента имя базовой ветки, в которую вы хотите включить свою работу, и URL репозитория, из которого мейнтейнер может получить ваши наработки. Команда выводит короткую сводку всех изменений, которые вы просите включить в проект. Например, если Джессика хочет послать Джону запрос на включение, когда она сделала пару коммитов в тематической ветке и уже отправила её на сервер, ей следует выполнить следующее:

```
$ 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(-)
```

Вывод может быть отправлен мейнтейнеру — он содержит список коммитов, информацию о том, где начинается ветка с изменениями, и указывает откуда можно забрать эти изменения. Для проекта, мейнтейнером которого вы не являетесь, проще иметь ветку master, которая отслеживает ветку origin/master, и выполнять работу в тематических ветках, которые вы легко сможете удалить, в случае если они будут отклонены. Если вы распределяете свои наработки по различным темам в тематических ветках, вам будет проще выполнить перемещение своей работы, в случае если верхушка главного репозитория

переместится за время работы и ваши коммиты уже не получится применить без конфликтов. Например, если вы планирует отправить в проект работу по другой теме, не продолжайте работать внутри тематической ветки, которую вы только что отправили, начните снова с ветки masterглавного репозитория:

```
$ git checkout -b featureB origin/master
```

```
$ (выполнение работы)

$ git commit

$ git push myfork featureB

$ (отправка письма мейнтейнеру)

$ git fetch origin
```

Теперь каждая из ваших тем представляет собой нечто похожее на очередь из патчей, которую вы можете перезаписывать, перемещать, модифицировать, не оказывая влияние на остальные, как на Рисунке 5-16.

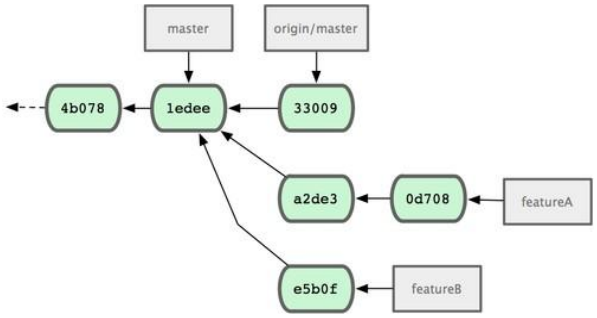

Рисунок 5.16: Исходная история коммитов при работе над featureB.

Давайте представим, что мейнтейнер проекта включил в основную версию чью-то группу патчей. Затем он попытался включить вашу первую ветку, но слияние уже не проходит гладко. В этом случае вы можете попробовать переместить эту ветку на верхушку ветки origin/ master, разрешить конфликты для мейнтейнера и затем заново представить свои изменения на рассмотрение:

```
$ git checkout featureA

$ git rebase origin/master

$ git push -f myfork featureA
```

Так вы перепишете свою историю коммитов, чтобы она выглядела так, как на Рисунке 5-17.

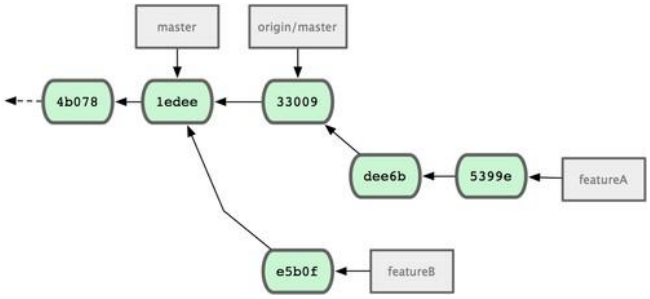

Рисунок 5.17: История коммитов после работы в featureA.

Так как вы переместили ветку, команде push вы должны передать опцию -f, чтобы иметь возможность заменить ветку featureA на сервере. Есть альтернатива — выложить новую работу на сервер в другую ветку (возможно, назвав её featureAv2).

Давайте рассмотрим более вероятный сценарий: мейнтейнер просмотрел на вашу работу во второй ветке и ему понравилась ваша идея, но он хотел бы, чтобы вы изменили некоторые детали реализации. Воспользуемся этой возможностью, чтобы заодно переместить вашу работу так, чтобы она базировалась на текущей версии ветки master в проекте. Создадим новую ветку, базирующуюся на текущей ветке origin/master, уплотним (squash) здесь изменения из ветки featureB, разрешим все конфликты, которые могут возникнуть, сделаем необходимые изменения в реализации вашей идеи и затем выложим всё это в виде новой ветки:

```
$ git checkout -b featureBv2 origin/master

$ git merge --no-commit --squash featureB

$ (изменение реализации)

$ git commit

$ git push myfork featureBv2
```

Опция --squash берёт всю работу на сливаемой ветке (featureB) и сжимает её в один коммит, не являющийся коммитом-слиянием, и помещает его на верхушку текущей ветки. Опция --no-commit сообщает Git'у, что не нужно автоматически записывать коммит. Это позволит вам внести все изменения с другой ветки и затем сделать ещё ряд изменений перед записью нового коммита.Теперь вы можете отправить мейнтейнеру сообщение о том, что вы сделали требуемые изменения, и они могут быть найдены в вашей ветке featureBv2 (смотри Рисунок 5-18).

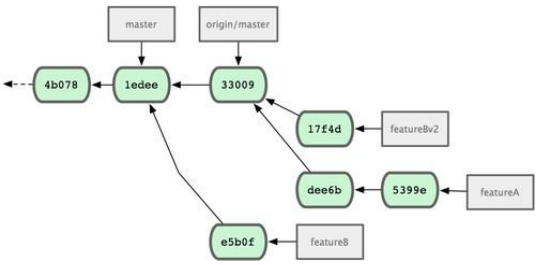

Рисунок 5.18: История коммитов после работы над featureBv2.

Большой открытый проект

Во многих крупных проектах есть установленные процедуры принятия патчей — вам потребуется выяснить точные правила для каждого проекта отдельно, так как они везде разные. Однако, многие крупные открытые проекты принимают патчи через списки рассылки для разработчиков, так что мы сейчас рассмотрим пример использования этого способа.

Рабочий процесс похож на описанный ранее — вы создаёте тематическую ветку для каждой серии патчей, над которой работаете. Отличие состоит в процессе внесения этих изменений в проект. Вместо того, чтобы создавать ответвление (fork) от проекта и отправлять наработки в свой собственный репозиторий с правами на запись, вы генерируете e-mail версию каждой серии коммитов и отправляете её в список рассылки для разработчиков:

```
$ git checkout -b topicA
$ (выполнение работы)
$ git commit
$ (выполнение работы)
$ git commit
```

Теперь у нас есть два коммита, которые теперь нужно отправить в список рассылки. Воспользуемся командой git format-patch, чтобы сгенерировать файлы в формате mbox, которые вы сможете отправить по почте. Эта команда превращает каждый коммит в электронное письмо, темой которого является первая строка сообщения коммита, а оставшаяся часть сообщения коммита и патч, который он представляет, являются телом письма. Хорошей особенностью этого является то, что применение патча из сгенерированного командой format-patch электронного письма сохраняет всю информацию о коммите. Мы увидим это в следующем разделе, когда будем применять такие патчи:

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

Команда format-patch создаёт файлы с патчами и выводит их названия. Опция - M сообщает Git’у о необходимости отслеживания переименований файлов. Итоговые патчи выглядят так:

```
                    $ 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
```

```
                                        --

                               1.6.2.rc1.20.g8c5b.dirty
```

Вы также можете отредактировать эти файлы с патчами, чтобы добавить в электронное письмо какую-то информацию, которую вы не хотите показывать в сообщении коммита. Если вы добавите текст между строкой -- и началом патча (строка lib/simplegit.rb), то разработчик сможет его прочитать, а при применении патча он будет выброшен.

Чтобы отправить эти файлы в список рассылки, вы можете либо вставить файл в своём почтовом клиенте, либо отправить его через специальную программу из командной строки. Вставка текста часто приводит к ошибкам форматирования, особенно в «умных» клиентах, которые не сохраняют символы перевода строки и пробельные символы в исходном виде. К счастью, Git предоставляет инструмент, позволяющий вам передавать через IMAP правильно отформатированные патчи. Для вас применение этого инструмента может оказаться более простым. Я покажу как отсылать патчи через Gmail, так как именно этот агент я и использую; вы можете прочесть подробные инструкции для множества почтовых программ в вышеупомянутом файле Documentation/SubmittingPatches, находящемся в исходном коде Git'а.

Для начала нам необходимо настроить секцию imap в файле ~/.gitconfig. Можете добавить все значения по одному несколькими командами git config, или можете добавить их все сразу вручную; но в итоге ваш файл конфигурации должен выглядеть примерно так:

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

Если ваш IMAP-сервер не использует SSL, две последние строки могут отсутствовать, а параметр host примет значение imap:// вместо imaps://. Когда закончите с настройками, воспользуйтесь командой git send-email, чтобы поместить свою серию патчей в папку Drafts на указанном IMAP-сервере:

```
$ 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
```

Затем Git выдаёт кучу служебных сообщений, которые для каждого отсылаемого патча выглядят следующим образом:

```
(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>
```

Если всё прошло успешно, то сейчас вы можете перейти в свою папку Drafts, изменить поле ‘To’ на адрес списка рассылки, в который вы собираетесь послать патчи,

возможно, указать адрес мейнтейнера или лица отвечающую за нужную часть проекта в поле ‘CC’ и отправить сообщение.

Итоги

В этом разделе мы рассмотрели ряд общепринятых рабочих процессов, применяемых в разных типах проектов использующих Git, с которыми вы наверняка столкнётесь. Также были представлены несколько новых инструментов, призванных помочь вам в организации этих процессов. Далее мы рассмотрим, как осуществляется работа с противоположной стороны баррикады — как сопровождать проект использующий Git. Вы научитесь роли благосклонного диктатора или роли менеджера по интеграции.

Сопровождение проекта

В дополнение к тому, как эффективно работать над проектом, вам, наверняка, необходимо также знать как самому поддерживать проект. Сопровождение проекта может заключаться в принятии и применении патчей, сгенерированных с помощью ‘format-patch’ и отправленных вам по почте, или в интеграции изменений из веток тех репозиториев, которые вы добавили в качестве удалённых (remotes) для вашего проекта. Неважно, поддерживаете ли вы эталонный репозиторий проекта или хотите помочь с проверкой и утверждением патчей, вам необходимо выработать метод приёма наработок, который будет наиболее понятным для других участников и не будет изменяться в течении длительного срока.

Работа с тематическими ветками

Если вы решаете интегрировать ли новые наработки, как правило неплохо было бы опробовать их в какой-нибудь временной тематической ветке, специально созданной для их тестирования.

Так будет легче подправить отдельные патчи или забросить их до лучших времён, если что- то не работает. Если вы дадите ветке простое имя, основанное на теме работы содержащейся в ней, например, ruby_client, или как-нибудь так же наглядно, то вы сможете легко вспомнить, для чего эта ветка, если вам вдруг придётся отложить работу с ней и вернуться к ней позднее. В проекте Git мейнтейнер, как правило, создаёт ветки с добавлением пространства имён — к примеру, ‘sc/ruby_client’, где ‘sc’ — это сокращённое имя автора, приславшего свою работу. Как вы уже знаете, создать ветку, основанную на вашей ветке master, можно следующим образом:

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

Или, если вы хотите сразу переключиться на создаваемую ветку, можно воспользоваться командой checkout -b:

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

Теперь вы готовы к тому, чтобы принять изменения в данную тематическую ветку и определить, хотите ли вы влить их в свои стабильные ветки или нет.

Применение патчей, отправленных по почте

Если вы получили по электронной почте патч, который вам нужно интегрировать в свой проект, вам необходимо применить патч в тематической ветке, чтобы его оценить. Есть два способа применения отправленных по почте патчей: с помощью команды git applyили команды git am.

Применение патчей с помощью команды apply Если вы получили чей-то патч, сгенерированный с помощью команды gitdiffили Unix-команды diff, вы можете применить его при помощи команды git apply. Полагая, что вы сохранили патч в /tmp/patch-ruby-client.patch, вы можете применить его следующим образом:

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

Эта команда внесёт изменения в файлы в рабочем каталоге. Она практически идентична выполнению команды patch -p1 для применения патча, хотя она более параноидальна и допускает меньше нечётких совпадений, чем patch. К тому же она способна справиться с добавлением, удалением и переименованием файлов, описанными в формате git diff, чего команда patchсделать не сможет. И наконец git applyреализует модель «применить всё или ничего», тогда как patch позволяет частично применять патч-файлы, оставляя ваш рабочий каталог в странном и непонятном состоянии. Команда git applyв целом гораздо более параноидальна, чем patch. Она не создаст для вас коммит — после выполнения команды вы должны вручную проиндексировать внесённые изменения и сделать коммит.

Кроме того, вы можете использоваться git apply, чтобы узнать, чисто ли накладывается патч, ещё до того, как вы будете применять его на самом деле — для этого выполните git ap- ply --check, указав нужный патч:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
           error: patch failed: ticgit.gemspec:1

                    error: ticgit.gemspec: patch does not apply
```

Если никакого вывода нет, то патч должен наложиться без ошибок. Если проверка прошла неудачно, то команда завершится с ненулевым статусом, так что вы можете использовать её при написании сценариев.

Применение патчей с помощью команды am Если разработчик является достаточно хорошим пользователем Git и применил команду format-patch для создания своего патча, то ваша задача становится проще, так как такой патч содержит информацию об авторе и сообщение коммита. Если есть возможность, поощряйте участников проекта на использование команды format-patch вместо diff при генерировании патчей для вас. Команду git apply стоит использовать, только если нет другого выхода, и патчи уже созданы при помощи diff.

Чтобы применить патч, созданный при помощи format-patch, используйте команду git am. С технической точки зрения, git am читает mbox-файл, который является простым текстовым форматом для хранения одного или нескольких электронных писем в одном текстовом файле. Он выглядит примерно следующим образом:

```
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
```

Это начало вывода команды format-patch, который мы уже видели в предыдущем разделе. Это одновременно и правильный mbox формат для e-mail. Если кто-то прислал вам по почте патч, правильно воспользовавшись для этого командой git send-email, и вы сохранили это сообщение в mbox-формате, тогда вы можете указать этот mbox-файл команде

git am — в результате команда начнёт применять все патчи, которые найдёт. Если вы пользуетесь почтовым клиентом, способным сохранять несколько электронных писем в один mbox-файл, то можете сохранить всю серию патчей в один файл и затем использовать команду git am для применения всех патчей сразу.

Однако, если кто-нибудь загрузил патч, созданный через format-patch, в тикет-систему или что-либо подобное, вы можете сохранить файл локально и затем передать его команде git am, чтобы его наложить:

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

Как видите, патч был применён без ошибок и за вас автоматически создан новый коммит. Информация об авторе берётся из полей From и Date письма, а сообщение коммита извлекается из поля Subject и тела (до начала самого патча) электронного письма. Например, если применить патч из mbox-файла приведённого выше примера, то созданный для него коммит будет выглядеть следующим образом:

```
                              $ 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
```

В поле Commit указан человек, применивший патч, а в CommitDate — время его применения.

Информация Author определяет человека, создавшего патч изначально, и время его создания. Однако возможна ситуация, когда патч не наложиться без ошибок. Возможно ваша основная ветка слишком далеко ушла вперёд относительно той, на которой патч был основан, или этот патч зависит от другого патча, который вы ещё не применили. В этом случае выполнение команды git amбудет приостановлено, а у вас спросят, что вы хотите сделать:

```
$ 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".
```

Эта команда выставляет отметки о конфликтах в каждый файл, с которым возникают проблемы, точно так же, как это происходит при операции слияния или перемещения с конфликтами. И разрешается данная ситуация тем же способом — отредактируйте файл, чтобы разрешить конфликт, добавьте новый файл в индекс, а затем выполните команду git am --resolved, чтобы перейти к следующему патчу:

```
                                  $ (исправление файла)

                                $ git add ticgit.gemspec

                                  $ git am --resolved

                             Applying: seeing if this helps the gem
```

Если вы хотите, чтобы Git постарался разрешить конфликт более умно, воспользуйтесь опцией -3, при которой Git попытается выполнить трёхходовую операцию слияния. Эта опция не включена по умолчанию, так как она не работает в случае, если коммита, на котором был основан патч, нет в вашем репозитории. Если этот коммит всё же

у вас есть — в случае, когда патч был основан на публичном коммите — то опция -3 как правило гораздо умнее в наложении конфликтных патчей:

```
$ 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.
```

В этом случае я пытался применить патч, который я уже применил. Без опции -3 это привело бы к конфликту.

При применении серии патчей из mbox-файла, вы также можете запустить команду am в интерактивном режиме — в этом случае команда останавливается на каждом найденном патче и спрашивает вас, хотите ли вы его применить:

```
$ 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
```

Это удобно, если у вас накопилось множество патчей, так как вы сможете сначала просмотреть патч, если вы забыли, что он из себя представляет, или отказаться применять патч, если он уже применён.

После того как вы примените все патчи по интересующей вас теме и сделаете для них коммиты в своей ветке, вы можете принять решение — интегрировать ли их в свои стабильные ветки и если да, то каким образом.

Проверка удалённых веток

Если к вам поступили наработки от человека, использующего Git и имеющего свой собственный репозиторий, в который он и отправил свои изменения, а вам он прислал ссылку на свой репозиторий и имя удалённой ветки, в которой находятся изменения, то вы можете добавить его репозиторий в качестве удалённого и выполнить слияния локально.

Например, если Джессика присылает вам письмо, в котором говорится, что у неё есть классная новая функция в ветке ruby-client в её репозитории, вы можете протестировать

её, добавив её репозиторий в качестве удалённого для вашего проекта и выгрузив содержимое этой ветки в рабочий каталог:

```
$ git remote add jessica git://github.com/jessica/myproject.git

$ git fetch jessica

$ git checkout -b rubyclient jessica/ruby-client
```

Если она снова пришлёт вам письмо с другой веткой и с новой замечательной функцией, вы сможете сразу извлечь эти наработки и переключиться на эту ветку, так как её репозиторий уже прописан в ваших удалённых репозиториях.

Этот метод наиболее удобен, если вы работаете с человеком постоянно. Если кто-то изредка представляет вам по одному патчу, то менее затратно по времени будет принимать их по e-mail, чем заставлять всех иметь свои собственные репозитории и постоянно добавлять и удалять удалённые репозитории, чтобы получить пару патчей. Также вы скорее всего не захотите иметь у себя сотни удалённых репозиториев — для всех, кто предоставил вам один или два патча. Хотя сценарии и функции хостингов могут упростить эту ситуацию — всё зависит от того, как ведёте разработку вы и участники вашего проекта.

Другим преимуществом данного подхода является тот факт, что вы получаете не только патчи, но и историю коммитов. Если вы даже обнаружите проблемы со слиянием, то вы по крайней мере будете знать, на каком коммите в вашей истории основана их работа. Правильное трёхходовое слияние в этом случае используется по умолчанию, что лучше, чем передать -3 и надеяться, что патч был сгенерирован на основе публичного коммита, к которому у вас есть доступ.

Если вы не работаете с человеком постоянно, но всё же хотите принять его изменения таким способом, можете указать URL его удалённого репозитория команде git pull. Так вы получите нужные изменения, а URL не будет сохранён в списке удалённых репозиториев:

```
$ git pull git://github.com/onetimeguy/project.git
     From git://github.com/onetimeguy/project

 * branch             HEAD       -> FETCH_HEAD
          Merge made by recursive.
```

Определение вносимых изменений

Сейчас у вас есть тематическая ветка, содержащая наработки участников проекта. На этом этапе вы можете определить, что бы вы хотели с ними сделать. В этом разделе мы снова

рассмотрим несколько команд, которые, как вы увидите, можно использовать для точного определения того, что вы собираетесь слить в свою основную ветку.

Часто полезно просмотреть все коммиты, которые есть в этой ветке, но нет в вашей ветке master. Исключить коммиты из ветки masterможно добавив опцию --notперед именем ветки. Например, если участник вашего проекта прислал вам два патча, и вы создали ветку с именем contribи применили эти патчи в ней, вы можете выполнить следующее:

```
$ 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
```

Чтобы увидеть какие изменения вносит каждый коммит, если помните, можно передать опцию -pкоманде git log— к каждому коммиту будет добавлен его diff.

Чтобы посмотреть полный diff того, что добавится при слиянии вашей тематической ветки с другой веткой, вам может понадобиться использовать странный трюк, чтобы получить нужный результат. Вы, возможно, решите выполнить такую команду:

```
$ git diff master
```

Эта команда выведет вам diff, но результат может ввести вас в заблуждение. Если ваша ветка master была промотана вперёд с того момента, когда вы создали на её основе тематическую ветку, вы, наверняка, увидите странный результат. Это происходит по той причине, что Git напрямую сравнивает снимок состояния последнего коммита тематической ветки, на которой вы находитесь, и снимок последнего коммита ветки master. Например, если вы добавили строку в файл в ветке master, прямое сравнение снимков покажет, что изменения в тематической ветке собираются эту строку удалить.

Если masterявляется прямым предком вашей тематической ветки, то проблем нет. Но если две линии истории разошлись, то diff будет выглядеть так, будто вы добавляете всё новое из вашей тематической ветки и удаляете всё уникальное в ветке master.

То, что вы действительно хотели бы видеть — это изменения, добавленные в тематической ветке, то есть те наработки, которые вы внесёте при слиянии этой ветки с веткой master. Это выполняется путём сравнения последнего коммита в вашей тематической ветке с первым общим с веткой masterпредком.

Технически, вы можете сделать это, выделив общего предка явным образом и выполнив затем команду diff:

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

$ git diff 36c7db
```

Однако это не очень удобно, так что в Git есть отдельное сокращённое обозначение для выполнения того же самого — запись с тремя точками. В контексте команды diff, вы можете поставить три точки после названия одной из веток, чтобы увидеть дельту между последним коммитом ветки, на которой вы находитесь, и их общим предком с другой веткой:

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

Эта команда покажет вам только те наработки в вашей текущей тематической ветке, которые были внесены после её ответвления от ветки master. Это очень удобный синтаксис и его надо запомнить.

Интегрирование чужих наработок

Когда все наработки в вашей тематической ветке готовы к интегрированию в более стабильную ветку, встаёт вопрос — как это сделать? Более того — какой рабочий процесс в целом вы хотите использовать, занимаясь поддержкой своего проекта? Есть множество вариантов, так что рассмотрим некоторые из них.

Процессы слияния Один из простых рабочих процессов заключается в слиянии наработок в ветку master. В этом случае ваша ветка master содержит основную стабильную версию кода. Если у вас в тематической ветке находится работа, которую вы уже доделали, или полученные от кого-то наработки, которые вы уже проверили, вы сливаете её в свою ветку master, удаляете тематическую ветку, а затем продолжаете работу. Если в вашем репозитории наработки находятся в двух ветках, названия которых ruby_client и php_client (Рисунок 5-19), и вы выполняете слияние сначала для ветки ruby_client, в потом для php_client, то ваша история коммитов в итоге будет выглядеть, как показано на Рисунке 5-20.

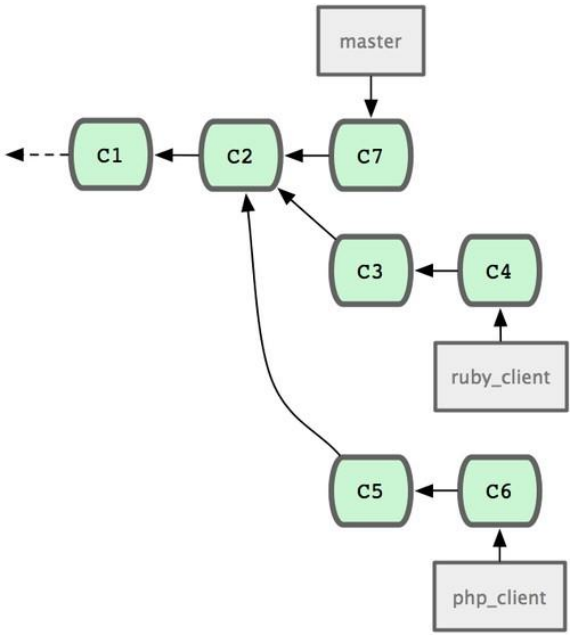

Рисунок 5.19: История коммитов с несколькими тематическими ветками.

Это, по всей видимости, наиболее простой рабочий процесс, но при работе с большими проектами здесь возникает ряд проблем.

Если ваш проект более крупный, или вы работаете с большим количеством разработчиков, вы, вероятно, будете применять по крайней мере двухэтапный цикл слияний. При этом сценарии у вас есть две долго живущие ветки, masterи develop, и вы решили, что ветка master обновляется только тогда, когда выходит очень стабильный релиз, а весь новый код включается в ветку develop. Изменения в обеих этих ветках регулярно отправляются в публичный репозиторий. Каждый раз, когда у вас появляется новая тематическая ветка для слияния (Рисунок 5-21), вы сначала сливаете её в develop (Рисунок 5-22); затем, когда вы выпускаете релиз, вы делаете перемотку (fast-forward) ветки masterна нужный стабильный коммит ветки de- velop(Рисунок 5-23).

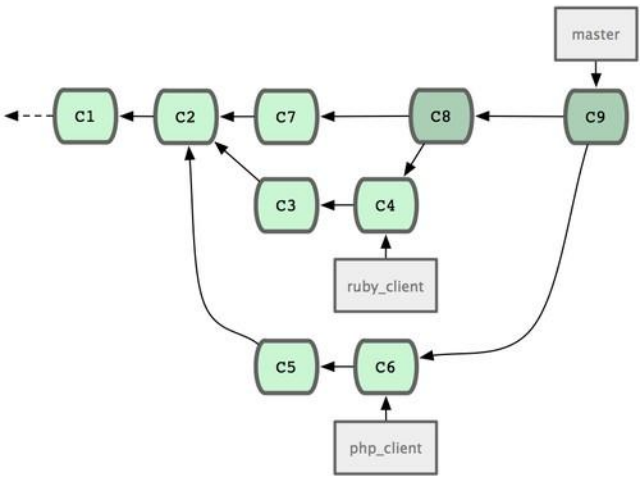

Рисунок 5.20: История коммитов после слияния тематических веток.

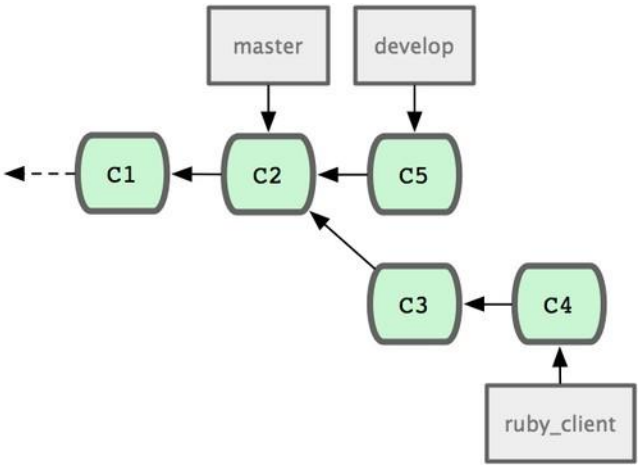

Рисунок 5.21: История коммитов до слияния тематической ветки.

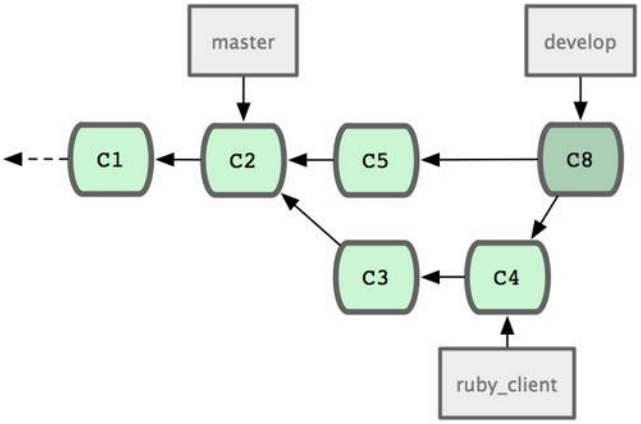

Рисунок 5.22: История коммитов после слияния тематической ветки.

При таком подходе, клонируя ваш репозиторий, люди могут либо выгрузить ветку master, чтобы получить последний стабильный релиз и легко поддерживать этот код обновлённым, либо переключиться на ветку develop, которая включает в себя всё самое свежее. Вы также можете развить данный подход, создав ветку для интегрирования, в которой будет происходить слияние всех наработок. И когда код на этой ветке станет стабилен и пройдёт все тесты, вы сольёте её в ветку develop; и если всё будет работать как надо в течение некоторого времени, вы выполните перемотку ветки master.

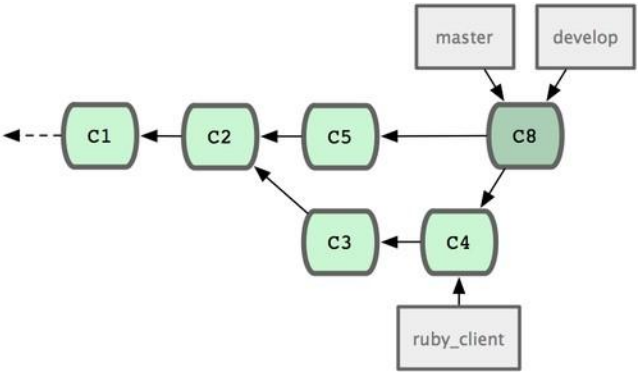

Рисунок 5.23: История коммитов после появления релиза.

Рабочие процессы с крупными слияниями Проект Git имеет четыре долго живущие ветки: master, next, pu (proposed updates) для новых наработок и maint для ретроподдержки (backports). Когда участники проекта подготавливают свои наработки, они собираются в тематических ветках в репозитории мейнтейнера проекта примерно так, как мы уже описывали (смотри Рисунок 5-24). На этом этапе проводится оценка проделанной работы — всё ли работает, как положено, стабилен ли код, или ему требуется доработка. Если всё в порядке, то тематические ветки сливаются в ветку next, которая отправляется на сервер, чтобы у каждого была возможность опробовать интегрированные воедино изменения из тематических веток.

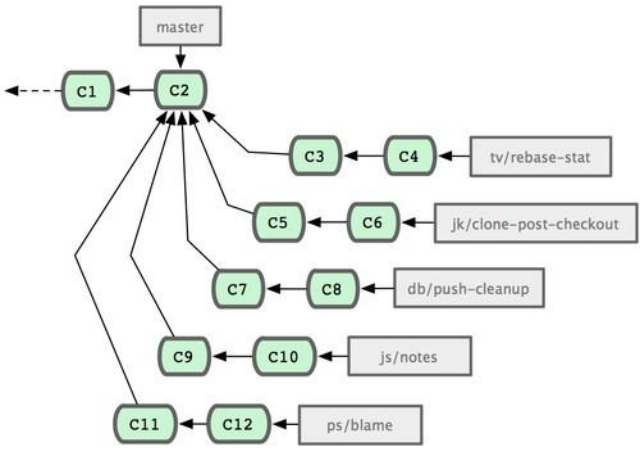

Рисунок 5.24: Управление группой параллельных тематических веток участников проекта.

Если тематические ветки требуют доработки, они сливаются в ветку pu. Когда будет установлено, что тематические ветки полностью стабильны, они переливаются в master, а ветки pu и next перестраиваются на основе тематических веток, находившихся в next, но ещё не дозревших до master. Это означает, что master практически всегда движется в прямом направлении, ветка next перемещается (rebase) иногда, а ветка pu перемещается чаще всех (смотри Рисунок 5-25).

Когда тематическая ветка была полностью слита в ветку master, она удаляется из репозитория.

В проекте Git есть ещё ветка maint, которая ответвлена от последнего релиза и предоставляет backport-патчи, на случай если потребуется выпуск корректировочной версии. Таким образом, когда вы клонируете Git-репозиторий, вы получаете четыре ветки, переключаясь на которые вы можете оценить проект на разных стадиях разработки (в зависимости от того, насколько свежую версию вы хотите получить, или от того, каким образом вы хотите внести в проект свою работу); а мейнтейнер, в свою очередь, имеет структурированный рабочий процесс, который помогает ему изучать новые присланные патчи.

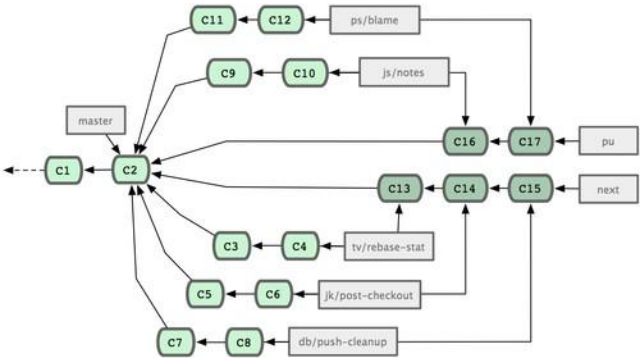

Рисунок 5.25: Слияние тематических веток участников проекта в долго живущие интеграционные ветки.

Рабочие процессы с перемещениями и отбором лучшего Другие мейнтейнеры вместо слияния предпочитают выполнять перемещение или отбор лучших наработок участников проекта на верхушку своей ветки master, чтобы иметь практически линейную историю разработки. Когда у вас есть наработки в тематической ветке, которые вы хотите интегрировать в проект, вы переходите на эту ветку и запускаете команду rebase, которая перемещает изменения на верхушку вашей текущей ветки master(или develop, и т.п.). Если всё прошло хорошо, то можете выполнить перемотку ветки master, получив тем самым линейную историю работы над проектом.

Другой вариант перемещения сделанных наработок из одной ветки в другую — отбор лучшего (cherry-pick). Отбор лучшего в Git является чем-то наподобие перемещения для отдельных коммитов. Берётся патч, который был представлен в коммите, и делается попытка применить его на ветке, на которой вы сейчас находитесь. Это удобно в том случае, если у вас в тематической ветке находится несколько коммитов, а вы хотите включить в проект только один из них, или если у вас только один коммит в тематической ветке, но вы предпочитаете выполнять отбор лучшего вместо перемещения. Например, предположим, ваш проект выглядит так, как показано на Рисунке 5-26. Если вы хотите вытащить коммит e43a6в ветку master, выполните:

```
$ 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(-)
```

Эта команда включит в ветку mastertакие же изменения, которые были добавлены в e43a6, но вы получите новое значение SHA-1 для этого коммита, так как у него будет другая дата применения. Теперь ваша история коммитов выглядит, как показано на Рисунке 5-27.

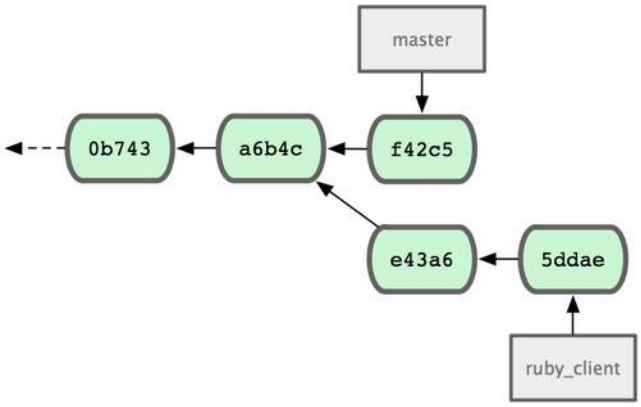

Рисунок 5.26: Пример истории коммитов перед отбором лучшего.

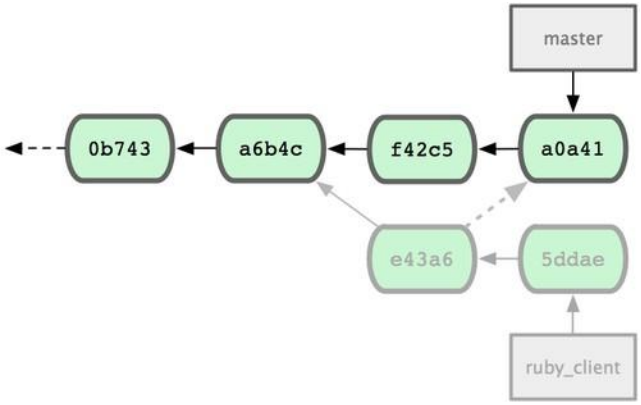

Рисунок 5.27: История коммитов после отбора лучшего коммита из тематической ветки.

Теперь вы можете удалить свою тематическую ветку и отбросить коммиты, которые вы не захотели включить в проект.

Отметка релизов

Если вы решили выпустить релиз, вы, вероятно, захотите присвоить ему метку, так чтобы вы потом смогли восстановить этот релиз в любой момент. Процесс создания новой метки обсуждался в Главе 2. Если вы решили подписать вашу метку как мейнтейнер, то процедура будет выглядеть примерно следующим образом:

```
$ 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
```

Если вы подписываете свои метки, у вас может возникнуть проблема с распространением открытого PGP-ключа, используемого для подписи ваших меток. Мейнтейнер проекта Git решил эту проблему, добавив свой публичный ключ в виде блоба (blob) прямо в репозиторий и затем выставив метку, указывающую прямо на содержимое ключа. Чтобы сделать это, определите какой ключ вам нужен, выполнив 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]
```

Затем вы можете напрямую импортировать ключ в базу данных Git'а, экспортировав его и передав по конвейеру команде git hash-object, которая создаст новый блоб с содержимым ключа и вернёт вам SHA-1 этого блоба:

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

Теперь, когда у вас в Git хранится ваш ключ, вы можете создать метку, напрямую указывающую на него, использовав значение SHA-1, возвращённое командой hash-object:

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

Если вы запустите команду git push --tags, то метка maintainer-pgp-pub станет доступна каждому. Если кто-нибудь захочет проверить какую-нибудь метку, он сможет напрямую импортировать ваш PGP-ключ, вытащив блоб прямо из базы данных и импортировав его в GPG:

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

Этот ключ может быть использован для проверки любых подписанных вами меток. Кроме того, если вы включите инструкции в сообщение метки, запуск git show <метка> позволит конечному пользователю получить инструкции по проверке меток.

Генерация номера сборки

Так как коммитам в Git не присваиваются монотонно возрастающие номера наподобие 'v123' или чего-то аналогичного, то в случае, если вы хотите присвоить коммиту имя удобное для восприятия, запустите команду git describe для этого коммита. Git вернёт вам имя ближайшей метки с числом коммитов сделанных поверх этой метки и частичное значения SHA-1 описываемого коммита:

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

Таким образом при экспорте снимка состояния проекта или его сборки вы можете дать им имя понятное для людей. На самом деле, если вы собираете Git из исходного кода, склонированного из Git-репозитория, git --version вернёт вам что-то подобное. Если вы описываете коммит, которому вы напрямую присвоили метку, команда вернёт вам имя метки.

Команду git describe хорошо использовать с аннотированными метками (метками, созданными при помощи опций -a или -s), так что если вы используете git describe, то метки для релизов должны создаваться этим способом — в этом случае вы сможете удостовериться, что при описании коммиту было дано правильное имя. Вы также можете использовать эту строку в командах checkout и show для указания нужного коммита, однако в будущем она может перестать работать правильно в силу того, что в строке присутствует сокращённое значение SHA-1. Например, в ядре Linux недавно перешли от 8 к 10 символам необходимым для обеспечения уникальности SHA-1 объектов, и поэтому старые имена, сгенерированные командой git de- scribe, стали недействительными.

Подготовка релиза

Теперь хотелось бы выпустить релиз сборки. Вероятно, вам захочется сделать архив последнего состояния вашего кода для тех бедолаг, которые не используют Git. Для этого используется команда git archive:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz

$ ls *.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Если кто-нибудь откроет этот tarball, он получит последний снимок состояния вашего проекта внутри каталога project. Таким же способом вы можете создать zip-архив, указав команде git archive опцию --format=zip:

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

Теперь у вас есть тарбол и zip-архив с релизом вашего проекта, которые вы можете загрузить на свой сайт или отправить людям по почте.

Команда shortlog

Пришло время написать письмо для списка рассылки, чтобы поделиться новостями проекта со всеми, кто им интересуется. При помощи команды git shortlog можно быстро получить что-то наподобие лога изменений (changelog), описывающего, что появилось нового в вашем проекте со времени последнего релиза или последнего письма в список рассылки. Лог изменений включает в себя все коммиты в указанном диапазоне; например, следующая команда вернёт вам сводку по всем коммитам, сделанным со времени прошлого релиза (если последний релиз имел метку v1.0.1):

```
$ 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
```

Мы получили аккуратную сводку по всем коммитам, начиная с метки v1.0.1, сгруппированным по авторам. Вывод этой команды можно послать в свой список рассылки.

Итоги

Вы должны чувствовать себя достаточно свободно, внося свой вклад в проект под управлением Git, а также занимаясь поддержкой своего собственного проекта или интегрированием наработок других пользователей. Поздравляем тебя, опытный Git-разработчик! В следующей главе вы познакомитесь с более мощными инструментами, а также получите советы по действию в сложных ситуациях, что сделает из вас настоящего мастера в Git.